John Derrick
Stefania Gnesi
Diego Latella
Helen Treharne (Eds.)

LNCS 7321

Integrated Formal Methods

9th International Conference, IFM 2012
Pisa, Italy, June 2012
Proceedings

Springer

Lecture Notes in Computer Science 7321

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

John Derrick Stefania Gnesi
Diego Latella Helen Treharne (Eds.)

Integrated Formal Methods

9th International Conference, IFM 2012
Pisa, Italy, June 18-21, 2012
Proceedings

Springer

Volume Editors

John Derrick
University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello Street
Sheffield S1 4DP, UK
E-mail: j.derrick@dcs.shef.ac.uk

Stefania Gnesi
Diego Latella
Consiglio Nazionale delle Ricerche
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Via Moruzzi 1
56124 Pisa, Italy
E-mail: {stefania.gnesi; diego.latella@isti.cnr.it}

Helen Treharne
University of Surrey
Department of Computing
Surrey GU2 7XH, UK
E-mail: h.treharne@surrey.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30728-7 e-ISBN 978-3-642-30729-4
DOI 10.1007/978-3-642-30729-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939244

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, F.4.1, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Foreword

iFM 2012, the 9th International Conference on Integrated Formal Methods, and ABZ 2012, the 3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z, were joined in a single event, iFM&ABZ 2012, to celebrate Egon Börger's 65th birthday and his contribution to state-based formal methods.

This colocation of iFM&ABZ 2012 was hosted by the Institute of Scienza e Tecnologie dell'Informazione "A. Faedo" of the National Research Council (ISTI-CNR) of Italy and took place at the Area della Ricerca del CNR in Pisa, during June 18–21, 2012.

We would like to thank everyone in Pisa for making us feel very welcome during our time there. It was a pleasure to run an event to honor Egon.

Professor Egon Börger was born in Bad Laer, Lower Saxony, Germany. Between 1965 and 1971 he studied at the Sorbonne, Paris (France), Université Catholique de Louvain and Institut Supérieur de Philosophie de Louvain (in Louvain-la-Neuve, Belgium), and the University of Münster (Germany). Since 1985 he has held a Chair in computer science at the University of Pisa, Italy. In September 2010 he was elected as member of the Academia Europaea. Throughout his work he has been a pioneer of applying logical methods in computer science. Particularly notable is his contribution as one of the founders of the Abstract State Machine (ASM) method. Egon Börger has been cofounder and Managing Director of the Abstract State Machines Research Center (see www.asmcenter.org).

Building on his work on ASM, he was a cofounder of the series of international ASM workshops, which were part of this year's conference under the ABZ banner. He contributed to the theoretical foundations of the method and initiated its industrial applications in a variety of fields, in particular programming languages, system architecture, requirements and software (re-)engineering, control systems, protocols and Web services. In 2007, he received the Humboldt Research Award.

He has been coauthor of several books and over 150 research papers; he has organized over 30 international conferences, workshops, schools in logic and computer science.

As one can see, his influence has been broad as well as deep. It is an influence that ones finds in all of the notations covered in the ABZ conference, as well as in the iFM event and the various integrations and combinations of formal methods seen therein. Neither iFM or ABZ have been here before, and it is thus especially fitting that we held such an event in Pisa, where Egon has held a Chair for many years.

In addition to contributed papers, the conference programme included two tutorials and three keynote speakers. The tutorials were offered by: Eric C.R. Hehner on "Practical Predicative Programming Primer" and Joost-Pieter Katoen, Thomas Noll, Alessandro Cimatti and Marco Bozzano on "Safety, Dependability and Performance Analysis of Extended AADL Models." We are grateful to Egon Börger, Muffy Calder and Ian J. Hayes for accepting our invitation to address the conference.

Each conference, ABZ and iFM, had its own Programme Committee Chairs and Programme Committees, and we leave it to them to describe their particular conference. We shared invited speakers, so all conference attendees had the opportunity to hear Egon, Muffy and Ian. We also shared some technical sessions so that all participants could see some of the best technical work from each conference.

We would like to thank the Programme Committee Chairs, Diego Latella, CNR/ ISTI, Italy and Helen Treharne, University of Surrey, UK for iFM 2012; Steve Reeves, University of Waikato, New Zealand and Elvinia Riccobene, University of Milan, Italy for ABZ 2012 for their efforts in setting up two high-quality conferences.

We also would like to thank the members of the Organizing Committee as well as several other people whose efforts contributed to making the conference a success and particular thanks go to the Organizing Committee Chair Maurice ter Beek.

June 2012

John Derrick
Stefania Gnesi

Preface

This volume contains the proceedings of iFM 2012, the 9th International Conference on Integrated Formal Methods, held during June 18–21, 2012, in Pisa, Italy, jointly with ABZ 2012, the 3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z, in honor of Egon Börger's 65th birthday. The ABZ proceedings appear as a separate LNCS volume, number 7316. The invited talk of Egon Börger appears in both proceedings.

The iFM conference programme also included an invited talk by Muffy Calder and the ABZ conference programme included an invited talk by Ian Hayes.

Previous iFM conferences were held in York, Dagstuhl, Turku, Canterbury, Eindhoven, Oxford, Düsserldorf and Nancy. The iFM conference series seeks to further research into the combination of different formal methods for modelling and analysis. However, the work of iFM goes beyond that, covering all aspects from language design, verification techniques, tools and the integration of formal methods into software engineering practice.

iFM 2012 received 59 submissions, covering the spectrum of integrated formal methods, ranging across formal and semi-formal modelling notations, semantics, proof frameworks, refinement, verification, timed systems, tools and case studies. Each submission was reviewed by at least three Programme Committee members. The committee decided to accept 22 papers.

The conference was preceded by a day dedicated to tutorials on "Practical Predicative Programming Primer" by Eric C. R. Hehner and "Safety, Dependability and Performance Analysis of Extended AADL Models" by Joost-Pieter Katoen, Thomas Noll, Alessandro Cimatti and Marco Bozzano.

We are grateful to the members of the Programme Committee and the external reviewers for their diligence and thoroughness. We also appreciate the support of EasyChair for managing the reviewing process and the preparation of the proceedings. We thank all those involved in organizing the conference and an important note of thanks must be extended to the members of CNR who helped locally.

June 2012

Diego Latella
Helen Treharne

Organization

General Chairs

John Derrick	University of Sheffield, UK
Stefania Gnesi	CNR/ISTI, Italy

Conference Chairs

Diego Latella	CNR/ISTI, Italy
Helen Treharne	University of Surrey, UK

Programme Committee

Marc Benveniste	STMicroelectronics, France
Eerke Boiten	University of Kent, UK
Jonathan P. Bowen	Museophile Limited, UK
Jim Davies	University of Oxford, UK
John Derrick	Unversity of Sheffield, UK
Jin Song Dong	National University of Singapore, Singapore
Kerstin Eder	University of Bristol, UK
Alessandro Fantechi	University of Florence and CNR/ISTI, Italy
John Fitzgerald	Newcastle University, UK
Andy Galloway	University of York, UK
Einar Broch Johnsen	University of Oslo, Norway
Rajeev Joshi	NASA Jet Propulsion Laboratory (JPL), USA
Michael Leuschel	University of Düsseldorf, Germany
Michele Loreti	University of Florence, Italy
Silvia Mazzini	Intecs S.p.A, Italy
Dominique Mery	LORIA and Université de Lorraine, France
Stephan Merz	INRIA Nancy and LORIA, France
Alexandre Mota	Centre of Informatics (CIn-UFPE), Brazil
Flemming Nielson	Technical University of Denmark and MT-LAB, Denmark
Luigia Petre	Åbo Akademi University, Finland
David Pichardie	INRIA Rennes, France
Thomas Santen	European Microsoft Innovation Center, Germany
Steve Schneider	University of Surrey, UK
Kaisa Sere	Åbo Akademi University, Finland
Graeme Smith	University of Queensland, Australia
Kenji Taguchi	National Institute of Advanced Industrial Science and Technology, Japan

Mirco Tribastone	Ludwig-Maximilians-Universität, Germany
Marina Waldén	Åbo Akademi University, Finland
Heike Wehrheim	University of Paderborn, Germany
Kirsten Winter	University of Queensland, Australia

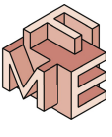
Additional Reviewers

Islam Abdel Halim	Carl Gamble	Ling Shi
Étienne André	Juliano Iyoda	Mihaela Sighireanu
Emilie Balland	Mohammad Mahdi Jaghoori	Tarciana Silva
Sebastien Bardin	Maryam Kamali	Neeraj-Kumar Singh
Cristiano Bertolini	Weiqiang Kong	Songzheng Song
Lorenzo Bettini	Linas Laibinis	Dominik Steenken
Irene Bicchierai	Shang-Wei Lin	Volker Stolz
Jean-Paul Bodeviex	Alberto Lluch Lafuente	Anton Tarasyuk
Carl Friedrich Bolz	Acciai Lucia	Maurice ter Beek
Pontus Boström	Toby Murray	Francesco Tiezzi
Jeremy W. Bryans	Keishi Okamoto	Max Tschaikowski
Laura Carnevali	Richard Payne	Leonidas Tsiopoulos
Márcio Cornélio	Stefano Pepi	Sven Walther
Fredrik Degerlund	Ken Pierce	Daniel Wonisch
Delphine Demange	Steve Riddle	Yoriyuki Yamagata
Andre Didier	Petter Sandvik	Shaojie Zhang
Johan Dovland	Rudolf Schlatte	Manchun Zheng
Neil Evans	Alexander Schremmer	Steffen Ziegert
Marc Fontaine		

Sponsoring Institutions

A final note of appreciation to our sponsors:

INTECS S.p.A.



Formal Methods Europe

Banca Nazionale del Lavoro S.p.A.

EATCS Italian Chapter

We particularly would like to thank *Formal Methods Europe* (FME), since it is due to their generous support that were able to invite Muffy Calder for a keynote presentation.

Table of Contents

Contribution to a Rigorous Analysis of Web Application Frameworks ... 1
Egon Börger, Antonio Cisternino, and Vincenzo Gervasi

Process Algebra for Event-Driven Runtime Verification: A Case Study of Wireless Network Management ... 21
Muffy Calder and Michele Sevegnani

Translating TLA$^+$ to B for Validation with ProB ... 24
Dominik Hansen and Michael Leuschel

Rely/Guarantee Reasoning for Teleo-reactive Programs over Multiple Time Bands ... 39
Brijesh Dongol and Ian J. Hayes

Safety and Line Capacity in Railways – An Approach in Timed CSP ... 54
Yoshinao Isobe, Faron Moller, Hoang Nga Nguyen, and Markus Roggenbach

Refinement-Based Development of Timed Systems ... 69
Jesper Berthing, Pontus Boström, Kaisa Sere, Leonidas Tsiopoulos, and Jüri Vain

Analysing and Closing Simulation Coverage by Automatic Generation and Verification of Formal Properties from Coverage Reports ... 84
Tim Blackmore, David Halliwell, Philip Barker, Kerstin Eder, and Naresh Ramaram

Model Checking as Static Analysis: Revisited ... 99
Fuyuan Zhang, Flemming Nielson, and Hanne Riis Nielson

Formal Verification of Compiler Transformations on Polychronous Equations ... 113
Van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Paul Le Guernic, and Loïc Besnard

Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples ... 128
Herbert Rocha, Raimundo Barreto, Lucas Cordeiro, and Arilo Dias Neto

MULE-Based Wireless Sensor Networks: Probabilistic Modeling and Quantitative Analysis ... 143
Fatemeh Kazemeyni, Einar Broch Johnsen, Olaf Owe, and Ilangko Balasingham

Mechanized Extraction of Topology Anti-patterns in Wireless Networks 158
Matthias Woehrle, Rena Bakhshi, and Mohammad Reza Mousavi

A Proof Framework for Concurrent Programs 174
Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen

A UTP Semantics of pGCL as a Homogeneous Relation 191
Riccardo Bresciani and Andrew Butterfield

Behaviour-Based Cheat Detection in Multiplayer Games with Event-B 206
HaiYun Tian, Phillip J. Brooke, and Anne-Gwenn Bosser

Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems 221
Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu

Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B 237
Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis

Partially-Supervised Plants: Embedding Control Requirements in Plant Components 253
Jasen Markovski, Dirk A. van Beek, and Jos Baeten

Early Fault Detection in Industry Using Models at Various Abstraction Levels 268
Jozef Hooman, Arjan J. Mooij, and Hans van Wezep

PE-KeY: A Partial Evaluator for Java Programs 283
Ran Ji and Richard Bubel

Specification-Driven Unit Test Generation for Java Generic Classes 296
Francisco Rebello de Andrade, João P. Faria, Antónia Lopes, and Ana C.R. Paiva

Specifying UML Protocol State Machines in Alloy 312
Ana Garis, Ana C.R. Paiva, Alcino Cunha, and Daniel Riesco

Patterns for a Log-Based Strengthening of Declarative Compliance Models 327
Dennis M.M. Schunselaar, Fabrizio Maria Maggi, and Natalia Sidorova

A Formal Interactive Verification Environment for the Plan Execution Interchange Language 343
Camilo Rocha, Héctor Cadavid, César Muñoz, and Radu Siminiceanu

Author Index 359

Contribution to a Rigorous Analysis of Web Application Frameworks

Egon Börger, Antonio Cisternino, and Vincenzo Gervasi

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
{boerger,cisterni,gervasi}@di.unipi.it

Abstract. We suggest an approach for accurate modeling and analysis of web application frameworks.

1 Introduction

In software engineering the term 'application' traditionally refers to a specific program or process users can invoke on a computer. The emergence of distributed systems and in particular of web applications has significantly changed this meaning of the term. Here functionality is provided by a set of indipendent cooperating modules with a distributed state, in web applications all offering a unified interface to their user—to the point that the user may have no way to distinguish whether a single application or a set of distributed web applications is used. Also recent non-web systems, like mobile apps, follow the same paradigm allowing the state of an application to be persistent and distributed, no longer tied to the traditional notion of operating system process and memory.

There is still no precise general definition or model of what a web application is. What is there is a variety of (often vague and partly incompatible) standards, web service description languages at different levels of abstraction (like BPEL, BPMN, workflow patterns, see [9] for a critical evaluation of the latter two) and difficult to compare techniques, architectures and frameworks offered for implementations of web applications, ranging from CGI (Common Gateway Interface [23]) scripts to PHP (Personal Home Page) and ASP (Application Server Page) applications and to frameworks such as ASP.NET [19] and Java Server Faces (JSF [1]). All of them seem to share that a web application consists of a dynamically changing network of systems that send and receive through the HTTP protocol data to and from other components and provide services of all kinds which are subject to continuous change (as services may become temporarily or permanently unavailable), to dynamic interference with other services (competing for resources, suffering from overload, etc.) and to all sorts of failures and attacks.

The challenge we see is to discover and formulate the pattern underlying such client-server architectures for (programming and executing concurrent distributed) web applications. We want to make their common structural aspects

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 1–20, 2012.

explicit by defining precise high-level (read: code, platform and framework independent) models for the main components of current web application systems such that the major currently existing implementations can be described as refinements of the abstract models. The goal of such a rational reconstruction is to make a rigorous mathematical analysis of web applications possible, including to precisely state and analyze the similarities and differences among existing frameworks, e.g. the similarities between PHP and ASP and the differences between PHP/ASP and JSP/ASP.NET. This has three beneficial consequences: a) it helps web application analysts to better understand different technologies before integrating them to make them cooperate; b) it builds a foundation for content-based certifiability of properties one would like to guarantee for web applications; c) it supports teachers and book authors to provide an accurate organic birds' perspective of a significant area of current computer technology.

For the present state of the art, given the lack of rigorous abstract models of (at least the core components of) web application frameworks, it is still a theoretical challenge to analyze, evaluate and classify web application systems along the lines of fundamental behavioral model properties which can be accurately stated and verified and be instantiated and checked for implementations.

The modeling concepts one needs to work on the challenge become clear if we consider the above mentioned feature all web applications have in common, namely to be an application whose interface is presented to the user via a web browser, whose state is split between a client and a server and where the only interaction between client and server is through the HTTP protocol. This implies that an attempt to abstractly model web application frameworks must define at least the following two major client-server architecture components with their subcomponents and the communication network supporting their interaction:

- the browser with all its subcomponents: launcher, netreader, (html, script, image) parsers, script interpreter, renderer, etc.
- the server with its modules providing runtimes of various programming languages (e.g. PHP, Python [2], ASP, ASP.NET, JSF),
- the asynchronous network which supports the interaction (in particular the communication) between the components.

This calls for a modeling framework with the following features:

- A notion of *agents* which execute each their (possibly dynamically changing) program concurrently, possibly at different sites.
- A notion of *abstract state* covering design and analysis at different levels of abstraction (to cope with heterogeneous data structures of the involved components) and the distributed character of the state of a web application.
- A sufficiently general *refinement method* to controllably link (using validation and/or verification) the different levels of abstraction, specifically to formulate different existing systems as instances of one general model.
- A flexible mechanism to express forms of *non-determinism* which can be restricted by a variety of constraints, e.g. by different degrees of transmission

reliability ranging from completely unreliable (over the internet) to safe and secure (like for components running on one isolated single machine).
- A flexible *environment adaptation mechanism* to uniformly describe web application executions modulo their dependence on run-time contexts.
- A smooth *support for traceable model change* and refinement changes due to changing requirements in the underlying (often de facto) standards.

1.1 Concrete Goals and Results So Far

As a first step towards the goal outlined above we started to model the client-server architecture of a browser interacting with a web server. In [17] the transport and stream levels of an abstract web browser model are defined. To this we add here models for the main components of the context level layer (Sect. 2) which together with the web server model defined in Sect. 3 allow one to describe one complete round of the Request-Reply pattern [18,8] that characterizes browser/server interactions (see Fig. 1).[1] In Sect. 3.1 a high-level functional Request-Reply web server view is defined which is then detailed (by refinement steps) for the two main approaches to module execution:

- the CGI-approach where the server delegates the execution of an external process to another agent (Sect. 3.3),
- the script-approach where the server itself executes script code (Sect. 3.4).

We explain how one can view existing implementations as instantiations of these models.

We use the ASM (Abstract State Machines) method [12] as modeling framework because it offers all the features listed above which are needed for our endeavor[2] and because various ASM models in the literature contribute specifically to the work undertaken here. For example both the browser and the server model use a third group of basic components, namely ScriptInterpreters for various Script languages, which can be specified by an ASM model adopting the method used in [22] to define an interpreter for Java (and reused in [11,15,16] to rigorously define the semantics of C# and the CLR). These models provide a significant part of the infrastructure web applications typically use. For example applets which run inside a browser, or the Tomcat application server [3], are written in Java. Furthermore, the method developed for modeling Java/JVM can be reused to define a model for the JavaScript interpreter (see [14] for some details) corresponding to the ECMAScript standard ECMA-262 [4], a standard that serves as glue to link various technologies together.

In Sect. 4 we list some verification goals we suggest to pursue on the basis of (appropriately completed) precise abstract models of web application framework components, i.e. to rigorously formulate and check (verify or falsify) properties of interest for the models and/or their implementations.

[1] In the Request-Reply pattern of two-way conversations the requestor (one application) sends a request to the provider (another application) and the provider returns a reply to the requestor.

[2] See [10] for the recent definition of a simple flexible *ambient ASM* concept.

The models we define and their properties we discuss come without any completeness claim and are intended to suggest an approach we consider to be promising for future FM research in a core area of computer technology.

2 Modeling Browser Components

Our browser models focus on those parts of the browser behaviour that are most relevant for the deployement and execution of web applications. The models are developed at four layers. The main components of the *transport layer* (expressing the TCP/IP communication via HTTP) and the *stream layer* (describing how information coming from the network is received and interpreted) are defined in [17]. In this section we add models for characteristic components of the *context layer*, which deals with the user interaction with the document represented by the Document Object Model (DOM). Without loss of generality we omit in this paper the *browser layer* where the behaviour of a web browser seen as an application of the host operating system is described. In practice, most web applications are entirely contained in a single browsing context; in fact an important issue in the development of web standards is how to ensure for security reasons that multiple browsing contexts in the same browser are sufficiently isolated from each other (a security property that we leave to future work).

2.1 Browsing Context

A *browsing context* is an environment in which documents are shown to the user, and where interaction with the user occurs. In web browsers, browsing contexts are usually associated with windows or tabs, but certain deprecated HTML structures (namely, frames) also introduce separate browsing contexts.

In our model, a browsing context is characterized primarily by five elements:

- a *document* (i.e. a DOM as described in [17]), which is the currently active document presented to the user;
- a *session history*, which is a navigable stack of documents the user has visited in this browsing context;
- a *window*, which is a designated operating system-dependent area where the Document is presented and where any user interaction takes place;
- a *renderer*, which is a component that produces a user-visible graphical rendering of the current Document (Section 2.2);
- an *event loop*, which is a component that receives and processes in an ordered way the various operating system-supplied events (such as user interaction or timer expiration) that serve as local input to the browser (Section 2.3).

We keep the *window* abstract, as its behaviour can be conveniently hidden by keeping the actual rendering abstract and by assuming that user interaction with the window is handled by the operating system. Thus we deal with events that have been already pre-processed by a window manager. We also omit the rather straightforward modeling of the *session history*.

When STARTing a newly created *B*rowsing *C*ontext k, $DOM(k)$ is initialized by a pre-defined implementation-dependent initial document *initialDOM*; it is usually referred to through the URL `about:blank` and may represent an empty page or a "welcome page" of some sort. Two agents are equipped with programs to execute the RENDERER and the EVENTLOOP for k.

STARTBC(k) =
  let a =**new** *Agent*, b =**new** *Agent* **in**
    $program(a)$:= RENDERER(k)
    $program(b)$:= EVENTLOOP(k)
    $DOM(k)$:= *initialDOM*

The RENDERER and EVENTLOOP macros are specified below.

2.2 Renderer

The RENDERER produces the user interface of the current *DOM* in the (implicit) given window. It is kept abstract by specifying only that it works when it is (a) supposed to perform (at system dependent *RenderingTime*) and (b) allowed to perform because no other agent has a lock on the *DOM* (e.g., while adding new nodes to the DOM during the stream-level loading of an HTML page).

RENDERER(k) =
  if $renderingTime(k)$ **and** $\neg locked(DOM(k))$ **then**
    GENERATEUI($DOM(k), k$)

2.3 Event Loop

We assume that *events* are communicated by the host environment (i.e., the specific operating system and UI toolkit of the client machine where the browser is executed) to the browser by means of an *event queue*. These UI events are merged and put in sequential order with other events that are generated in the course of the computation, e.g. DOM manipulation events (fired whenever an operation on the DOM, caused by user actions or by Javascript operations, leads to the execution of a Javascript handler or similar processing) or History traversal events (fired whenever a user operates on the Back and Forward buttons offered by most browsers to navigate through the page stack).

Here we detail the basic mechanism used in (the simplest form of) web applications to prepare a Request to be sent to the server (with the understanding that when a Response is received, it will replace the current page in the same browsing context). HTML *forms* are used to collect related data items, usually entered by the user, and to package them in a single Request. Figure 1 shows when the macros defined below and in [17] are invoked; lifelines represent agents executing a rule. Remember that ASM agents can change their program dynamically (e.g., when RECEIVE becomes HTMLPROC) and that operations by an agent in the same activation, albeit shown in sequence, happen in parallel.

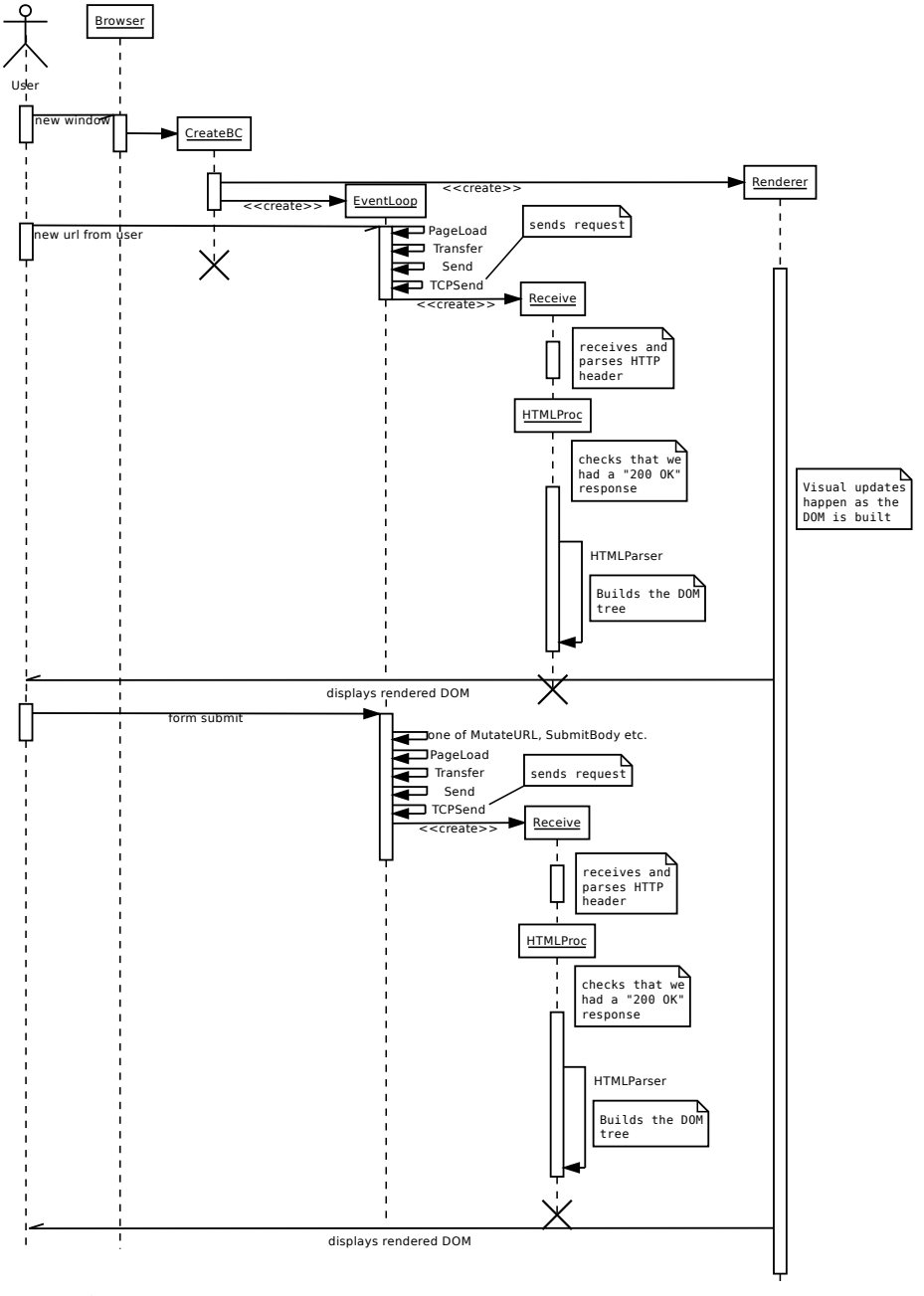


Fig. 1. A diagram depicting the behaviour of our browser model for a user who opens a new window in a browser, manually loads the first page of a web application, interacts locally with a form, and then sends the data back to the server, receiving a new or updated page in response

An HTML form is introduced by a `<FORM>` element in the page. All the input elements[3] that appear in the subtree of the DOM rooted at the `<FORM>` are said to belong to that form. Among the various input elements, there is normally a designated one (whose UI representation is often an appropriately labeled button) tasked with the function of *submitting* a form. This involves collecting all the data elements in the form, encoding them in an appropriate format, and sending them to a destination server through various means. This may include sending the data by email or initiating an FTP transfer, although these possibilities are seldom, if ever, used in contemporary web applications.

It is also of interest to note that submission of a form may be initiated from a script, by invoking the `submit()` method of the form object, and hence happen indipendently from user behaviour. In the following, we will not concern ourselves with the details of how a submit operation has been initiated, but only with the emergence of the submit event in the event queue, whatever its origin.

We model the existence of a separate event queue for each browsing context, which is processed by a dedicated agent created in the StartBC macro above. When an event is extracted from the event queue that indicates that the user has provided a new URL to load (e.g., by typing it in a browser's address bar, or by selecting an entry from a bookmarks list, etc.), the browsing context is navigated to the provided URL by starting an asynchronous transfer (in the normal case, the HTTP Request will be sent to the host mentioned in the URL, and later processing of the Response will replace the DOM displayed in the page).

When an event is extracted from the event queue that indicates a form submission, the form and related parameters are extracted from the event, appropriate encoding of the data is performed based on the action and method attributes as specified in the `<FORM>` node, and finally either the data is sent out (e.g., in the case of a `mailto:` action) or the browsing context is populated with the results returned from a web server identified by the form's action. In normal usage, that will be the same web server hosting the web application that originally sent out the page with the form, thus completing the loop between server and client and realizing the well-known page-navigation paradigm of web applications[4].

As for Renderer, the event loop receives a parameter, k, which identifies the particular instance. The macro PageLoad is defined below.

EventLoop(k) =
  if *eventAvailable*(*eventQueue*(k)) **then**
    let e = *headEvent*(*eventQueue*(k)) **in**
      dequeue e **from** *eventQueue*(k)
      if *isNewUrlFromUser*(e) **then**
        PageLoad(GET, *url*(e), $\langle\rangle$, k)
      elseif *isFormSubmit*(e) **then**

[3] These include elements such as `<INPUT>`, `<SELECT>`, `<OPTION>` etc.

[4] Notice that we are not considering here AJAX applications, where a Request is sent out directly from Javascript code, and the results are returned as raw data to the same script, instead of being used to replace the contents of the page. The general processing for this case is, however, similar to the one we describe here.

let $f = formElement(e), data = encodeFormData(f),$
$a = action(f), m = method(f), u = resolveUrl(f, a)$ **in**
match $(schema(u), m)$:
case $(\texttt{http}, \texttt{GET})$: MutateUrl$(u, data, k)$
case $(\texttt{http}, \texttt{POST})$: SubmitBody$(u, data, k)$
case $(\texttt{ftp}, \texttt{GET})$: GetAction$(u, data, k)$
case $(\texttt{ftp}, \texttt{POST})$: GetAction$(u, data, k)$
case $(\texttt{javascript}, \texttt{GET})$: GetAction$(u, data, k)$
case $(\texttt{javascript}, \texttt{POST})$: GetAction$(u, data, k)$
case $(\texttt{data}, \texttt{GET})$: GetAction$(u, data, k)$
case $(\texttt{data}, \texttt{POST})$: PostAction$(u, data, k)$
case $(\texttt{mailto}, \texttt{GET})$: MailHead$(a, data)$
case $(\texttt{mailto}, \texttt{POST})$: MailBody$(a, data)$
else
handle other events

We do not further specify here the mail-related variants MailHead and MailBody (although it is interesting to remark that they do not need further access to the browsing context, contrary to most other methods, since no reply is expected from them – and thus their applicability in web applications is close to nil). We also glide over the possibility of using a `https` schema, which however implies the same processing as `http`, with the only additional step of properly encrypting the communication. Given the purposes of this paper we omit a definition of GetAction and PostAction, since they involve URL schemas (namely: `ftp`, `javascript` and `data`) that have not been addressed in the transport layer model in [17]. Thus, below we only refine MutateUrl and SubmitBody together with PageLoad.

The macro MutateUrl consists in synthesizing a new URL from the action and the form data (which are encoded as query parameters in the URL) and in causing the browsing context to navigate to the new URL:

MutateUrl$(u, data, k) =$
let $u' = u \cdot \texttt{?} \cdot data$ **in** PageLoad$(GET, u', \langle\rangle, k)$

The macro SubmitBody differs only in the way the data is encoded in the request, namely not as part of the URL, as above, but as body of the request:

SubmitBody$(u, data, k) =$ PageLoad$(POST, u, data, k)$

The macro PageLoad starts an asynchronous Transfer—which is defined in [17]—and (re-)initializes the browsing context and the HTMLProcessor; the latter is also defined in [17] and will handle the Response:

PageLoad$(m, u, data, k) =$
Transfer$(m, u, data,$ HTMLProc$, k)$
$htmlParserMode(k) := Parsing$
let $d =$**new** Dom **in**
$DOM(k) := d$
$curNode(k) := root(d)$

Notice that while for the sake of brevity we have modeled navigation to the response provided by the server as a direct TRANSFER here, in reality it would require a few additional steps, including: storing the previous document and associated data in the session history, releasing resources used in the original page (e.g., freeing images or stopping plug-ins that were running), etc. While resource management can be conveniently abstracted, handling of history navigation (i.e., the Back, Forward and Reload commands available in most browsers) is a critical component in proving robustness, safety and correctness properties of web applications, and will be addressed in future work.

3 A High-Level WEBSERVER Model

We define here a companion model to the browser model: a high-level model WEBSERVER (Sect. 3.1) with typical refinements for the underlying handler modules, namely for file transfer (Sect. 3.2), CGI (Sect. 3.3) and scripting modules (Sect. 3.4).

To concentrate on the core issues we abstract in this section from the transmission protocol phase during which the connection between client and server is established and rely upon an abstract SEND mechanism; the missing elements to incorporate this phase can be defined as shown in detail for the browser component models in [17].

3.1 Functional Request-Reply Web Server View

In the high-level view the server appears as dispatcher which to handle a request finds and triggers the code (a 'module') the execution of which will provide a response to the request.[5] Thus a high-level web server model can be formulated as an ASM WEBSERVER which in a reactive manner, upon any *request* in its *requestQueue*, will delegate to a new agent (read: a thread we call *request handler*) to handle the EXECution of the request—if the *request* passes the *Security* check and the *requestedModule* is *Available* in and can be loaded by the server.

We succinctly describe checking various kinds of *Property* (here access security, module availability and loadability) by functions (here *checkSecurity*, *findModule loadModule*) whose values are

- either three-digit-values v in an interval $[n00, n99]$, for some $n \in [0, 9]$ as defined for each *Property* of interest in [5, Sect.4.1] to indicate that the *Property* holds or fails to hold (in the latter case of *PropertyFailure*(v) the value v also indicates the reason for the failure), or
- some different value, like a found requested module, which implicitly also indicates that the checked *Property* holds, e.g. that the requested module is available or could be successfully loaded.

[5] The ASM model for the Virtual Provider (VP) defined in [7] has a similar structure: it receives requests, forwards them to appropriate providers and collects the replies from the providers to return them to the original requestor.

Since in case *PropertyFailure*(v) is true the function value v is assumed to indicate the reason for the failure, the value appears in the *failureReport* the WebServer will Send to the client. The function *failureReport* abstracts from the details of formatting the response message out of the parameters.

The *requestedModule* depends on the server *env*ironment, the *resourceName* that appears as part of the *request* and the *header*(*request*). For a loaded *module* StartHandler creates a new thread and puts it into its *init*ial state from where the thread will start its program, namely to Execute the *module*. A loaded *module* is of one of finitely many kinds. For the fundamental CGI and scripting module types we will detail in Sect. 3.3,3.4 what it means to Execute such a module.

To reflect the functional client/server request/reply view StartHandler appears as atomic action of the WebServer which goes together with deleting the *request* from the *requestQueue*. At the transmission protocol level the latter action becomes closing the connection. The atomicity reflects the fact that once a request has been handled, the server is ready to handle the next request.[6]

```
WebServer =
let request = head(requestQueue)
if request ≠ undef then  // react if there is some request
  let env = env(server, request)
  let s = checkSecurity(request, env)
  if SecurityFailure(s)
  then Send(failureReport(request, s))
  else
    let requestedModule =
      findModule(env, resourceName(request), header(request))
    if ResourceAvailabilityFailure(requestedModule) then
      Send(failureReport(request, requestedModule))
    else
      let module = loadModule(requestedModule, env)
      if ModuleLoadabilityFailure(module)
      then Send(failureReport(request, module))
      else StartHandler(module, request, env)
  Close(request)
where
  SecurityFailure(s) iff s = 403
  ResourceAvailabilityFailure(m) iff m = 503
  ModuleLoadabilityFailure(module) iff module = 500
  StartHandler(module, request, env) =
    let a = new (Agent) // launch a request handler thread
    program(a) := Exec(module)(request, env)
    mode(a) := init
  Close(request) = Delete(request, requestQueue)
```

[6] The ASM model supports this view due to the reactive character of ASMs.

3.2 Refinement for File Transfer EXECution

To start with a simple case we illustrate how the machine EXEC(*module*) can be detailed to a machine EXECFILETRANSFER(*module*) which handles file transfer *module*s, the earliest form of server module. Such a *module* simply buffers the requested *file* in an output buffer if the *file* is present at the location determined by the path from the *root*(*env*) to the *resourceName*(*request*). We use a machine TRANSFERDATAFROMTO which abstracts from the details of the (not at all atomic, but durative) transfer action of the requested file data to the output. The function *requestOutput*(*request*) abstractly represents the appropriate socket through which the response data are sent from the server to the requesting browser.[7]

We leave it open what the scheduler does with the request handler when the latter is DEACTIVATEd once the file transfer *isFinished*, i.e. when it has been detected (here via TRANSFERDATAFROMTO) that no more data are to be expected for the transfer.

EXECFILETRANSFER(*module*)(*request*, *env*) =
let *file* = *makePath*(*root*(*env*), *resourceName*(*request*))
if *mode*(**self**) = *init* **then**
  if *UndefinedFile*(*file*) **then**
    SEND(*failureReport*(*request*, *ErrorCode*(*UndefinedFile*)))
    DEACTIVATE(**self**) // request handler termination
  else
    SEND(*successReport*(*request*, *OkResponseCode*))
    mode(**self**) := *transferData* // Start to transfer the *file*
  if *mode*(**self**) = *transferData* **then**
    TRANSFERDATAFROMTO(*file*, *requestOutput*(*request*))
  if *isFinished*(*file*) **then** DEACTIVATE(**self**)
  where
    ErrorCode(*UndefinedFile*) = 404
    OkResponseCode = 200
    DEACTIVATE(**self**) = (*mode*(**self**) := *final*)

3.3 Refinement for Common Gateway Module EXECution

A Common Gateway Interface (CGI) [23] *module* allows the request handler to pass requests from a client web browser to an (agent which executes an) external application and to return application output to the web browser. There are two main forms of CGI modules, the historically first one (called CGI) and an optimized one called FastCGI [13]. They differ in the way they introduce agents for external process execution: CGI creates one agent for each request, whereas FastCGI creates one agent and re-uses it for subsequent requests to the same application (though with different parameters).

[7] Again this can be made precise as shown in detail for the browser model in [17].

CGI Module. A CGI *module* sends an error message if the *executable* for the requested process is not defined at the indicated location. Otherwise the requested process execution (by an independent newly created agent a, not by the request handler)[8] is triggered for the appropriate *requestVariables* (also called environment variables containing the request data), like Auth(entication)-Type, Query-String, Path-Info, RemoteAddr (of the requesting browser) and Remote-Host (of the browser's machine), etc.(see [23, Sect.5]) and a positive response is sent to the requesting client. Once the new agent a has been CONNECTed the request handler

- accepts any further *requestInput* stream (read: data stream coming from the browser) as input for the execution of the process by a, namely via the *stdin* stream of the *module*, and
- transmits any output which (via a's processing the *executable*) becomes available on the *module*'s *stdout* stream to the *requestOutput* stream (from where it will be sent to the requesting browser)—as long as there are data on the *requestInput* resp. on the *stdout* stream.

Thus to CONNECT a to (the agent **self** executing) the CGI *module* a channel is established between the $inputStream(a)$ and the *module*'s *stdin* stream resp. between the $outputStream(a)$ and the *module*'s *stdout* stream[9].

It is usually assumed that the executable $program(a)$ agent a gets equipped with eventually disconnects a (from the request handler **self**) so that the predicate $Connected(a, \mathbf{self})$ becomes false. Then EXEC(*module*) terminates wherefor the request handler is DEACTIVATEd. Nevertheless the agent a even after having been disconnected may continue the execution of the associated *executable* and may not terminate at all, but such a further execution would be unrelated to the computation of the request handler and from the WEBSERVER's point of view yields a garbage process. Even more, no guarantee is given that $program(a)$ does disconnect a. In these cases the operating system has to close the connection and/or to kill the process by descheduling its executing agent (e.g. via a time-out). The CGI standard [23] leaves this issue open, but is has to be investigated if one wants to provide some behavioral guarantees for the execution of CGI modules.

EXEC(*module*)(*request*, *env*) =
let *executable* = *makePath*(*root*(*env*), *resourceName*(*request*), *env*)
if *mode*(**self**) = *init* **then**

[8] Therefore each request triggers a fresh instance of the associated external application program to be executed. This is a possible source for exceeding the workload capacity of the machine where the server runs.

[9] In ASM terms $inputStream(a)$ is a monitored and $outputStream(a)$ an output location for the *executable*, whereas for the *module stdin* is an output location (whereby the request handler **self** passes input to a for the processing of the *executable*) and *stdout* a monitored location (whereby the request handler **self** receives from a output produced through processing the *executable*.)

if *UndefinedProcess*(*executable*) **then**
SEND(*failureReport*(*request*, *ErrorCode*(*UndefinedProcess*)))
DEACTIVATE(**self**)
else
let a = **new** (*Agent*) // launch a new process instance
program(a) := *executable*(*processEnv*(*env*, *requestVariables*(*request*))))
CONNECT(a, **self**)
SEND(*request*, *OkResponseCode*)
mode(**self**) := *transferData*
if *mode*(**self**) = *transferData* **then**
if *DataAvailable*(*stdout*)
TRANSFERDATAFROMTO(*stdout*, *requestOutput*(*request*))
if *verb*(*request*) = *POST* **and** *DataAvailable*(*requestInput*(*request*))
then TRANSFERDATAFROMTO(*requestInput*(*request*), *stdin*)
if *isDisconnected*(a) **then** DEACTIVATE(**self**)
where
ErrorCode(*UndefinedProcess*) = 404
OkResponseCode = 200
isDisconnected(a) = **not** *Connected*(a, **self**)

Remark. The server *env*ironment is needed as argument to compute the path information in *makePath*. This is particularly important for the optimized FastCGI version we describe now.

FastCGI Module. Concerning the execution of external processes a FastCGI module has the same function as a CGI module. There are two behavioral differences:

- A FastCGI module creates a new agent for the execution of a process only upon the first invocation of the latter by the request handler. An agent a which has been created to process an *executable* is kept alive once this processing *isFinished* so that the agent can become active again for the next invocation of that *executable*—with the new values for the *requestVariables*. To CONNECT(a, **self**) now means to link its (local variables for) input resp. output locations, denoted below by *in*(a), *out*(a), to corresponding locations of the (request handler **self** executing the) *module* from where resp. to which the data transfer from *requestInput* resp. to *requestOutput* is operated. In particular *in*(a) is used to pass the parameters *requestVariables*(*request*) of the process to initialize the *executable*.
- It is assumed that the program *program*(a) agent a gets equipped with eventually sets a location *EndOfRequest* for the current *request* to false, namely by updating this location during the TRANSFERDATAFROMCGI action. This makes the request handler terminate.

Thus the CGI structure is refined to the FastCGI module structure as follows:

```
EXEC(module)(request, env) =
let executable = makePath(root(env), resourceName(request), env)
if mode(self) = init then
  if UndefinedProcess(executable) then
    SEND(failureReport(request, ErrorCode(UndefinedProcess)))
    DEACTIVATE(self)
  else
    if thereisno a ∈ Agent with
        program(a) = executable(processEnv(env))
      then
        let a = new (Agent)
        program(a) := executable(processEnv(env))
    mode(self) := connect
if mode(self) = connect then
  let a = ιx(x ∈ Agent and
    program(a) = executable(processEnv(env)))
  CONNECT(a, self)
  INITIALIZE(program(a))
  mode(self) := transferData
if mode(self) = transferData then
  let reqin = requestInput(request), reqout = requestOutput(request)
  if DataAvailable(out(a))
    TRANSFERDATAFROMCGI(out(a), reqout, EndOfRequest(request))
  if verb(request) = POST and DataAvailable(reqin) then
    TRANSFERDATATOCGI(reqin, in(a))
if EndOfRequest(request) then DEACTIVATE(self)
where
  ErrorCode(UndefinedProcess) = 404
  INITIALIZE(program(a)) =
    PASSPARAMS(requestVariables(request), in(a))
    EndOfRequest(request) := false
```

TRANSFERDATATOCGI implies an encapsulation of the to be transmitted content into messages which carry either data or control information; inversely TRANSFERDATAFROMCGI implies a decoding of this encapsulation.

3.4 Refinement for Scripting Module EXECution

Scripting modules like ASP, PHP, JSP all provide dynamic web page facilities by allowing the server to run (directly through its request handler) dynamically provided code. We define here a scheme which makes the common structure of such scripting modules explicit.

As for CGI modules first the file for the to be executed code is searched at the place indicated by the *resourceName* of the *request*, starting at the *root* of the

server *env*ironment. If the file is defined, the code is executed not by an independent agent as for CGI modules, but directly by the request handler which uses as program the ScriptInterpreter. For the state management accross different server invocations by a series of requests from the same client the uniquely determined *sessionID* (associated to the *request* under the given *env*ironment) and the corresponding session and application (if any) have to be computed. The computation of session and application comprises that a new session resp. application is created in case none is defined yet in the server *env*ironment for the *sessionID* resp. *applicationName* of the *request*.[10] Furthermore the syntax conversion of the *script* file from quotation to full script code (denoted here by a machine QuoteToScript which is refined below for ASP, PHP and JSP) has to be performed and the corresponding host objects have to be created to be passed as parameters to the ScriptInterpreter call.

The functions involved to ComputeSession and to ComputeApplication, which allow the server to track state information between different requests of a same client, depend on the *module*, namely *sessionID*, *makeSession* (and therefore *session*), *applicationName*, *makeApplication* (and therefore *application*). Similarly for the functions involved to ComputeInterpreterObjects. We express this using the **amb** notation as defined in [10].

```
Exec(module)(request, env) =
let script = makePath(root(env), resourceName(request))
amb module in  // NB: use of module sensitive functions
  if mode(self) = init then
    if script = ErrorCode(UndefinedScript) then
      Send(failureReport(request, ErrorCode(UndefinedScript)))
      Deactivate(self)
    else
      let id =  sessionID(request, env)
        ComputeSession(id, request, env)
      let applName =  applicationName(resourceName(request))
        ComputeApplication(applName, request, env)
      scriptCode(request) ←  QuoteToScript(script, env)[11]
      mode(self) := compInterprObjs
  if mode(self) = compInterprObjs then
    ComputeInterpreterObjects(request, id, applName)
    program(self) :=
      ScriptInterpreter(scriptCode(request), InterpreterObjects))
  where
    ErrorCode(UndefinedScript) = 404
    ComputeSession(id, request, env) =
```

[10] Typical refinements of the *sessionID* function also contain specific security policies we necessarily have to abstract from in this high-level description.

[11] The definition of ASMs with return value supporting the notation $l \leftarrow M(x)$ is taken from [12, Def.4.1.7.].

```
    if session(id) = undef then
      session(id) := makeSession(request, env, id)
  ComputeApplication(applName, request, env) =
    if application(applName) = undef then
      application(applName) := makeApplication(request, env, applName)
  ComputeInterpreterObjects(request, id, applName) =
    reqObj(request) := makeRequestHostObj(request)
    responseObj(request) := makeResponseHostObj(request)
    sessionObj(request) := makeSessionHostObj(session(id))
    applObj(request) := makeApplicationHostObj(application(applName))
    serverObj(request) := makeServerHostObj(request, env)
  InterpreterObjects =
    [reqObj(request), responseObj(request),
      sessionObj(request), applObj(request), serverObj(request)]
```

ASP/PHP/JSP Module. ASP, PHP and JSP modules are instances of the scripting module scheme described above. In fact their Exec(*module*) is defined as for the scripting scheme but each with a specific way to produce dynamic webpages, in particular with a specific computation of QuoteToScript, as we are going to describe below.

Also the following auxiliary functions and the called ScriptInterpreter are specific (as indicated by an index ASP, PHP, JSP) though not furthermore detailed here:

- The *make* ... *HostObj* functions are specialized to *make* ... $HostObj_{index}$ functions for each $index \in \{ASP, PHP, JSP\}$.
- ScriptInterpreter becomes $\text{ScriptInterpreter}_{index}$ for any *index* out of ASP, PHP, JSP.

See [14] for explanations how to construct an ASM model of the JavaScript interpreter as described in [4].

A PHP module acts as a filter: it takes input from a file or stream containing text or special PHP instructions and via their $\text{ScriptInterpreter}_{PHP}$ interpretation outputs another data stream for display.

ASP modules choose the appropriate interpreter for the computed *scriptCode* (so-called *active scripting*). Examples of the type of script code are JavaScript, Visual Basic and Perl.

Thus for ASP the definition of $\text{ScriptInterpreter}_{ASP}$ has the following form:

```
  ScriptInterpreter_ASP(scriptCode, InterprObjs) =
    let scriptType = type(scriptCode)
    ScriptInterpreter_scriptType(scriptCode, InterprObjs)
```

The value of *scriptCode*(*request*) is defined as the **result** computed by a machine QuoteToScript for a *script* argument. For the original version of PHP, to mention one early example, this machine simply computed a syntax transformation *transform*(*script*). Later versions introduced some optimization. At the

first invocation of QuoteToScript(*script*)—i.e. when the syntactical transformation of (the code text recorded at) *script* has not yet been *compiled*—or upon later invocations for a *script* (with code text) changed since the last compilation of *transform*(*script*), due to some code text replacement stored at *script* that is out of the control of the web werver, the target bytecode is *compiled* and *timeStamp*ed, using a *compile*r which can be specified using the techniques explained for Java2JVM compilation in [22]. At later invocations of the same *script* the already available *compiled*(*transform*(*script*)) bytecode is taken as *scriptCode* instead of recompiling again. Since the value of the code text located at *script* is not controlled by the web server, the function *timeStamp*(*script*) appears in this model as a monitored function.

scriptCode(*request*) ← QuoteToScript(*script*, *env*)
where
  QuoteToScript(*script*) =
    let *s* = *transform*(*script*)
    if *compiled*(*s*) = **undef or**
      timeStamp(*lastCompiled*(*script*)) ≤ *timeStamp*(*script*)
    then
      compiled(*s*) := *compile*(*s*)
      result:= *compile*(*s*)
      timeStamp(*lastCompiled*(*script*)) := *now*
      type(*compile*(*s*)) := *typeOf*(*script*, *env*)
    else result:= *compiled*(*s*)

For ASP and PHP the QuoteToScript machine describes an optional optimization[12] that cannot be observed from outside. For ASP the machine has the additional update for the *type* of the computed **result** (namely the *scriptCode*) that uses a syntax function *typeOf* which typically yields a directive, e.g.

$$< \%@Language = \text{“}JScript\text{”} \% >$$

or a default value.

The type of the *scriptCode* depends on the *script* and on the *env*ironment; for example the *env*ironment typically defines a default type for the case that nothing else is specified.

For JSP no syntax translation is required (formally the *transform* function is the identity function) because *scriptCode* is a class file (Servlet which comes with a certain number of fixed interfaces like `doPost()`, `doGet()`, etc.) so that the operations are performed by a JVM. This permits to embed predefined actions (implemented by Java code which can also be included from some predefined file via appropriate JSP directives) into static content. Here the machine QuoteToScript is mandatory because different invocations of the same *scriptCode* can communicate with each other via the values of static class variables.

[12] It is an ASM refinement of the non-optimized original PHP version.

JSF/ASP.NET Modules. It seems that a detailed high-level description of EXEC(*module*) for the *module*s as offered by the Java Server Faces (JSF [1]) and Active Server Pages (ASP.NET [19]) frameworks can be obtained as a refinement of the ASM defined above for the execution of scripting modules. As mentioned above PHP, ASP and JSP use a character based approach in which the script outputs characters (either explicitly through the Response object or implicitly by using the special notation converted by QUOTETOSCRIPT). The JSF and ASP.NET frameworks use their virtual-machine based environment (JVM resp. CLR) to provide more flexible ways for the SCRIPTINTERPRETER to write on the response stream (e.g. in ASP.NET based on the Windows environment) and to define a server-side event and state management model that relieves the programmer from having to explicitly deal with the state of a web page made up by several components. The programming model offered by these environments provides a sort of DOM tree where each node upon being visited is asked for the data to be sent as part of the response so that the programmer has the impression of manipulating objects rather than generating text of a Web page. For example, a request handled by the ASP.NET module triggers a complex lifecycle[13] which allows the programmer to manipulate a tree of components each of which has its own state, in part stored inside the web page (in the form of a hidden field) and in part put by the application into the session state. We are currently working on modeling these features as refinements of the ASM model for scripting module execution.

4 The Challenge of Accurate Analysis

Once sufficiently rich rigorous abstract web application models have been defined they can be used to accurately define properties of interest one would like to prove or falsify for the models via proofs or counterexamples which are preserved by correct refinements for existing implementations. This is by no means an easy task. For an illustrative example we can refer to [22] where in terms of rigorous models for Java, the JVM and a compiler Java2JVM the mere mathematically precise formulation of the compiler correctness property stated in Theorem 14.1.1. (p.177-178) needs 10 pages, the entire section 14.1.[14] A formulation in terms of some logic language understood by a theorem prover (e.g. in the language of KIV which has been used for various mechanical verifications of properties of ASMs [20,21] or in Event-B [6]) is still harder and will be considerably longer, as characteristic for formalizations.

We list here some properties of web applications we suggest to precisely formulate and prove or disprove in terms of abstract web application models.

A first group consists of correctness properties for the crucial session and state management:

- Session management refers to the ability of an application to maintain the status of the interaction with a particular browser. A typical property is that

[13] See http://msdn.microsoft.com/en-us/library/ms178472.aspx.

[14] In comparison the proof occupies 24 pages, the rest of chapter 14.

session state is not corrupted by user actions like hitting the *Back/Forward* buttons or navigating away from the page and then coming back.

- State management is about the virtual state of the application, which is usually distributed among multiple components on both client and server side, with parts of the state 'embedded' into the local state of several programs, and often also replicated entirely or partially. Typical desirable properties are that at significant time instants replicated parts of the state
  - are consistent, that is they are allowed to be out-of-sync at times and consistence is considered up to appropriate abstraction functions,
  - are equivalent between the client-side and the server-side of the state,
  - can be reconstructed, e.g. when the client can change and its state must be persisted to another client (for example from desktop to mobile).

A second group concerns robustness e.g. upon loss of a session or client and server state going out-of-sync, security and liveness.

A third group consists of what we consider to be the most challenging properties which are also of greatest interest to the users, namely application correctness properties. These properties are about the dependence of the intended application-focussed behavior of web applications on the programming and execution infrastructure—on the used browser, web server, net infrastructure (e.g. firewall, router, DNS), connection, plug-ins, etc. Such components are based on their own (not necessarily compatible) standards and therefore may influence the desired application behavior in unexpected ways. This makes their rigorous high-level description mandatory for a precise analysis. An outstanding class of such application-group-specific properties is about application integration where common services are offered on an application-independent basis (e.g. authentication or electronic payment services). We see such investigations as a first step towards defining objective content-based criteria for the reliability of web application software and for building reliable web applications, read: web applications whose properties of interest can be certifiably guaranteed—by theorem proving or model checking or testing or combinations of these activities—to hold under precisely formulated boundary conditions.

Acknowledgement. This paper is published in the two Proceedings volumes of the joint iFM2012 and ABZ2012 Conference held in Pisa (Springer LNCS 7321 and 7316).

References

1. Java Server Faces, http://www.jcp.org/en/jsr/detail?id=314
2. Python, http://www.python.org/
3. Tomcat, http://tomcat.apache.org/
4. ECMAScript language specification. Standard ECMA-262, Edition 5.1 (June 2011), http://www.ecma-international.org/publications/standards/Ecma-262.htm

5. HTTP1.1 part 2 message semantics, http://www.ietf.org (cosulted February 2012)
6. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
7. Altenhofen, M., Börger, E., Friesen, A., Lemcke, J.: A high-level specification for virtual providers. IJBPIM 1(4), 267–278 (2006)
8. Barros, A., Börger, E.: A Compositional Framework for Service Interaction Patterns and Interaction Flows. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 5–35. Springer, Heidelberg (2005)
9. Börger, E.: Approaches to modeling business processes. A critical analysis of BPMN, workflow patterns and YAWL. JSSM, 1–14 (2011), doi:10.1007/s10270-011-0214-z
10. Börger, E., Cisternino, A., Gervasi, V.: Ambient Abstract State Machines with applications. JCSS 78(3), 939–959 (2012)
11. Börger, E., Fruja, G., Gervasi, V., Stärk, R.: A high-level modular definition of the semantics of C#. Theoretical Computer Science 336(2-3), 235–284 (2005)
12. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
13. Brown, M.R.: Fast CGI specification (April 1996), http://www.fastcgi.com/
14. Dittamo, C., Gervasi, V., Börger, E., Cisternino, A.: A formal specification of the semantics of ECMAScript. In: VSTTE 2010, Edinburgh, Poster session (2010)
15. Fruja, N.G.: Towards proving type safety of .NET CIL. SCP 72(3), 176–219 (2008)
16. Fruja, N.G., Börger, E.: Modeling the.NET CLR Exception Handling Mechanism for a Mathematical Analysis. Journal of Object Technology 5(3), 5–34 (2006)
17. Gervasi, V.: An ASM model of concurrency in a web browser. In: Proceedings ABZ 2012. LNCS. Springer, Heidelberg (2012)
18. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions. Addison-Wesley Longman Publishing (2003)
19. Microsoft. ASP.NET, http://www.asp.net
20. Schellhorn, G., Ahrendt, W.: The WAM case study: Verifying compiler correctness for Prolog with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction – A Basis for Applications, vol. III, pp. 165–194 (1998)
21. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 16–31. Springer, Heidelberg (2006)
22. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer (2001)
23. W3C. CGI: Common Gateway Interface, http://www.w3.org/CGI/

Process Algebra for Event-Driven Runtime Verification: A Case Study of Wireless Network Management

Muffy Calder* and Michele Sevegnani

School of Computing Science, University of Glasgow, UK

Abstract. Runtime verification is analysis based on information extracted from a running system. Traditionally this involves reasoning about system states, for example using trace predicates. We have been investigating runtime verification for event-driven systems and in that context we propose a higher level of abstraction can be useful, namely reasoning at the level of user-perceived system events. And when considering events, then the natural formalism for verification is a form of *process algebra*.

We employ a universal process algebra that encapsulates both dynamic and spatial behaviour, based on Robin Milner's *bigraphs* [1]. Our models are an extension of his bigraphical reactive systems. These consist of a set of bigraphs that describe spatial and communication relationships, and a set of bigraphical reaction rules that define how bigraphs can evolve over time. We have extended the basic formalism to bigraphical reactive systems *with sharing* [2], to allow for spatial locations that can overlap.

In this talk we present a case study involving wireless home network management and the automatic generation of bigraphical models, and their analysis, in real-time. Wireless home networking is chosen as our case study because it is notoriously difficult to install and manage, especially for non-expert users. The Homework network management system [4] has been designed to provide user-oriented support in home wireless local area network (WLAN) environments. The Homework user interface includes drag and drop, comic-strip style interaction for users, and the information plane uses a stream database to record (raw and derived) events. Events include network behaviours such as detecting that a new machine has joined the network, resulting in new links and granting a DCHP lease, and user-intiated behaviours such as enforcing or dropping a policy. Policies forbid or allow access control; for example, a policy might block UDP and TCP traffic from a given site. All network and policy events (simple and derived) are recorded as a stream of tuples in the stream database. This part of the management system is illustrated in the left hand side of Figure 1.

On the right hand side of Figure 1 we depict our addtion to the Homework system: additional runtime verification components, and feedback

* This work is supported by the Engineering and Physical Sciences Research Council, under grant EP/F064225/1.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 21–23, 2012.

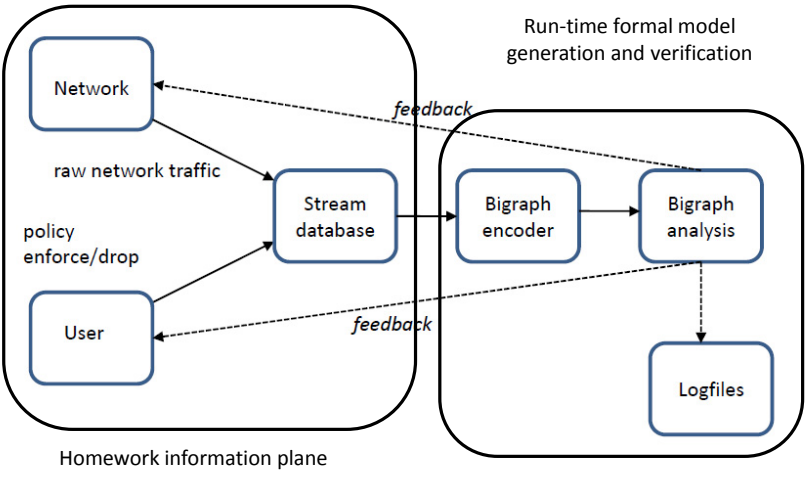


Fig. 1. Run-time model generation, analysis and feedback

from the verification to the network and users. In this talk we focus first on the bigraphical representations of networks topologies, the encodings of events that modify topologies as bigraph reaction rules, and the encodings of access control policy enforcements and revokations as bigraph reaction rules, and second on how the two components are deployed at run-time and their interplay. Both components are part of a larger bigraph evaluation and rewriting toolkit [3].

Briefly, the *Bigraph encoder* component encodes events (network topology or policy) as bigraphical reaction rules, in real-time, as they are stored in the stream database. The *Bigraph analysis* component has two roles. First, it generates the bigraphical representation of the current configuration of the WLAN, according to the sequences of reaction rules received from the *Bigraph encoder*. Namely, a sequence of bigraphs is generated. A simple example bigraph of a WLAN with one router (R), one machine (M1), and their respective wireless signals (S), is given in Figure 2.

Second, it analyses the current configuration by checking predicates encoded as instances of bigraph matching. These predicates encapsulate properties required for correct encoding of topology or policy events, as well as system properties, including detecting configurations that violate user-invoked access control policies. Example predicates include: "Machine 01:23:45:67:89:ab is in the range of the router's signal", "Host Laptop has access to the Internet", and "TCP traffic is blocked for machine with IP address 192.168.0.3". The results are logged and fed back to the system, or to the user, when a verification fails. An explanation of the failure, or a counter-example can be displayed to a user, using the graphical bigraph notation. An indication of failure is also sent to the network, if appropriate, e.g. to deny activation of a policy, and/or simply stored in a logfile.

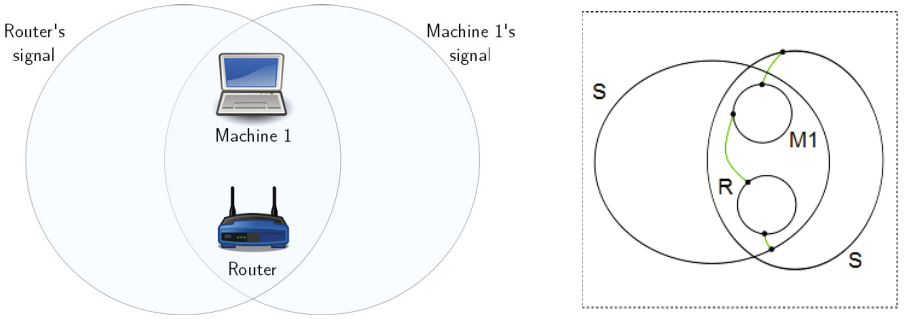


Fig. 2. Simple WLAN on the left and bigraph model on the right

The encoding and analysis components have been implemented on the router itself, and we give some empirical evidence of runtime verification from experiments using actual and synthetic network data.

References

1. Milner, R.: The space and motion of communicating agents. Cambridge University Press (2009)
2. Sevegnani, M., Calder, M.: Stochastic bigraphs with sharing. Glasgow University Computing Science Technical Report TR-2010-310 (2010)
3. Bigrapher, http://www.dcs.gla.ac.uk/~michele/bigrapher.html
4. Sventek, J., Koliousis, A., Sharma, O., Dulay, N., Pediaditakis, D., Sloman, M., Rodden, T., Lodge, T., Bedwell, B., Glover, K.: An Information Plane Architecture Supporting Home Network Management. In: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (2011)

Translating TLA$^+$ to B for Validation with ProB

Dominik Hansen and Michael Leuschel

Institut für Informatik, Universität Düsseldorf*
Universitätsstr. 1, D-40225 Düsseldorf
dominik.hansen@uni-duesseldorf.de, leuschel@cs.uni-duesseldorf.de

Abstract. TLA$^+$ and B share the common base of predicate logic, arithmetic and set theory. However, there are still considerable differences, such as very different approaches to typing and modularization. There is also considerable difference in the available tool support. In this paper, we present a translation of the non-temporal part of TLA$^+$ to B, which makes it possible to feed TLA$^+$ specifications into existing tools for B. Part of this translation must include a type inference algorithm, in order to produce typed B specifications. There are many other tricky aspects, such as translating modules as well as LET/IN and IF/THEN/ELSE expressions. We also present an integration of our translation into ProB. ProB thus provides a complementary tool to the explicit state model checker TLC, with convenient animation and constraint solving for TLA$^+$. We also present a series of case studies, highlighting the complementarity to TLC. In particular, we highlight the sometimes dramatic difference in performance when it comes to solving complicated constraints in TLA$^+$.

Keywords: TLA, B-Method, Tool Support, Model Checking, Animation.

1 Introduction and Motivation

TLA$^+$ [5] and B [1] are both state-based formal methods rooted in predicate logic, combined with arithmetic and set theory. There are, however, considerable differences:

- TLA$^+$ is untyped, while B is strongly typed.
- The concepts of modularization are very different (as we will see later in the paper).
- TLA$^+$ and B both support sets and functions. However, functions in TLA$^+$ are total, while B supports relations, partial functions, injections, bijections, etc.
- TLA$^+$ has several constructs which are lacking in B, such as an IF/THEN/ELSE for expressions and predicates[1], a LET/IN construct or the CHOOSE operator. The latter enables one to define recursive functions over sets, which are akwkard to define in B.

* Part of this research has been sponsored by the EU funded FP7 projects 214158 (DEPLOY) and 287563 (ADVANCE).

[1] B only provides an IF/THEN/ELSE for substitutions.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 24–38, 2012.

– TLA$^+$ allows to specify liveness properties while B is limited to invariance properties (temporal formulas such as liveness conditions will be excluded from our translation).

As far as tool support is concerned, TLA$^+$ is supported by the explicit state model checker TLC [13], and more recently by the TLAPS prover [2]. B has extensive proof support, e.g., in the form of the commercial product AtelierB [3]. The animator and model checker ProB [6] can also be applied to B specifications. Both AtelierB and ProB are being used by companies, mainly in the railway sector for safety critical control software. Some of the goals of our translation are

– to gain a better understanding of the common core and of the differences between TLA$^+$ and B,
– to obtain an animator for TLA$^+$,
– and to obtain a constraint solver for TLA$^+$.

Indeed, TLC is a very efficient model checker for TLA$^+$ with an efficient disk-based algorithm and support for fairness. ProB has an LTL model checker, but does not support fairness (yet) and is entirely RAM-based. The model checking core of ProB is less tuned than TLA$^+$. However, ProB offers several features which are absent from TLC, notably an interactive animator with various visualization options. More importantly, the ProB kernel provides for constraint solving over predicate logic, set theory and arithmetic. ProB can also deal quite well with large data values. This has many applications, from constraint-based invariant or deadlock checking [4], over to test-case generation and on to improved animation because the user has to provide much less concrete values than with other tools. It also makes certain specifications "executable" which are beyond the reach of other tools such as TLC.

We suppose that the reader is familiar with either TLA$^+$ or B. Indeed, we hope that through our translation, TLA$^+$ constructs can be understood by B users and vice-versa. Below, in Sect. 2 we introduce the essentials of our translation on a simple example, while in Sect. 3 we present the translation more formally. In Sect. 4 we present case studies and experiments, and will also compare the tools ProB and TLC. We conclude with more related and future work in Sect. 5.

2 An Example Translation from TLA$^+$ to B

To allow B users to become familiar with TLA$^+$, we present a variation of the well known HourClock example from Chapter 2 of [5]. Figure 1 shows the *MyHourClock* module, which avoids the IF/THEN/ELSE expression of the original at this point. The specification describes the typical behavior of a digital clock displaying only hours. The module starts with the MODULE clause followed by the name of the specification. The analogous clause of a B machine is MACHINE or MODEL. At the beginning of the module body, arithmetic operators such as

MODULE *MyHourClock*

EXTENDS *Integers*
CONSTANTS c
ASSUME $c \in 1 \,..\, 12$
VARIABLES hr
$Init \triangleq hr = c$
$add\_1(p) \triangleq p + 1$
$Inc \triangleq hr < 12 \wedge hr' = add\_1(hr)$
$Reset \triangleq hr = 12 \wedge hr' = 1$
$Next \triangleq Inc \vee Reset$

Fig. 1. Module MyHourClock

+ or ".." are loaded via EXTENDS from the standard module *Integers*. These operators are not built-in operators in TLA$^+$ and can either be defined by the user or imported with their usual meaning as here. The declaration of constants and variables is identical in both languages. The ASSUME clause in TLA$^+$ correspondes to the PROPERTIES clause in B.

To understand the meaning of the other definitions in the module we need some additional information.[2] For our translation we use a configuration file, as TLC also uses, telling us the initial state and the next-state relation of the module. For this example we suppose *Init* to be the initial state predicate and *Next* to be the next-state relation. *Init* indicates that the variable *hr* has the value of the constant c in the initial state. *Next* is separated into two actions by the disjunction operator. An action is a before-after predicate describing a transition to a next-state with the aid of the prime operator($'$). A primed variable represents the variable in the next-state. The use of the additional *add_1* operator may seem artificial here; its purpose is to demonstrate another aspect of our translation.

Figure 2 shows the translated B Machine of the *MyHourClock* example. We use the BECOMES/SUCH/THAT substitution under the INITIALISATION clause to initialize the variables of the B machine. It assigns a value to the variable such that the predicate in the brackets is satisfied. The TLA$^+$ actions *Inc* and *Reset* are translated as separate B operations. We represent the prime operator in B by adding a local auxiliary variable for every variable. The auxiliary variables (with suffix "_n") are generated in the ANY part of the ANY/WHERE/THEN substitution and get their value in the WHERE part. If the predicate in the WHERE part is not satisfiable the operation can not be executed. Finally, the values of the auxiliary variables are assigned to the corresponding global variables in the THEN part.

Operators such as *add_1* are translated using B definitions. B definitions are a kind of macro and help to write frequently used expressions. They are syntactic sugar and will be resolved in the the parsing phase. Furthermore, they can have parameters. Using B definitions avoids to replace an operator call by the

[2] "What those definitions represent [...] lies outside the scope of mathematics and therefore outside the scope of TLA$^+$" (see p. 21 of [5]).

```
MACHINE MyHourClock
DEFINITIONS  add_1(p) == p + 1
CONSTANTS c
PROPERTIES c ∈ 1..12
VARIABLES hr
INVARIANT hr ∈ ℤ
INITIALISATION hr :(hr = c)
OPERATIONS
  Inc_Op = ANY hr_n
           WHERE hr < 12 ∧ hr_n = add_1(hr)
           THEN hr := hr_n
           END
  Reset_Op = ANY hr_n
           WHERE hr = 12 ∧ hr_n = 1
           THEN hr := hr_n
           END
END
```

Fig. 2. Machine MyHourClock

definition of the operator. The arithmetic operators are translated with the use of B's built-in operators. Therefore, they do not appear in the DEFINITIONS clause. Finally, to obtain a correct B machine our translation has inferred and added the types of the variables in the INVARIANT clause.

3 The Translation from TLA$^+$ to B

3.1 Type System

The basis of our translation is a mapping of TLA$^+$ values to B values. Due to the strict type system of B, every B value has to be associated with a type. Below we list the translations of the TLA$^+$ values and the resulting restrictions:

- Numbers: In B real numbers are not supported. Thus, only integers can be translated. They get the B type $\mathbb{Z}$.
- The boolean values TRUE and FALSE are identical in both languages. They get the B Type BOOL.
- The concepts of strings are different in both languages. In TLA$^+$ a string is a sequence of characters and a single character can be accessed. However, a string in B is atomic and has the base type STRING. For the translation we currently reject strings if they are used as tuples.
- A model value is none of TLA$^+$'s own values but one of TLC's. But it is established to use TLA$^+$ together with TLC so we deliver a suitable translation. TLC allows to assign a model value or a set of model values to a constant in the configuration file. The equivalent of a model value is an element of an enumerated set in B. To make different model values comparable to each other we put them in the same enumerated set named: ENUM$_i$. However if two model values are never compared in the specification, we put

them in different sets (such as ENUM_1 and ENUM_2). The B type of a model value is the name of the enumerated set containing it.
- There is one main difference between sets in TLA$^+$ and B. In B all elements of a set must have the same type, i.e., a set has the B type $\mathbb{P}\tau$ where τ is the type of all its elements.
- In both languages functions are a mapping from a domain to a range. The B type of a function is $\mathbb{P}(\tau_1 \times \tau_2)$, where $\mathbb{P}\tau_1$ is the type of the domain and $\mathbb{P}\tau_2$ is the type of the range.
- In TLA$^+$ a record is a special case of a function whose domain is a set of strings (the field names). In B records have their own type $struct(h_1 : \tau_1, \ldots, h_n : \tau_n)$, where $h_1, \ldots, h_n$ are the names of the fields and $\tau_1, \ldots, \tau_n$ the corresponding field types.
- Likewise, tuples are based on functions in TLA$^+$. The domain is the interval from 1 to n, where n is the number of components. We translate tuples as sequences with the type $\mathbb{P}(\mathbb{Z} \times \tau)$. Thereby all components of a tuple must be of the same type τ.

In B only values of the same type are comparable to each other and variables as well as constants can only have one type. To verify these rules a type checking algorithm is required. Moreover we need a type inference algorithm to add missing type declarations to the translated B machine as shown in the example in Sect 2. Type checking and type inference are closely related and can be handled simultaneously.

We use an inference algorithm similar to [9], adapted to the B type system, where we add an extra type u representing an unspecified type. At the beginning each variable and constant have this type. The algorithm is based on the recursive method $eval(e, \epsilon)$, dealing with a TLA$^+$ expression e and an expected type ϵ. Evaluating an expression *eval* is applied recursively to its subexpressions. Moreover *eval* tries to unify the expected type with the type of the expression and returns the resulting type. The expected type of a subexpression is deduced from type informations of the operator calling this subexpression. Type informations of an operator arise from the translation. As an example the TLA$^+$ operator $+$ is translated by the B built-in operator $+$ and its operands are assumed to be integers. There are polymorphic operators such as $=$, which only require that both operands have the same type. In this case the expected type for both operands is u but the resulting types of both sides have to be unified. Due to unification, the *eval* method is only once applied to each (used) expression of the TLA$^+$ module. Moreover, declarations in the configuration file are also taken into account to infer the types of constants. The algorithm fails either if a unification of two types fails or if a variable or constant still has a variable type u (or a type constructor containing u such as $\mathbb{P}u$) at the end of algorithm.

3.2 Translation Rules

In this section we present translation rules for concepts which are different in TLA$^+$ and B, or even missing in B.

In contrast to TLA$^+$, B distinguishes between boolean values and predicates. The difference is already present at the syntactical level. Logical operators such as $\wedge$ or $\vee$ cannot be applied to boolean values. Similarly, variables or constants can not take a predicate as a value. Though, there is a way to convert from a predicate to boolean and vice versa. A predicate can be converted to a boolean value using the *bool* operator. The other way around, we can turn a boolean value into a predicate by comparing it with TRUE. The translation of the TLA$^+$ predicate

$$\text{TRUE} = (\text{TRUE} \vee \text{FALSE})$$

demonstrates both conversions:

$$\text{TRUE} = \mathit{bool}((\text{TRUE} = \text{TRUE}) \vee (\text{FALSE} = \text{TRUE}))$$

The IF/THEN/ELSE construct can be used in variety of ways in TLA$^+$. The two branches can consist of arbitrary expressions with or without primed variables. There is no general way to translate this construct with the IF/THEN/ELSE substitution of B. In order to make a translation to B possible, we first have to restrict both branches to the same type. In case the branches are predicates the construct

$$\text{IF } P \text{ THEN } e_1 \text{ ELSE } e_2$$

can be translated using two implications as

$$(P \Rightarrow e_1) \wedge (\neg(P) \Rightarrow e_2)$$

If e_1 and e_2 are expressions, we cannot use this scheme. Our solution is to create for both branches a lambda function, with respectively e_1 and e_2 as result expression. Moreover we choose TRUE as the sole dummy element of the domains. The "trick" is to add the condition P respectively its negation $\neg(P)$ to the corresponding function. As a consequence, one of the functions is always empty. As already mentioned, B functions are sets and we can apply $\cup$ to combine them (the result is still a function here). To get the value of the IF/THEN/ELSE construct, we just have to call the function with the value TRUE as argument:

$$(\lambda t.(t \in \{\text{TRUE}\} \wedge P | e_1) \cup \lambda t.(t \in \{\text{TRUE}\} \wedge \neg P | e_2))\,(\text{TRUE})$$

Compared to other possible translations, ours has the advantage that the expressions e_1 and e_2 are guarded by P and $\neg P$, i.e., the translation of IF $x = 0$ THEN 1 ELSE $1/x$ is well-defined in B. The translation of the CASE construct is based on the same principle. However, every case is treated as single branch and only one case can be true at the same time.

The $\mathit{LET}\ d \triangleq f\ \mathit{IN}\ e$ construct allows to define a "local" operator d which can only be used in the expression e. This operator is treated as an ordinary operator and translated with the aid of a B definition; conserving the scope of the operator. In TLA$^+$ operators within different LET/IN constructs could have the same name. We avoid name clashes by adding suffixes to multiply used names.

In TLA$^+$ the CHOOSE operator is used to choose an arbitrary value of a set. The operator works in a deterministic way and chooses always the same value for a given set. It is often combined with a recursive function such as determining the sum of a set. In B there is no way to express the general functionality of the CHOOSE operator[3] and recursive functions are still not (well) supported by B tools. Hence, we developed a way to handle frequently used operators which are based on the CHOOSE operator or on recursive functions. The principle is inspired by the way TLC overrides operators by its Java implementation: we create a new TLA$^+$ standard module (see Figure 3.2) with some useful operators, and during the translation these operators will be overridden by B built-in operators.

MODULE *TLA2B*

```
EXTENDS Integers, Sequences
MinOfSet(S) ≜ CHOOSE p ∈ S : ∀ n ∈ S : p ≤ n
MaxOfSet(S) ≜ CHOOSE p ∈ S : ∀ n ∈ S : p ≥ n
SetProduct(p) ≜
        LET prod[S ∈ SUBSET Int] ≜
                IF S = {} THEN 1
                 ELSE LET q ≜ CHOOSE pr ∈ S : TRUE
                          IN   q * prod[S \ {q}]
        IN   prod[p]
SetSummation(p) ≜
        LET sum[S ∈ SUBSET Int] ≜
                IF S = {} THEN 0
                 ELSE LET q ≜ CHOOSE pr ∈ S : TRUE
                          IN   q + sum[S \ {q}]
        IN   sum[p]
PermutedSequences(S) ≜
  LET perms[ss ∈ SUBSET S] ≜
        IF ss = {} THEN {⟨⟩}
         ELSE LET ps ≜ [x ∈ ss ↦
                          {Append(sq, x) : sq ∈ perms[ss \ {x}]}]
              IN   UNION {ps[x] : x ∈ ss}
  IN   perms[S]
```

The concepts of modularization are different in TLA$^+$ and B. In B a machine is a closed system. Indeed, a machine can be included by another machine but variables can only be modified by its operations. As a result, a single machine of a compound system can be verified individually. A TLA$^+$ module does not need to satisfy this property. Hence, we translate a compound of TLA$^+$ modules as a single B machine:

– A module extending another module will be treated as a single module containing declarations and definitions (including local definitions of the

[3] Even though the operator does appear inside mathematical constructions of [1].

extended module) of both modules. Otherwise, there are no further differences in comparison to a translation of a single module.
- The statement

$$I \triangleq \mathit{INSTANCE}\ M\ \mathit{WITH}\ v_M \leftarrow v,\ c_M \leftarrow c$$

allows the specifier to use the definitions of the module M. Thereby, all variables and constants of M have to be overridden by variables and constants (or constant expressions) of the module instantiating M. A definition d_M of Module M can be accessed via $I!d_M$. Also multiple instantiations of the same module are possible. We translate every definition d_M of M as an ordinary definition by only renaming it to I_d_M and by overriding variables and constants as described in the statement.

In Sect. 2 we translated a TLA$^+$ action from Fig. 1 to a B operation in Fig. 2, but we did not exactly define what TLA$^+$ actions are and how they are extracted. An action is defined to be "an ordinary mathematical formula, except that it contains primed as well as unprimed variables"(see p. 16 of [5]). Following this definition we could handle the whole next state relation as a single action. However, this is not advisable, amongst others because of poor user feedback for animation, proof and model checking. Consequently we separate actions with the aid of the disjunction operator. If a disjunction of two actions occurs in a subdefinition of the next state relation, we also will treat them as separate actions unless the subdefinition has no parameter. Parameters indicate that a subdefinition can be used in different variations and multiple times; the translation should not dissolve this structure of a module. In this case the subdefinition is translated with a B definition. The mechanism splitting the next-state relation into separate actions is similar to what TLC does when it pre-processes TLA$^+$ specifications, except that TLC resolves definitions regardless if they have parameters.

A special translation is possible if an action contains an existential quantifier:

$$act \triangleq \exists x \in S : P(x)$$

The bounded variables of the quantification are handled as parameters of the resulting B operation:

$$act\_op(x) = \text{ANY} \ldots \text{WHERE}\ x \in S \wedge P(x)\ \text{THEN} \ldots \text{END}$$

The advantage is that a user can choose a possible value for the parameter x during the animation process (values for x satisfying $P(x)$ are generated by ProB).

4 Implementation and Experiments

The translator is implemented in Java and is called TLA2B. The frontend of TLA2B is based on SANY (cf., Chapter 12 of [5]) for parsing the module and

performing a semantic analysis. Likewise, SANY serves as the frontend of the modelchecker TLC. Moreover, we reuse the configuration file parser of TLC. But the semantic analysis of the configuration file ist different: TLC requires a value for every constant of the corresponding module. In our case, a constant only has to be given a value if the type of the constant cannot be inferred from the module. Otherwise, values of constants can be chosen at a later point in time (ProB infers values for a constant satisfying possible restrictions of the ASSUME clause). TLA2B can handle the clauses SPECIFICATION (temporal description of the specification), INVARIANT (an invariant holding in every state) and overriding of constants and definitions beside the already mentioned INIT (initial state) and NEXT (next state relation). Before inferring and checking types, we conduct a further analysis phase discarding the unused definitions of a module. As an example, temporal definitions are excluded from the translation. The remaining part of the TLA2B consists of implementations of the algorithms described in Sections 2 and 3. Finally, TLA2B creates a B machine file (.mch) containing the translated B machine.

TLA2B has been integrated into ProB as of version 1.3.5: opening a TLA$^+$ module ProB invokes TLA2B to translate the module. As can bee seen in Fig. 3, the TLA$^+$ module is displayed in the editor while ProB runs the translated B machine in the background. The editor offers syntax highlighting and gives an easy way to modify the module.

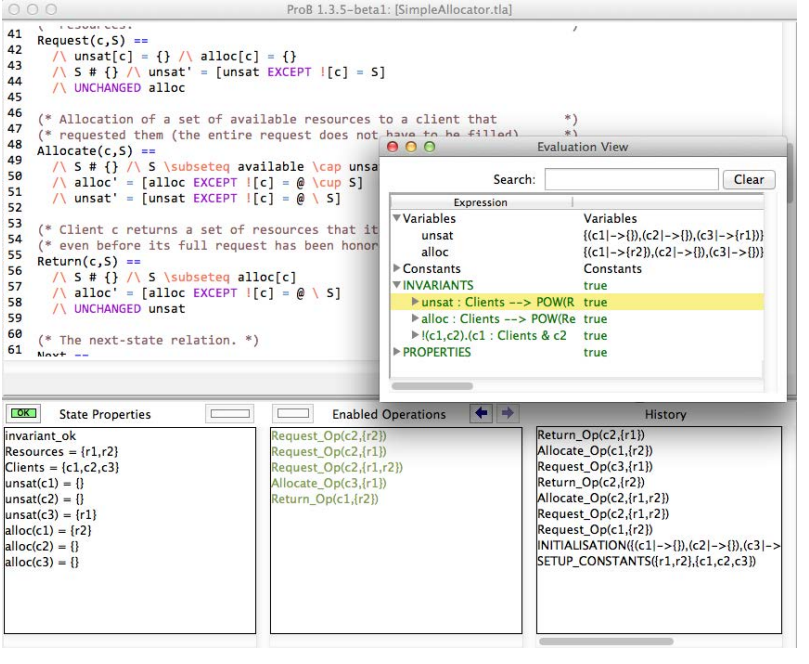


Fig. 3. ProB animator for the SimpleAllocator specification

The following examples show some fields of application of TLA2B in combination with ProB.[4] It is not our intention to present a complete comparison between ProB and TLC. The experiments were all run on a system with Intel Core2 Duo 2 GHz processor, running Windows Vista 32 Bit, TLC2 2.03 and ProB 1.3.5-beta1.

Note that both ProB and TLC support symmetry, but in different ways. In TLC symmetries are provided by the user (e.g., in a configuration file) and are not checked, ProB identifies symmetries over given sets automatically.

SimpleAllocator. As the first example we use the resource allocator case study from [8]. The purpose of the system is to manage a set of resources that are shared among a number of client processes. The first abstract specification of the system is the SimpleAllocator. TLA2B translates the module without the need for any modification (the TLA$^+$ module and the translated B machine are shown in Appendix A). Clients and resources are specified as sets of model values and allow TLC as well as ProB to use symmetry. Table 1 summarises the running times of model checking for TLC and ProB. Without symmetry TLC is superior to ProB, but for larger set sizes ProB's symmetry outperforms TLC. It seems that TLC's symmetry reduction cannot deal well with larger base set sizes and a lot of symmetrical states exist (incidentally, a situation where symmetry reduction could be particularly useful). This is actually to be expected, given the description of the symmetry reduction algorithm in [5]: when a state is added TLC checks for every permutation of it whether it already exists. This is expensive when there are many such permutations.

Table 1. SimpleAllocator: Runtimes of Model Checking (times in seconds)

Clients	Resources	TLC (no symmetry)	TLC (symmetry)	ProB (no symmetry)	ProB (symmetry)
3	2	<1	<1	<2	<1
4	3	28	2	678	8
5	3	450	29	-	28
6	3	>4200	573	-	90

Login. To specifically test this aspect of symmetry reduction, we have written the TLA specification Login which simply allows users to login and logout and deadlocks if all users have logged in. Here, for 9 Users, TLC without symmetry reduction takes 1 second to find the deadlock, but did not terminate within 105 minutes with symmetry enabled. ProB takes 0.73 seconds without symmetry, and 0.04 using hash symmetry reduction [7]. For 21 users, TLC requires 141 seconds to find the deadlock without symmetry, and with symmetry an error message is generated.[5] ProB with hash symmetry takes 0.29 seconds to find

[4] The source code of the examples are available in the technical report at: http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschelTLA2012 .

[5] "Attempted to construct a set with too many elements (>1000000)". This error message already appears with 15 Users.

the deadlock for 21 users. The constraint-based deadlock checking algorithm [4] finds a deadlock in less than 0.01 seconds for 21 users.

SchedulingAllocator. This is an advanced version of the SimpleAllocator from [8]. However, this time a small modification is required to be able to validate the specification using our tool. Indeed, the SchedulingAllocator contains the definition PermSeqs(S) that is based on a recursive function and computes the set of permutation sequences of the set S. To translate this to the B built-in operator, we have to override PermSeqs with the PermutedSequences definition provided by our TLA2B module. For this, we simply have to create a new module MCSchedulingAllocator extending the SchedulingAllocator as well as the TLA2B module and then add the override statement `PermSeqs <- PermutedSequences` to the configuration file. The results of model checking are comparable to the SimpleAllocator specification, and can be found in Table 2.

Table 2. McSchedulingAllocator: Running times of Model Checking (times in seconds)

Clients	Resources	TLC (symmetry)	ProB (symmetry)
3	2	1	2
4	3	70	165
5	3	>3600	1579

Producer-Consumer. Another example is the specification of a multi-threaded program by Charpentier taken from `http://www.cs.unh.edu/%7Echarpov/Teaching/TLA/`. The specification describes a system of threads working on a buffer. In case of a critical ratio between consumer and producer threads, the system can deadlock. After translation ProB reproduces the various deadlocks by model checking. For example, for 11 producers and 10 consumers a deadlock can be reached after 431 steps. Using the AtelierB provers we have also managed to prove the invariant of that model, i.e., that the buffer capacity is never exceeded and that the waitSet only contains valid participants. This required 8 interactive proofs and 5 automatic ones.

Constraint Solving: GraphIso and N-Queens. One of the distinguishing features of ProB is its ability to solve complicated high-level constraints. For example, to find an isomorphism between two graphs (of out-degree exactly one and with nine vertices) , ProB requires less than a second to find all solutions, while TLC requires over two hours to find the first solution.

As another example, consider the well-known N-Queens puzzle. We have experimented with two encodings of the puzzle: one[6] where we use the model checker to search for all valid placements of N queens on an N×N chessboard and a more declarative encoding where we directly write a predicate describing

[6] The specification was written by S. Merz and is included in the TLA$^+$ Tools Distribution.

all valid solutions (i.e., all solutions are generated in single set-builder rather than through an iterative algorithm). As can be seen in Table 3, the model checking approach can only deal with very small values of N. In contrast, ProB can handle values of N up to 13 for the declarative version of N-Queens. Furthermore, when one is interested in only one solution, ProB can, e.g., find a solution for N=50 in less than a second. (Restricting to single solutions does not make much of a performance difference for TLC, however.)

Table 3. Finding *all* solutions for N-Queens (times in seconds)

N	Solutions	N-Queens (imperative)		N-Queens (declarative)	
		TLC	ProB	TLC	ProB
4	2	1	<2	<1	<1
5	10	>3600	>3600	<1	<1
6	4	-	-	1	<1
7	40	-	-	16	<1
8	92	-	-	375	<1
9	352	-	-	2970	<1
10	724	-	-	-	<1
11	2,680	-	-	-	<1
12	14,200	-	-	-	9
13	73,712	-	-	-	41

We have also successfully animated several other existing models from the literature, but several specifications are rejected by TLA2B due to type conflicts or unsupported concepts such as real numbers. In summary, the ProB constraint solving capabilities open up the way to animate and validate new kinds of specifications, which are outside the reach of TLC. TLC on the other hand is extremely valuable when it comes to explicit state model checking for large state spaces. However, ProB's symmetry reduction techniques seem to scale better than TLC's.

5 More Related Work, Discussion and Conclusion

The paper by Mokhtari and Merz [10] presents an animator and model checker for an executable subset of TLA$^+$. The article clearly outlines the needs for an animator for TLA$^+$; unfortunately the tool seems to be no longer to be available.

Mosbahi et al. [11] describe an approach of a translation from B to TLA$^+$. In contrast to our translation they have to deal with concepts which are missing in TLA$^+$ such as partial functions. Moreover their main intention is to let TLC verify liveness properties on the translated TLA$^+$ specification, to overcome the restriction of the B-Method to invariance properties. Otherwise, [12] presents a LTL model checker, implemented inside ProB, that can verify liveness properties. So far, this model checker does not support fairness conditions, but an extension would give us the possibility to enlarge TLA2B to support the temporal part of TLA$^+$.

In terms of features, we also plan to provide for TLA$^+$ the graphical visualization features of ProB already available for B, Z and Event-B. More work on translating various constructs effectively to B, such as the CHOOSE operator or recursive functions, is planned. Another important avenue of further work lies in improving our translation to B. In particular, we aim to generate various B style substitutions such as assignments or IF/THEN/ELSE constructs, rather than generic ANY substitutions. This makes the B translation more readable, but would also lead to noticeable performance improvements with ProB. E.g., in our experiments, this would lead, to a further 20 % runtime improvement for the SimpleAllocator example and up to a factor 2 for other examples.

We would also like to better exploit the symmetry reduction provided by ProB. While the SimpleAllocator, SchedulingAllocator and Login example worked well, the symmetry in the Producer-Consumer example could only be exploited by manually tweaking the B translation. We would like to automate this as much as possible, as it can lead to a considerable performance boost (after tweaking ProB with symmetry requires about a minute to find the 413 step counter example for the Producer-Consumer example; TLC requires more than three and a half hours to to find a deadlock[7]). We are also interested in the correctness of our translation. A formal correctness proof is probably not feasible, but we hope to be able to extensively validate our translation, e.g., by exporting the state space computed by ProB to TLC and use TLC to check that it conforms to the original specification.

In conclusion, we have presented a translation from TLA$^+$ to B, which makes use of a type inference algorithm and effectively translates a large subset of TLA$^+$ to B. The complicated aspects of the translation are linked to the different modularization concepts, as well as to various operators which are missing in B. The translation also identifies operations and parameters within the TLA$^+$ specification formula, in order to make the translation more readable as well as to enable effective application of B tools. In particular, by integrating our translation into the ProB validation tool, we obtain a new tool for TLA$^+$ specifications which is complementary to TLC, providing convenient animation, expression evaluation, constraint solving and improved symmetry reduction. As our experiments show, TLC remains more effective for brute-force explicit state model checking, at least for those specifications which do not require solving complicated constraints. As such it is very useful that both these tools can be applied to TLA$^+$ specifications. The translation itself is also human readable, and we hope that the paper also provides a bridge between the TLA$^+$ and B communities.

Acknowledgements. We are grateful to Daniel Plagge for various discussions and helpful comments that helped in developing the translator. We also would like to thank Leslie Lamport and Stephan Merz for very useful feedback concerning TLA$^+$ and TLC and for giving us access to various specifications. Finally, we are thankful to anonymous referees for their useful feedback.

[7] When trying to use symmetry, the same error message occurs as in the Login example.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press (1996)
2. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, p. 44. Springer, Heidelberg (2010)
3. ClearSy. Atelier B, User and Reference Manuals. Aix-en-Provence, France (2009), http://www.atelierb.eu/
4. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. TPLP 11(4-5), 767–782 (2011)
5. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
6. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
7. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. Annals of Mathematics and Artificial Intelligence 59(1), 81–106 (2010)
8. Merz, S.: TLA+ Case Study: A Resource Allocator. Technical Report A04-R-101, INRIA Lorraine - LORIA (2004), http://hal.inria.fr/inria-00107809
9. Merz, S., Vanzetto, H.: Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012)
10. Mokhtari, Y., Merz, S.: Animating TLA Specifications. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705, pp. 92–110. Springer, Heidelberg (1999)
11. Mosbahi, O., Jemni, L., Jaray, J.: A formal approach for the development of automated systems. In: Filipe, J., Shishkov, B., Helfert, M. (eds.) ICSOFT (SE), pp. 304–310. INSTICC Press (2007)
12. Plagge, D., Leuschel, M.: Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. STTT 11, 9–21 (2010)
13. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA$^+$ Specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

A A More Complicated Translation: SimpleAllocator

The configuration file for the model below is as follows:
INIT *Init* NEXT *Next* CONSTANTS *Clients* $= \{c1, c2, c3\}$ *Resources* $= \{r1, r2\}$

MODULE *SimpleAllocator*

EXTENDS *FiniteSets*, *TLC*
CONSTANTS *Clients*, *Resources*
ASSUME *IsFiniteSet*(*Resources*)
VARIABLES *unsat*, *alloc*
TypeInvariant $\triangleq$ $\land$ *unsat* $\in$ [*Clients* $\rightarrow$ SUBSET *Resources*]
$\land$ *alloc* $\in$ [*Clients* $\rightarrow$ SUBSET *Resources*]
available $\triangleq$ *Resources* $\setminus$ (UNION $\{$*alloc*[*c*] : *c* $\in$ *Clients*$\}$)

$Init \triangleq \ \wedge\ unsat = [c \in Clients \mapsto \{\}] \wedge alloc = [c \in Clients \mapsto \{\}]$
$Request(c,\ S) \triangleq \ \wedge\ unsat[c] = \{\} \wedge alloc[c] = \{\}$
$\quad \wedge\ S \neq \{\} \wedge unsat' \quad = [unsat \text{ EXCEPT } ![c] = S] \wedge \text{UNCHANGED } alloc$
$Allocate(c,\ S) \triangleq \ \wedge\ S \neq \{\} \wedge S \subseteq available \cap unsat[c]$
$\quad \wedge\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S] \wedge unsat' = [unsat \text{ EXCEPT } ![c] = @ \setminus S]$
$Return(c,\ S) \triangleq \ \wedge\ S \neq \{\} \wedge S \subseteq alloc[c]$
$\quad \wedge\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \setminus S] \wedge \text{UNCHANGED } unsat$
$Next \triangleq \exists\, c \in Clients,\ S \in \text{SUBSET } Resources :$
$\quad Request(c,\ S) \vee Allocate(c,\ S) \vee Return(c,\ S)$

MACHINE SimpleAllocator
SETS $\text{ENUM}_1 = \{r1, r2\}$; $\text{ENUM}_2 = \{c1, c2, c3\}$
CONSTANTS Clients, Resources
PROPERTIES Clients = ENUM_2 ∧ Resources = ENUM_1
∧ ∃seq_.(seq_ ∈ seq(Resources) ∧ ∀s.(s ∈ Resources ⇒
∃n.(n ∈ 1 .. size(seq_) ∧ seq_(n) = s)))
DEFINITIONS
TypeInvariant == unsat ∈ Clients → ℙ(Resources)
∧ alloc ∈ Clients → ℙ(Resources);
available == Resources - union(t_|∃c.(c ∈ Clients ∧ t_ = alloc(c)));
Init == unsat = λc.(c ∈ Clients| {}) ∧ alloc = λc.(c ∈ Clients| {});
Request(c,S) == unsat(c) = {} ∧ alloc(c) = {}
∧ (S ≠ {} ∧ unsat_n = unsat ⩤ {c ↦ S});
Allocate(c,S) == S ≠ {} ∧ S ⊆ available ∩ unsat(c)
∧ alloc_n = alloc ⩤ {c ↦ (alloc(c) ∪ S)}
∧ unsat_n = unsat ⩤ {c ↦ (unsat(c) - S)};
Return(c,S) == S ≠ {} ∧ S ⊆ alloc(c) ∧ alloc_n = alloc ⩤ {c ↦ (alloc(c) - S)};
ResourceMutex == ∀c1,c2.(c1 ∈ Clients ∧ c2 ∈ Clients ⇒
(c1 ≠ c2 ⇒ alloc(c1) ∧ alloc(c2) = {}));
VARIABLES unsat, alloc
INVARIANT unsat ∈ ℙ(ENUM_2×ℙ(ENUM_1))
∧ alloc ∈ ℙ(ENUM_2×ℙ(ENUM_1)) ∧ TypeInvariant ∧ ResourceMutex
INITIALISATION unsat, alloc ∈ (Init)
OPERATIONS
Request_Op(c, S) = ANY unsat_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Request(c, S)
THEN unsat := unsat_n END;
Allocate_Op(c, S) = ANY unsat_n, alloc_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Allocate(c, S)
THEN unsat, alloc := unsat_n, alloc_n END;
Return_Op(c, S) = ANY alloc_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Return(c, S)
THEN alloc := alloc_n END;
END

Rely/Guarantee Reasoning for Teleo-reactive Programs over Multiple Time Bands

Brijesh Dongol[1,2] and Ian J. Hayes[1]

[1] School of Information Technology and Electrical Engineering
The University of Queensland, Australia
[2] Department of Computer Science, The University of Sheffield, UK
{brijesh,Ian.Hayes}@itee.uq.edu.au

Abstract. A complex real-time system consists of components at multiple time abstractions with varying notions of granularity and precision. Existing hybrid frameworks only allow reasoning at a single granularity and at an absolute level of precision, which can be problematic because the models that are developed can become unimplementable. In this paper, we develop a framework that incorporates time bands so that the behaviour of each component may be specified at a time granularity that is appropriate for the component and its properties. We implement our controllers using teleo-reactive programs, which are high-level programs that are well-suited to controlling reactive systems in dynamic environments. We develop rely/guarantee-style reasoning rules and as an example, prove properties of a well-known mine-pump system.

1 Introduction

Autonomous controllers are increasingly being used in safety-critical real-time systems, where failures have a high cost and/or endanger human lives. As the systems under consideration become more complex, we must develop high-level languages and logics to ensure their dependability. In this paper we implement our controllers using Nilsson's teleo-reactive programs [18,7,5] and develop an interval-based logic to formalise its semantics (Section 2). Our logic incorporates the time bands framework [2,3] and includes methods for reasoning about sampling [3,5] (Section 3). We develop rules that allow one to prove that a program executing in an environment formalised by a *rely* condition satisfies a *guarantee* condition (Section 4).

As a motivating example, we consider the well-known mine-pump system [4] (see Fig. 1), where an autonomous controller must read methane and water-level sensors (from the environment) and send signals to the water pump to turn it on or off. To prevent an explosion, the pump must be stopped whenever the level of methane in the mine is above a critical level. The challenge is to show that both the program and its environment (specified over multiple time bands) satisfy the guarantee (which may be specified at time bands different from the program and its environment).

Time Bands. A complex system may consist of multiple components that operate in different time scales and hence reasoning about a component that operates over a coarse-grained time scale (e.g., days) using a finer-grained scale (e.g., seconds) and vice versa over-complicates the reasoning. Furthermore, developing a specification using a single

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 39–53, 2012.

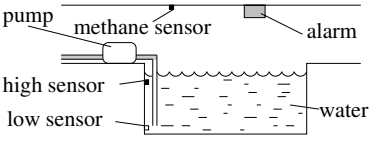


Fig. 1. Mine-pump system

$$\mathsf{mp} \mathrel{\widehat{=}} \mathsf{Out}\, ps, w\_pump\, \mathsf{Init}\, \overrightarrow{ps} = stopped \bullet \left\langle \begin{array}{l} m \geq CT \longrightarrow Alarm \wedge Stop\_Pump\ , \\ true \longrightarrow \mathsf{pump} \end{array} \right\rangle \dagger M$$

$$\mathsf{pump} \mathrel{\widehat{=}} \left\langle \begin{array}{l} high \vee \\ (\neg low \wedge ps_0 = stopped) \end{array} \longrightarrow Run\_Pump\ , \right\rangle \dagger W$$

$$true \longrightarrow Stop\_Pump$$

Fig. 2. Teleo-reactive controller for mine-pump

time scale can lead to difficulties in implementation, where it becomes impossible for any real system to satisfy the timing requirements [21]. In this paper, we use Burns' time bands framework [2,3], which enables one to make allowances for the timing restrictions of an implementation as part of the specification. Within each time band, one may distinguish between *activities*, which may take time within the band, and *events*, which execute within the *precision* of the band (i.e., the amount of time that may be regarded as "instantaneous"). The behaviour of an event of a coarse-grained time band is defined by mapping the event to an activity in a finer-grained time band. The behaviour of an activity may be formalised by using a continuous function on the state or as a sequence of events.

Example 1. In the mine-pump system one may differentiate between the water, pump and methane time bands (denoted W, P and M, respectively). In the pump time band P, turning the pump on/off is an instantaneous event, however, in a finer-grained time band, pump on/off events correspond to activities that specify the pump's acceleration/deceleration. We assume that the methane time band M is finer-grained than that of the water W because there may be a sudden surge of methane in the mine, whereas the water changes at a slower rate.

Sampling. Reactive controllers use (discrete) *sampling events* to determine the state of their (continuous) environments. Sampling events are prone to *timing precision errors* (where there is a range of possible sampled values due to imprecise timing of when the sample is taken) and *sampling anomalies* (where sampling two or more sensors causes a non-existent state to be returned because they are sampled at slightly different times). To simplify the presentation, in this paper, we ignore the possibility of *sensor errors*, where the sensors have inaccuracies in measuring the environment; these are straightforward to incorporate.

Example 2. Consider the three sampling events se_1, se_2 and se_3 in Fig. 3, which respectively correspond to sampling activities sa_1, sa_2 and sa_3, where environment variables x and y are sampled at different times within the interval. Sampling event se_1 will return $x < y$ regardless of when the values of x and y are read within the sampling interval, because $x < y$ *definitely* holds for all sampled values of x and y. Sampling event se_2 may return either $x > y$, $x = y$ or $x < y$ because it is *possibly* true that $x > y$, $x = y$ and $x < y$ hold. Event se_3 may have a sampling anomaly. Although $x > y$ holds throughout sa_3, because x and y are sampled at different times, it is *possible* for sa_3 to return either $x > y$, $x = y$ or $x < y$.

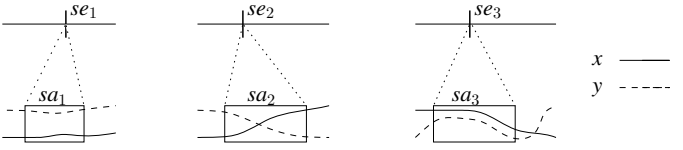


Fig. 3. Sampling events and corresponding activities

Teleo-reactive Programs. Teleo-reactive programs are excellent candidates for implementing controllers for goal-directed agents in dynamically changing real-time environments [7,9,18]. A guarded program $c \rightarrow \mathsf{M}$ executes program M over an interval if the guard c continues to hold over the interval. An ideal execution of a program would continuously evaluate its guards all the time. Of course, continuous evaluation is not feasible and it has to be approximated by repeated sampling and evaluation. One of the issues addressed in this paper is handling the imprecision of such approximations by making use of a time band framework [2,3,5].

For a teleo-reactive program that consists of a prioritised sequence of guarded programs, the first alternative that has a true guard is executed continuously while that guard evaluates to true and no earlier guard in the sequence evaluates to true. As soon as that guard evaluates to false or an earlier guard evaluates to true, execution switches to the first guard in the sequence that evaluates to true.

Example 3. Consider the teleo-reactive program in Fig. 2 that controls the mine-pump in Fig. 1. The program has inputs m (methane level sensor), *high*, *low* (high and low water-level sensors) and ps_0 (fed back value of the pump state) and has outputs ps (pump state) and *w_pump* (total amount of water pumped out). We assume $ps = stopped$ denotes that the pump has physically come to a stop, which we distinguish from the controller sending a stop signal. The main program (mp) assumes $\overrightarrow{ps} = stopped$ holds initially and consists of a sequence of two guarded programs. The meaning of $\overrightarrow{ps} = stopped$ is explained in Section 2.2. Program pump is executed while the methane level m is sampled to be less than the critical threshold CT (i.e., m is sampled so that $m < CT$ holds). In doing so, pump may switch back and forth between its two alternatives, depending on the water level. As soon as the methane is sampled to be not below CT, execution of pump is immediately terminated and control is passed to the first alternative of mp, i.e., $Alarm \wedge Stop\_Pump$. Thus, within a hierarchical teleo-reactive program, the top-level guards take precedence over all lower-level guards.

Programs mp and pump execute in time bands M and W, respectively, which ensures that sampling events of mp and pump take place within the precisions of M and W, respectively. Note that in the top-level program mp, methane monitoring takes place in a time band M, yet mp is able to control a pump that operates in a time band P and execute controller pump, which monitors the water-level sensors in the coarser-grained time band W. The pump can be stopped either because of a high methane level or a low water level, but the controller reacts more quickly to a high methane level.

Related Work. Unlike other formalisms, such as hybrid action systems [19], TLA$^+$ [14] and hybrid automata [12], in which controller actions are assumed to take no time

to execute, the execution of a teleo-reactive action is *durative*, i.e., it takes time. Defining the semantics of durative actions using a discrete model of time such as linear temporal logic [15] is inappropriate. Instead, we consider interval-based logics. An interval temporal logic for discrete traces [17] has been extended to continuous streams to obtain the duration calculus [22]. However, the duration calculus assumes all intervals are closed, allows adjoining intervals to overlap and uses the *almost everywhere* operator. (A state predicate holds almost everywhere in an interval iff it is only false for a set of measure zero.) Almost everywhere is inappropriate for our purposes because we do not fix the absolute precision of a time band, i.e., a single time in any time band may expand to an interval of time at another finer-grained time band. Thus, an almost everywhere property in a time band may not be preserved in finer-grained time bands. An extended duration calculus that uses *everywhere* as opposed to almost everywhere as been defined. However, the framework only allows open intervals [23], and hence is problematic in the context of time bands because the properties at the boundaries of the intervals are undefined. In this paper, we present a logic that is influenced by the duration calculus [22] but is better suited to incorporation with time bands [3].

The idea of structuring systems using multiple abstractions of time (e.g., to handle sampling) is not new. Moszkowski presents a method of abstracting between different time granularities for interval temporal logic using a projection operator, however the framework uses a discrete model of time [17] as opposed to our continuous model. Guelev and Hung present a projection operator for duration calculus where computation is assumed to take time, however, they assume that the time taken is negligible [10]. Montanari et al explore multiple granularities in a real-time context, but the theory is not well developed [16]. Henzinger presents a theory of refinement over multiple granularities where sampling events are executed by a separate process [13]. Broy [1] considers sampling to be an abstraction (via discretisation) of continuous behaviour, however, the framework does not include methods for reasoning about sampling. Burns' time bands theory [3] has been incorporated into Circus [20], however, the focus is on the specification of systems, as opposed to developing real-time controllers.

We have used the logic of sampling and timebands to develop methods of approximating idealised specifications [5] and used the logic of sampling to derive real-time action systems [6].

2 A Real-Time Framework

2.1 Intervals, Streams and Interval Stream Predicates

Our logic is based on non-empty sets of contiguous real numbers (of type *Interval*) which may be open, closed or infinite at either end. Using '.' for function application, we use *glb.SS* and *lub.SS* to refer to the *greatest lower* and *least upper bounds* of a set of numbers *SS*, respectively. For intervals Δ, Δ' we define:

$$\ell.\Delta \mathrel{\widehat{=}} lub.\Delta - glb.\Delta$$
$$\Delta \propto \Delta' \mathrel{\widehat{=}} (lub.\Delta = glb.\Delta') \wedge (\Delta \cup \Delta' \in Interval) \wedge (\Delta \cap \Delta' = \{\})$$

That is, $\ell.\Delta$ denotes the *length* of Δ and $\Delta \propto \Delta'$ states that Δ' is an interval that immediately follows Δ (i.e., Δ *adjoins* Δ'). Within the definition of $\Delta \propto \Delta'$, conjunct

$lub.\Delta = glb.\Delta'$ ensures Δ' follows Δ, conjunct $\Delta \cup \Delta' \in \mathit{Interval}$ ensures $\Delta \cup \Delta'$ is contiguous and conjunct $\Delta \cap \Delta' = \{\}$ ensures that Δ and Δ' are disjoint.

Given that variable names are taken from the set *Var*, a *state space* over a set of variables $V \subseteq \mathit{Var}$ is given by $\Sigma_V \mathrel{\widehat{=}} V \to \mathit{Val}$, which is a total function from variables in V to values in *Val*. We leave out the subscript when the value of V is clear from the context. A *state* is a member of Σ_V, and the stream of behaviours over time of variables in V is given by $\mathit{Stream}_V \mathrel{\widehat{=}} \mathbb{R} \to \Sigma_V$ which is a total function from real numbers to states. A *predicate* over a type X is given by $\mathcal{P}X \mathrel{\widehat{=}} X \to \mathbb{B}$ (e.g., a *stream predicate* is a member of $\mathcal{P}\mathit{Stream}_V$). An interval predicate has type $\mathit{IntvPred}_V \mathrel{\widehat{=}} \mathit{Interval} \to \mathcal{P}\mathit{Stream}_V$. For interval predicates p, p_1 and p_2, we define:

$$(\mathbf{prev}\, p).\Delta \mathrel{\widehat{=}} \exists \Delta' : \mathit{Interval} \bullet (\Delta' \propto \Delta) \wedge p.\Delta'$$
$$(\mathbf{next}\, p).\Delta \mathrel{\widehat{=}} \exists \Delta' : \mathit{Interval} \bullet (\Delta \propto \Delta') \wedge p.\Delta'$$
$$(\boxplus p).\Delta \mathrel{\widehat{=}} \forall \Delta' : \mathit{Interval} \bullet \Delta' \subseteq \Delta \Rightarrow p.\Delta'$$

Thus $(\mathbf{prev}\, p).\Delta$ and $(\mathbf{next}\, p).\Delta$ hold iff p holds in some interval that immediately precedes and follows Δ, respectively and $(\boxplus p).\Delta$ holds iff p holds in all subintervals of Δ. Note that the stream s is implicit in both sides of each of the definitions above.

We assume pointwise lifting of the boolean operators on stream and interval predicates in the normal manner, e.g., if p_1 and p_2 are interval predicates, Δ is an interval and s is a stream, we have $(p_1 \wedge p_2).\Delta.s = (p_1.\Delta.s \wedge p_2.\Delta.s)$. Furthermore, when reasoning about properties of programs, we would like to state that whenever a property p_1 holds over any interval Δ and stream s, a property p_2 also holds over Δ and s. Hence, we define universal implication for interval predicates p_1 and p_2 as $p_1.\Delta \Rrightarrow p_2.\Delta \mathrel{\widehat{=}} \forall s : \mathit{Stream} \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s$ and $p_1 \Rrightarrow p_2 \mathrel{\widehat{=}} \forall \Delta : \mathit{Interval} \bullet p_1.\Delta \Rrightarrow p_2.\Delta$. Both '$\equiv$' and '$\Lleftarrow$' are similarly defined.

2.2 Evaluating State Predicates over an Interval

A state predicate over a set of variables V has type $\mathcal{P}\Sigma_V$. Because there are multiple states in a stream within a non-point interval, there are several possible ways of interpreting the value of a state predicate with respect to a given interval and stream. We use $\lim_{x \to a^+} f.x$ and $\lim_{x \to a^-} f.x$ to denote the limit of $f.x$ as x tends to a from above and below, respectively. To ensure that limits are well defined, we assume all variables are piecewise continuous [8]. For a variable v, a time t and stream s, we use $(v@t).s \mathrel{\widehat{=}} (s.t).v$ to denote the value of v in state $s.t$. For an interval Δ, we define:

$$\overrightarrow{v}.\Delta \mathrel{\widehat{=}} \begin{cases} v@(lub.\Delta) & \text{if } lub.\Delta \in \Delta \\ \lim\limits_{t \to lub.\Delta^-} v@t & \text{otherwise} \end{cases} \qquad \overleftarrow{v}.\Delta \mathrel{\widehat{=}} \begin{cases} v@(glb.\Delta) & \text{if } glb.\Delta \in \Delta \\ \lim\limits_{t \to glb.\Delta^+} v@t & \text{otherwise} \end{cases}$$

Thus, $\overrightarrow{v}.\Delta.s$ denotes the value of v at the right limit of Δ if Δ is right closed and the value of v in s as the value approaches the right limit if Δ is right open.

In an implementation, evaluating a state predicate over an interval takes time, and hence the value of the state predicate returned by an evaluation is dependent on the evaluation strategy used [11]. The simplest evaluation strategy considers the set of states over the interval in which the predicate is evaluated and evaluates the predicate in one of

these states. For a state predicate c and interval Δ, we define the *always* and *sometime* operators as follows[1]:

$$(\circledast c).\Delta \mathrel{\widehat{=}} \forall t{:}\, \Delta \bullet c@t \qquad (\boxdot c).\Delta \mathrel{\widehat{=}} \exists t{:}\, \Delta \bullet c@t$$

Example 4. Consider variable x such that $(x@0) = 10$ and $(\circledast \widehat{x}).[0,2] = 1$ hold, where $\widehat{x}$ denotes the rate of change of variable x (c.f. [12]). Thus, the value of x at time 0 is 10 and the rate of change of x throughout the closed interval $[0,2]$ is 1. Then for adjoining intervals $[0,1)$ and $[1,2]$, both $(\overrightarrow{x} = 11).[0,1)$ and $(\overleftarrow{x} = 11).[1,2]$ hold (because x is continuous), and both $(\circledast(x < 11)).[0,1)$ and $(\circledast(x \geq 11)).[1,2]$ hold.

Thus, the value of a variable at the boundaries of adjoining intervals can be precisely defined. For a set of variables V, we define:

$$\mathbf{st}\, V \mathrel{\widehat{=}} \forall v{:}\, V \bullet \exists k{:}\, \mathit{Val} \bullet \mathbf{prev}(\overrightarrow{v} = k) \wedge \circledast(v = k)$$

Hence, $\mathbf{st}\, V$ holds iff the value of each variable in V is *stable*. Such a definition of invariance is necessary because adjoining intervals are disjoint, and hence $\mathbf{prev}(\overrightarrow{v} = k)$ does not necessarily imply $\overleftarrow{v} = k$, e.g., if v is discrete.

Example 5. We consider the specification of a safety property of the mine pump in Fig. 1. A required safety condition is that $\circledast((m \geq C) \Rightarrow (ps = \mathit{stopped}))$ holds, i.e., in any state of the real-time stream, if the methane level m is above the critical level C, then the state of the pump is such that it is stopped. A program should only be required to satisfy this property if $ps = \mathit{stopped}$ holds initially, hence we obtain:

$$\mathbf{prev}(\overrightarrow{ps} = \mathit{stopped}) \Rightarrow \circledast((m \geq C) \Rightarrow (ps = \mathit{stopped})) \tag{1}$$

Note that C refers to the actual critical value of the methane as opposed to the critical threshold value CT used in the program in Fig. 2. We obtain relationships between the values of C and CT as part of the proof of safety in Section 4.2.

2.3 Chop, Iterated Chop and Alternation

The *chop* operator ';' allows an interval to be split as follows:

$$(p_1\ ;\ p_2).\Delta \mathrel{\widehat{=}} \exists \Delta_1, \Delta_2{:}\, \mathit{Interval} \bullet (\Delta = \Delta_1 \cup \Delta_2) \wedge (\Delta_1 \propto \Delta_2) \wedge p_1.\Delta_1 \wedge p_2.\Delta_2$$

Unlike the duration calculus [22], our chop operator does not restrict intervals Δ_1 and Δ_2 to be closed. Thus, for x as given in Example 4, both $(\circledast(x < 11);\ \circledast(x \geq 11)).[0,2]$ and $(\circledast(x \leq 11)\ ;\ \circledast(x > 11)).[0,2]$ hold, however $(\circledast(x < 11)\ ;\ \circledast(x > 11)).[0,2]$ does not. Using chop we define the *weak chop* and *iterated chop* operators as:

$$p_1 : p_2 \mathrel{\widehat{=}} p_1 \vee (p_1\ ;\ p_2) \qquad p^{\omega} \mathrel{\widehat{=}} \mu z \bullet p : z$$

[1] The notations $\circledast$ and $\boxdot$ follow the nomenclature of Burns and Hayes [3] and should not be confused with temporal operator 'always' ($\Box$). Instead, we ask the reader to focus on the '$*$' (which represents "for all") and '$\cdot$' (which represents "for some") as used in regular expressions. This also applies to $\circledcirc$ and $\odot$ in Section 3.1, both of which should not be confused with temporal operator 'next' ($\bigcirc$).

Thus, $(p_1 : p_2)$ holds iff either p_1 holds for the whole interval or we can split the interval so that $(p_1 \,;\, p_2)$ holds. The iterated chop p^ω is the least fixed point of the weak chop (which defines both finite and infinite iteration of p) assuming that predicates are ordered using $\Leftarrow$.

Because we have a dense notion of time, there is a possibility for an iteration to behave in a Zeno-like manner, where a predicate iterates an infinite number of times within a finite interval. We can rule out Zeno-like behaviour for our implementations because there is a physical lower limit on the time taken to execute each iteration and hence a specification that allows Zeno-like behaviour can be safely ignored. However, we must be careful not to require Zeno-like behaviour, which would cause our specifications to become unimplementable.

We use the iterated chop to define *strict alternation* and *alternation* between interval predicates p_1 and p_2 as follows.

$$p_1 \textbf{ salt } p_2 \mathrel{\widehat{=}} (p_1 : (p_2 \,;\, p_1)^\omega) \vee (p_1 \,;\, p_2)^\omega \qquad p_1 \textbf{ alt } p_2 \mathrel{\widehat{=}} (p_1 \textbf{ salt } p_2) \vee (p_2 \textbf{ salt } p_1)$$

Thus, p_1 **salt** p_2 alternates between p_1 and p_2 starting with p_1 and p_1 **alt** p_2 may alternate between p_1 and p_2 starting with either p_1 or p_2. Proofs of properties that involve chop may be decomposed if the interval under consideration splits and/or joins.

Definition 6. *Interval predicate* p splits *iff* $p \Rrightarrow \boxplus p$ *and* joins *iff* $p^\omega \Rrightarrow p$.

For example, $\circledast c$ both splits and joins, $\boxdot c$ joins but does not split, $\ell < 2$ splits but does not join and $\ell = 2$ neither splits nor joins. In particular, if $(\boxdot c \,;\, \boxdot c).\Delta$, then $(\boxdot c).\Delta$ must hold, but if $(\boxdot c).\Delta$, there may be a subinterval Δ' of Δ such that $(\circledast \neg c).\Delta'$ holds.

Lemma 7. *If* p *splits then* $p \wedge (p_1 \textbf{ alt } p_2) \Rrightarrow ((p \wedge p_1) \textbf{ alt } (p \wedge p_2))$.

3 Multi-time-Band Systems

3.1 Sampling and Time Bands

Although $\circledast$ and $\boxdot$ accommodate for the fact that expression evaluation takes time, they implicitly assume that a snapshot of the entire state is taken when an expression is evaluated, which may not be implementable because variables are usually sampled at different times [11]. Real-time controllers usually evaluate expressions by sampling the environment variables that occur in the expression once, then evaluate the expression using these sampled values. Note that if a variable occurs multiple times within an expression, the same sampled value is used for all occurrences of the variable. Hence, expression $x = x$ is guaranteed to evaluate to true, however, $x > y$ may evaluate to false even if $\circledast(x > y)$ holds as in sa_3 of Fig. 3 (cf [3]). For a set of states $SS \subseteq \Sigma_V$, we define the apparent states over SS as $app.SS \mathrel{\widehat{=}} \{\sigma : \Sigma \mid \forall v : V \bullet \sigma.v \in \{\sigma : SS \bullet \sigma.v\}\}$, where $\{\sigma : SS \bullet \sigma.v\}$ is equivalent to $\{x : Val \mid \exists \sigma : SS \bullet x = \sigma.v\}$. Thus, $\{\sigma : SS \bullet \sigma.v\}$ maps each variable v in SS to its set of values in the set of states SS, and hence $app.SS$ represents the set of possible states that a sampling event may observe. Evaluating an expression in one of the states of $app.SS$ represents a possible evaluation of the expression. The set of all states that occur within an interval Δ of stream s is given by

$states.\Delta.s \mathrel{\widehat{=}} \{t{:}\,\Delta \bullet s.t\}$ and we formalise state predicates that are *definitely* true (denoted $\circledast$) and *possibly* true (denoted $\odot$) over a given interval Δ and stream s as follows:

$$(\circledast c).\Delta.s \mathrel{\widehat{=}} \forall \sigma{:}\, app.(states.\Delta.s) \bullet c.\sigma \qquad (\odot c).\Delta.s \mathrel{\widehat{=}} \exists \sigma{:}\, app.(states.\Delta.s) \bullet c.\sigma$$

That is, $(\circledast c).\Delta.s$ and $(\odot c).\Delta.s$ hold iff c holds in each and some apparent state of s within Δ, respectively. Note that $\neg\circledast c \equiv \odot\neg c$ and $\odot(c \wedge d) \Rrightarrow \odot c \wedge \odot d$. As an example, if Δ_1, Δ_2 and Δ_3 correspond to sa_1, sa_2 and sa_3 in Fig. 3, respectively, both $\circledast(x < y).\Delta_1$ and $(\odot(x < y) \wedge \odot(x \geq y)).\Delta_2$ hold. For sa_3 (which has a sampling anomaly), both $\odot(x > y).\Delta_3$ and $\boxplus(x > y).\Delta_3$ hold, but $\circledast(x > y).\Delta_3$ does not, i.e., $\neg\circledast(x > y).\Delta_3$ holds, which is equivalent to stating that $\odot(x \leq y).\Delta_3$ holds.

We let $vars.c$ denote the set of all variables V that occur (free) in $c \in \mathcal{P}\Sigma_V$ and obtain the following lemma.

Lemma 8. *For a state predicate c, $\circledast c \Rrightarrow \boxplus c$ and $\boxdot c \Rrightarrow \odot c$ hold [3]. Furthermore, for any variable v, $\mathbf{st}(vars.c \backslash \{v\}) \Rrightarrow (\circledast c = \boxplus c) \wedge (\odot c = \boxdot c)$ [11].*

The set of all time bands is given by the primitive type *TimeBand* [3]. In this paper, we are mainly interested in the *precision* of a time band, which defines a constraint on the duration of an instantaneous event of the time band. Hence, we define $\rho{:}\, TimeBand \to \mathbb{R}^{>0}$ to be a function that returns the precision of the given time band. We define the type of a *time band predicate* as $TBPred_V{:}\, TimeBand \to IntvPred_V$, which for a given time band returns an interval predicate. As with interval predicates, we assume time band predicates are lifted pointwise over boolean operators and for time band predicates tp_1 and tp_2, we define universal implication $tp_1 \Rrightarrow tp_2 \mathrel{\widehat{=}} \forall \beta{:}\, TimeBand \bullet tp_1.\beta \Rrightarrow tp_2.\beta$ (and similarly $\Lleftarrow$ and $\equiv$).

Real-time controllers use approximate values of environment variables using sampling events [5]. We relate the actual and sampled values of a real-valued variable, say v, in a time band, say b, using $accuracy.v.b$, which describes the maximum possible change to v within events of time band b. For an interval Δ and stream s, we define:

$$(diff.v).\Delta.s \mathrel{\widehat{=}} \mathsf{let}\ vs = \{\sigma{:}\, states.\Delta.s \bullet \sigma.v\}\ \mathsf{in}\ lub.vs - glb.vs$$

which returns the difference between the greatest and least values of v in s within Δ. The maximum possible change to v within a time band is an assumption on the behaviour of v. In this paper, for any real-valued variable v, we implicitly assume a rely condition:

$$\forall \beta{:}\, Timeband \bullet \boxplus(\ell \leq \rho.\beta \Rightarrow diff.v \leq accuracy.v.\beta) \tag{2}$$

That is, we assume that in any time band β, the maximum possible change to v within events of time band β is bounded above by $accuracy.v.\beta$.

For a state predicate c, we define the following time band predicates to simplify reasoning about repeated sampling.

$$\odot_\rho c \mathrel{\widehat{=}} (\ell \leq \rho) \wedge \odot c \qquad (\mathsf{eval}\, c).\beta \mathrel{\widehat{=}} (\odot_{\rho.\beta} c)^\omega$$

Hence, $(\odot_\rho c).\Delta$ holds iff the length of Δ is at most ρ and c possibly holds in Δ. Predicate $(\mathsf{eval}\, c).\beta$ holds iff $\odot_{\rho.\beta} c$ holds iteratively. We assume that it takes time for the a variable v to be set to a new (constant) value k. Thus, we define the following time band predicate.

$$\mathsf{set}(v,k) \mathrel{\widehat{=}} \lambda\beta \bullet (\mathbf{prev}(\overrightarrow{v} = k) \Rightarrow \circledast(v = k)) \wedge (\ell < \rho.\beta : \circledast(v = k))$$

Thus, $(\mathsf{set}(v,k)).\beta$ holds iff $v = k$ is invariant and if $v = k$ does not hold at the start, then there is a delay of at most $\rho.\beta$ before $\circledast(v = k)$ is established. Note that if $(\mathsf{set}(v = k) \wedge \mathbf{prev}(\overrightarrow{v} \neq k) \wedge (\ell < \rho.\beta)).\Delta$, then $v = k$ may not be established within Δ.

We combine the time band predicates above and obtain the following lemma.

Lemma 9. *If v is a real-valued variable and k is a constant, then* $\mathsf{eval}(v < k) \Rrightarrow \circledast(v < k + accuracy.v)$.

3.2 Teleo-reactive Programs

Definition 10. *For an interval predicate p, set of variables* $V \subseteq Var$*, time band* β *and state predicate c, the syntax of a* teleo-reactive program *is given by P, where*

$$P ::= (\mathsf{Out}\ V\ \mathsf{Init}\ p \bullet SP) \mid P_1 \overrightarrow{\parallel} P_2 \qquad SP ::= p \mid \mathrm{seq}.GP \dagger \beta \qquad GP ::= c \rightarrow SP$$

Thus, $\mathsf{Out}\ V\ \mathsf{Init}\ p \bullet SP$ denotes a program with output variables V that begins executing immediately after an interval satisfying p and $P_1 \overrightarrow{\parallel} P_2$ denotes the parallel composition of programs P_1 and P_2. A simple program (of type SP) may either be an interval predicate or a sequence of guarded simple programs within a time band.

We follow the convention of using T for a teleo-reactive program, M for a simple program and S and T for sequences of guarded simple programs. Sequences are written within brackets '$\langle$' and '$\rangle$' and '$\frown$' is used for sequence concatenation. Given that $vars.p$ denotes the set of all variables that occur free in interval predicate p, the sets of inputs and outputs of a programs $\mathsf{T} \mathrel{\widehat{=}} \mathsf{Out}\ V\ \mathsf{Init}\ p \bullet \mathsf{M}$ and $\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2$ are defined as follows:

$$in.\mathsf{T} \mathrel{\widehat{=}} cin.V.\mathsf{M} \qquad out.\mathsf{T} \mathrel{\widehat{=}} V$$
$$in.(\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2) \mathrel{\widehat{=}} in.\mathsf{T}_1 \cup (in.\mathsf{T}_2 \backslash out.\mathsf{T}_1) \qquad out.(\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2) \mathrel{\widehat{=}} out.\mathsf{T}_1 \cup out.\mathsf{T}_2$$

where
$$cin.V.p \mathrel{\widehat{=}} vars.p \backslash V$$
$$cin.V.\langle\rangle \mathrel{\widehat{=}} \{\}$$
$$cin.V.((\langle c \rightarrow \mathsf{M}\rangle \frown S) \dagger \beta) \mathrel{\widehat{=}} vars.c \cup cin.V.\mathsf{M} \cup cin.V.S$$

We let $vars.\mathsf{T} \mathrel{\widehat{=}} in.\mathsf{T} \cup out.\mathsf{T}$ denote the set of all variables of program T. For any teleo-reactive program T, we require that $in.\mathsf{T} \cap out.\mathsf{T} = \{\}$, i.e., the inputs and outputs of the program are distinct. Two programs executing in parallel may not modify the same outputs. Hence, we require $out.\mathsf{T}_1 \cap out.\mathsf{T}_2 = \{\}$ for any program $\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2$. Note that parallel composition is not necessarily commutative because the outputs of T_1 may be used as inputs to T_2, and hence $\mathsf{T}_2 \overrightarrow{\parallel} \mathsf{T}_1$ may not be well-defined. Furthermore, because $in.(\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2) \cap out.(\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2) = \{\}$ is assumed, $in.\mathsf{T}_1 \cap out.\mathsf{T}_2 = \{\}$ holds.

Definition 11. *Suppose p is an interval predicate, c is a state predicate, S and T are a sequences of guarded programs,* β *is a time band, V is a set of variables,* M *is a simple program and* T_1 *and* T_2 *are a teleo-reactive programs. We define:*

$$beh.p \mathrel{\widehat{=}} p \tag{3}$$

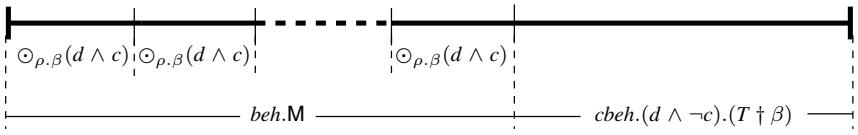


Fig. 4. $cbeh.d.((\langle c \rightarrow \mathsf{M}\rangle \frown T) \dagger \beta)$

$$beh.(S \dagger \beta) \mathrel{\widehat{=}} cbeh.true.(S \dagger \beta) \tag{4}$$

$$beh.(\mathsf{Out}\, V\, \mathsf{Init}\, p \bullet \mathsf{M}) \mathrel{\widehat{=}} \mathbf{prev}\, p \wedge beh.\mathsf{M} \tag{5}$$

$$beh.(\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2) \mathrel{\widehat{=}} beh.\mathsf{T}_1 \wedge beh.\mathsf{T}_2 \tag{6}$$

where:

$$cbeh.d.(\langle\rangle \dagger \beta) \mathrel{\widehat{=}} (\mathsf{eval}\, d).\beta$$
$$cbeh.d.(\langle c \rightarrow \mathsf{M}\rangle \frown T \dagger \beta) \mathrel{\widehat{=}} ((\mathsf{eval}(d \wedge c)).\beta \wedge beh.\mathsf{M})\ \mathbf{alt}\ cbeh.(d \wedge \neg c).(T \dagger \beta)$$

By (3), the behaviour of an action p is given by p itself. By (4), the behaviour of a sequence of guarded programs is defined using the *contextual behaviour* function *cbeh* (see Fig. 4), where the contextual behaviour of an empty sequence is the iterated evaluation of the context in time band β, and the contextual behaviour of a non-empty sequence $(\langle c \rightarrow \mathsf{M}\rangle \frown T) \dagger \beta)$ in context d alternates between execution of M with $d \wedge c$ evaluated iteratively in time band β and execution of $T \dagger \beta$ in context $d \wedge \neg c$. Such a definition is necessary because the negations of the earlier guards become conjuncts to later guards. By using the contextual behaviour function, we ensure that the variables of the context are evaluated in the same apparent state as those of the guard, i.e., we evaluate $\odot(d \wedge c)$ as opposed to the weaker interval predicate $\odot d \wedge \odot c$. By (5), the behaviour of $\mathsf{Out}\, V\, \mathsf{Init}\, p \bullet \mathsf{M}$ holds if $beh.\mathsf{M}$ holds immediately after an interval that satisfies p. By (6), the behaviour of the parallel composition of two programs is defined to be the conjunction of both behaviours.

Example 12. Fig. 2 provides a teleo-reactive controller for the mine pump in Fig. 1. We now further specify the controller by formalising its (durative) actions *Run_Pump* and *Stop_Pump*. The water level w is controlled by both the environment and the pump, thus, w may not be a direct output of either. Instead, we split w into w_env and w_pump, which denote the total amount of water added and removed from the mine by the environment and water pump, respectively. Hence, the water level is given by $w = w\_env - w\_pump$. To formalise the specification of the pump, we use:

$$\circledast(\widehat{w\_pump} \geq 0) \tag{7}$$

$$\circledast(ps = running) \Rightarrow \boxplus(\ell \geq \rho.W \Rightarrow (accuracy.w\_env.W < diff.w\_pump)) \tag{8}$$

$$\circledast(ps = stopped) \Rightarrow \circledast(\widehat{w\_pump} = 0) \tag{9}$$

Condition (7) states that the rate of change of w_pump is non-negative (i.e., water never flows back through the pump). Condition (8) states that if the pump is running, then in any interval whose length is at least $\rho.W$, the maximum difference between any two

w_pump values in the interval must be above the accuracy of *w_env* in time band W. Because we implicitly assume that the maximum change to *w_env* within any event of time band W is bounded above by $accuracy.w\_env.W$ (see (2)), the water level is guaranteed to reduce if the pump is running and (8) holds. Similarly, over any interval in which the pump is stopped, *w_pump* does not change (9).

The durative actions that run and stop the pump are specified as follows.

$$Run\_Pump \mathrel{\widehat{=}} (\mathsf{set}(ps, running)).P \wedge (7) \wedge \boxplus(8)$$
$$Stop\_Pump \mathrel{\widehat{=}} (\mathsf{set}(ps, stopped)).P \wedge (7) \wedge \boxplus(9)$$

If *Run_Pump* is executing then $\mathsf{set}(ps, running)$ holds for the pump time band P, i.e., $ps = running$ is invariant, or $ps = running$ becomes true within the precision of P. Furthermore, *w_pump* changes as described by (7) and $\boxplus$(8). The specification of *Stop_Pump* is similar.

Note that our specifications of *Run_Pump* and *Stop_Pump* are quite general and do not overly restrict the manner in which the pump starts/stops or the manner in which the pump modifies *w_pump* while the pump is running. For example, if $\mathbf{prev}(\overrightarrow{ps} \neq running)$ and $\mathsf{set}(ps, running).P$ hold, an implementation must only guarantee that $ps = running$ becomes true within the precision of time band P and the acceleration/deceleration of the pump is unspecified. By (8), while running, the pump is only required to guarantee a certain throughput in the precision of time band W and the instantaneous rate of change of *w_pump* may vary (c.f. the stricter requirement $\circledast(\widehat{w\_pump} = 0)$ in (9)). Hence, there are several possible implementations of the pump specification at higher-precision time bands (e.g., piston-driven versus centrifugal pumps).

4 Rely/Guarantee Reasoning

4.1 Rely/Guarantee Rules

Teleo-reactive programs are often only required to execute correctly under certain environment assumptions; these assumptions may be formalised within a rely condition. To prevent circular reasoning, a program must not depend on its own output, and hence, we say interval predicate r is a *rely condition* of program T iff $vars.r \cap out.\mathsf{T} = \{\}$.

Definition 13. *For a teleo-reactive program* T *with rely condition* r*, and interval predicate* g *(representing the* guarantee *of* T*), we define* $\{r\}\,\mathsf{T}\,\{g\} \mathrel{\widehat{=}} r \wedge beh.\mathsf{T} \Rrightarrow g$.

Thus, $\{r\}\,\mathsf{T}\,\{g\}$ states that if the environment behaves as specified by rely condition r and the program T behaves as specified by $beh.\mathsf{T}$, then the guarantee condition g holds. We obtain some straightforward properties for proving rely/guarantee properties.

$$(r \Rrightarrow r_1) \wedge \{r_1\}\,\mathsf{T}\,\{g_1\} \wedge (g_1 \Rrightarrow g) \Rightarrow \{r\}\,\mathsf{T}\,\{g\} \tag{10}$$
$$\{r\}\,\mathsf{T}\,\{g_1\} \wedge \{r\}\,\mathsf{T}\,\{g_2\} \Rightarrow \{r\}\,\mathsf{T}\,\{g_1 \wedge g_2\} \tag{11}$$
$$\{r_1\}\,\mathsf{T}\,\{g\} \wedge \{r_2\}\,\mathsf{T}\,\{g\} \Rightarrow \{r_1 \vee r_2\}\,\mathsf{T}\,\{g\} \tag{12}$$
$$\{r_1 \wedge r_2\}\,\mathsf{T}\,\{g\} = \{r_1\}\,\mathsf{T}\,\{r_2 \Rightarrow g\} \tag{13}$$

Because the behaviour of $\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2$ is defined to be the conjunction of the behaviours of T_1 and T_2 (see (6)), rely/guarantee reasoning for parallel composition may be decomposed in a straightforward manner.

Theorem 14. $\{r_1\}\,\mathsf{T}_1\,\{g_1\} \wedge \{r_2\}\,\mathsf{T}_2\,\{g_1 \Rightarrow g_2\} \Rightarrow \{r_1 \wedge r_2\}\,\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2\,\{g_1 \wedge g_2\}$

Proof.
$\{r_1\}\,\mathsf{T}_1\,\{g_1\} \wedge \{r_2\}\,\mathsf{T}_2\,\{g_1 \Rightarrow g_2\}$
$=$ definition and logic
$(r_1 \wedge \mathit{beh}.\mathsf{T}_1 \Rrightarrow g_1) \wedge (r_2 \wedge \mathit{beh}.\mathsf{T}_2 \Rrightarrow (g_1 \Rightarrow g_2))$
$\Rightarrow$ logic, weaken antecedents
$r_1 \wedge r_2 \wedge \mathit{beh}.\mathsf{T}_1 \wedge \mathit{beh}.\mathsf{T}_2 \Rrightarrow g_1 \wedge (g_1 \Rightarrow g_2)$
$=$ (6), definitions and logic
$\{r_1 \wedge r_2\}\,\mathsf{T}_1 \overrightarrow{\parallel} \mathsf{T}_2\,\{g_1 \wedge g_2\}$ □

Within a program $\mathsf{P} \mathrel{\widehat{=}} \mathsf{Out}\,V\,\mathsf{Init}\,p \bullet \mathsf{M}$, the simple programs contained in M (including M itself) execute within an *output context* V. To prevent circular reasoning, we disallow the rely conditions of a simple program from referring to the variables of its output context. Thus, for any simple program M with output context V, interval predicate r is a *rely condition* of M iff $\mathit{vars}.r \cap V = \{\}$. For a simple program M with rely condition r and an interval predicate g, we define $\{r\}\,\mathsf{M}\,\{g\} \mathrel{\widehat{=}} r \wedge \mathit{beh}.\mathsf{M} \Rrightarrow g$. Each of the properties (10)-(13) with T replaced by M holds. Furthermore, we have the following straightforward property for a conjunction of actions p_1 and p_2.

$$\{r\}\,p_1\,\{g\} \vee \{r\}\,p_2\,\{g\} \Rightarrow \{r\}\,(p_1 \wedge p_2)\,\{g\} \tag{14}$$

For any $S \in \mathit{seq}.GP$, the effective guard of any branch contains the negations of all preceding guards as conjuncts. We use functions $\mathit{grd}.(c \rightarrow \mathsf{M}) \mathrel{\widehat{=}} c$ and $\mathit{body}.(c \rightarrow \mathsf{M}) \mathrel{\widehat{=}} \mathsf{M}$ to return the guard and body of the guarded simple program $c \rightarrow \mathsf{M}$, respectively. For any $S \in \mathrm{seq}.GP$ and $i \in \mathrm{dom}.S$, the *effective guard* of $S.i$ is given by

$$\mathit{eff}.S.i \mathrel{\widehat{=}} \mathit{grd}.(S.i) \wedge \textstyle\bigwedge_{j:0..i-1} \neg \mathit{grd}.(S.j)$$

i.e., the effective guard of $S.i$ is the actual guard of $S.i$ in conjunction with the negations of all guards that precede i in S.

Lemma 15. *For any $S \in \mathrm{seq}.GP$ and $i, j \in \mathrm{dom}.S$, $i \neq j \wedge \mathit{eff}.S.i \Rightarrow \neg \mathit{eff}.S.j$.*

The following theorem allows one to decompose a proof of a sequence of guarded teleo-reactive programs. We let $\mathit{ALT}_{i:0..n}\ X_i \mathrel{\widehat{=}} X_0\ \mathbf{alt}\ X_1\ \mathbf{alt} \cdots \mathbf{alt}\ X_n$ and $\mathit{exec}.S.i.\beta \mathrel{\widehat{=}} (\mathsf{eval}\,\mathit{eff}.S.i).\beta \wedge \mathit{beh}.(\mathit{body}.(S.i))$.

Theorem 16. *If r and g are interval predicates such that r splits and g joins, then $\{r\}\ \mathsf{Out}\,V\,\mathsf{Init}\,p \bullet S \dagger \beta\,\{g\}$ holds provided that for each $i \in \mathrm{dom}.S$:*

$$\{r \wedge (\mathsf{eval}\,\mathit{eff}.S.i).\beta\}\ \mathit{body}.(S.i)\ \{\mathbf{prev}(p \vee \odot_{\rho.\beta} \neg \mathit{eff}.S.i) \Rightarrow g\} \tag{15}$$

Proof.
$\{r\}\ \mathsf{Out}\,V\,\mathsf{Init}\,p \bullet S \dagger \beta\,\{g\}$
$=$ expand rely/guarantee triple and $\mathit{beh}.(S \dagger \beta)$
$r \wedge \mathbf{prev}\,p \wedge (\mathit{ALT}_{i:\mathrm{dom}.S}\ \mathit{exec}.S.i.\beta) \Rrightarrow g$
$\Leftarrow$ r splits Lemma 7, **alt** is monotonic
$\mathbf{prev}\,p \wedge (\mathit{ALT}_{i:\mathrm{dom}.S}\ r \wedge \mathit{exec}_V.S.i.\beta) \Rrightarrow g$
$\Leftarrow$ Lemma 15 and logic

$$
\begin{array}{ll}
 & (ALT_{i:\mathrm{dom}.S}\ r \wedge \mathbf{prev}(p \vee \odot_{\rho.\beta} \neg \mathit{eff}.S.i) \wedge \mathit{exec}.S.i.\beta) \Rrightarrow g \\
\Leftarrow & \quad (15) \\
 & (ALT_{i:\mathrm{dom}.S}\ g) \Rrightarrow g \\
= & \quad (g\ \mathbf{alt}\ g) \equiv g^{\omega} \text{ and } (g^{\omega}\ \mathbf{alt}\ g) \equiv g^{\omega} \\
 & g^{\omega} \Rrightarrow g \\
\Leftarrow & \quad g \text{ joins} \\
 & \mathit{true}
\end{array}
$$

□

Here, p specifies an initial assumption and $S \dagger \beta$ implements the controller. By (15), if the rely r holds and program $\mathit{body}.(S.j)$ executes, then if the guard of $S.j$ possibly holds in time band β, and either there is a previous interval in which p holds or there is a previous sampling interval in which $\neg \mathit{grd}.(S.j)$ can be sampled, then g holds.

4.2 Proof of Safety for mp

We prove $\{\mathit{true}\}\ \mathsf{mp}\ \{(1)\}$ i.e., under the implicit rely condition that the maximum difference between any two values of m within an event (including sampling events) of time bands M and P are bounded by the accuracy of m in M and P, respectively (see (2)), execution of program mp satisfies the safety condition (1). Assuming $\mathit{Stopped} \mathrel{\widehat{=}} (ps = \mathit{stopped})$ and $\overrightarrow{\mathit{Stopped}} \mathrel{\widehat{=}} (\overrightarrow{ps} = \mathit{stopped})$, we have:

$$
\begin{array}{ll}
 & \{\mathit{true}\}\ \mathsf{mp}\ \{(1)\} \\
\Leftarrow & \quad \text{Theorem 16 then weaken rely/strengthen guarantee (10) and then (14)} \\
 & \{\mathit{true}\}\ \mathit{Stop\_Pump}\ \left\{ \begin{array}{l} \mathbf{prev}(\overrightarrow{\mathit{Stopped}} \vee \odot_{\rho.M}(m < CT)) \Rightarrow \\ \qquad \circledast(m \geq C \Rightarrow \mathit{Stopped}) \end{array} \right\} \wedge \qquad (A) \\
 & \{(\mathsf{eval}(m < CT)).M\}\ \mathsf{pump}\ \{\circledast(m \geq C \Rightarrow \mathit{Stopped})\} \qquad (B)
\end{array}
$$

For the conjunct (B) above, noting that $\circledast(c \Rightarrow d)$ is implied by $\boxdot c \Rightarrow \circledast d$, assuming $CT < C - \mathit{accuracy}.m.M$ we obtain

$$
\begin{array}{ll}
 & \{(\mathsf{eval}(m < CT)).M\}\ \mathsf{pump}\ \{\boxdot(m \geq C) \Rightarrow \circledast \mathit{Stopped}\} \\
\Leftarrow & \quad \text{Lemma 9 and (13)} \\
 & \{\circledast(m < C) \wedge \boxdot(m \geq C)\}\ \mathsf{pump}\ \{\circledast \mathit{Stopped}\} \\
= & \quad \text{rely simplifies to } \mathit{false} \\
 & \mathit{true}
\end{array}
$$

For conjunct (A) above, because $\mathbf{prev}(p_1 \vee p_2) \equiv \mathbf{prev}\, p_1 \vee \mathbf{prev}\, p_2$, we obtain:

$$\{\mathit{true}\}\ \mathit{Stop\_Pump}\ \{\mathbf{prev}\ \overrightarrow{\mathit{Stopped}} \Rightarrow \circledast(m \geq C \Rightarrow \mathit{Stopped})\} \qquad (16)$$

$$\{\mathit{true}\}\ \mathit{Stop\_Pump}\ \{\mathbf{prev}(\odot_{\rho.M}(m < CT)) \Rightarrow \circledast(m \geq C \Rightarrow \mathit{Stopped})\} \qquad (17)$$

For (16), because $\mathit{beh}.\mathit{Stop\_Pump}$ implies $\mathit{Stopped}$ is invariant and because $\mathit{Stopped}$ holds initially, we have $\circledast \mathit{Stopped}$ and hence $\circledast(m \geq C \Rightarrow \mathit{Stopped})$. For (17), we strengthen our assumption on CT so that $CT < C - \mathit{accuracy}.m.M - \mathit{accuracy}.m.P$ and obtain the following.

$$
\begin{array}{ll}
 & \mathbf{prev}(\odot_{\rho.M}(m < CT)) \Rightarrow \circledast(m \geq C \Rightarrow \mathit{Stopped}) \\
\Leftarrow & \quad \text{strengthen consequent}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{prev}(\odot_{\rho.M}(m < CT)) \Rightarrow ((\ell < \rho.P) : \circledast Stopped) \wedge \circledast(m \geq C \Rightarrow Stopped) \\
\Leftarrow & \quad \circledast c \text{ joins and logic} \\
 & \mathbf{prev}(\odot_{\rho.M}(m < CT)) \Rightarrow ((\ell < \rho.P \wedge \circledast(m \geq C \Rightarrow Stopped)) : \circledast Stopped) \\
\Leftarrow & \quad \text{Lemma 9 and logic} \\
 & \overleftarrow{m} < CT + accuracy.m.M \Rightarrow ((\ell < \rho.P \wedge \circledast(m < C)) : \circledast Stopped) \\
\Leftarrow & \quad \text{assumption } CT < C - accuracy.m.M - accuracy.m.P \\
 & \overleftarrow{m} < C - accuracy.m.P \Rightarrow ((\ell < \rho.P \wedge \circledast(m < C)) : \circledast Stopped) \\
\Leftarrow & \quad \text{Lemma 9} \\
 & (\ell < \rho.P) : \circledast Stopped \\
\Leftarrow & \ beh.Stop\_Pump
\end{array}
$$

5 Conclusions

We have presented an interval-based framework for specifying real-time systems with components that operate over multiple time bands and rely/guarantee-style rules for reasoning about such systems. The controllers are implemented using teleo-reactive programs [18,7], which allow reactive real-time controllers to be implemented in a straightforward manner.

Unlike existing hybrid methods (e.g., hybrid automata [12], TLA$^+$ [14], hybrid action systems [19]) where controller actions are assumed to take no time to execute, teleo-reactive programs execute durative actions. The durative actions are much closer to an abstract specification, which allows programs to be describe their intended behaviour. Sequences of guarded programs are structured in a goal-directed manner [18,7] and hierarchical nesting of programs is incorporated into the abstract syntax. We model true concurrency between a controller and its environment then map "instantaneous" sampling events to a higher precision time band to enable sampling anomalies to be taken into consideration. By incorporating a time bands framework, we may distinguish between fine and coarse grained sampling and develop programs that take both into account. We also accommodate the time taken to execute start and stop events of the pump in our reasoning.

As an example, we have specified and proved properties of the well-known mine-pump example. An unexpected aspect of our controller is that the time band of the top-level controller, which is monitoring for a critical level of methane, is finer grained than the time band of its component water monitoring controller. Interestingly, our approach handles this unexpected time band structure because we allow the time band of the top-level controller's guard evaluation to differ arbitrarily from the time bands of its component actions.

Teleo-reactive programs are much closer to specifications than action systems [6], which are much closer to code. As future work, we aim to further explore this link to simplify development of real-time controllers.

Acknowledgements. This research is supported by ARC Discovery Grant DP0987452 and EPSRC Grant EP/J003727/1 *Verifying Lock-Free Algorithms*.

References

1. Broy, M.: Refinement of time. Theor. Comput. Sci. 253(1), 3–26 (2001)

2. Burns, A., Baxter, G.: Time bands in systems structure. In: Structure for Dependability, pp. 74–88. Springer (2006)
3. Burns, A., Hayes, I.J.: A timeband framework for modelling real-time systems. Real-Time Systems 45(1), 106–142 (2010)
4. Burns, A., Lister, A.M.: A framework for building dependable systems. Comput. J. 34(2), 173–181 (1991)
5. Dongol, B., Hayes, I.J.: Approximating idealised real-time specifications using time bands. In: AVoCS 2011. ECEASST, vol. 46, pp. 1–16. EASST (2012)
6. Dongol, B., Hayes, I.J.: Deriving real-time action systems in a sampling logic. Sci. Comput. Program. (Special Issue of MPC 2010) (2012) (accepted October 17, 2011)
7. Dongol, B., Hayes, I.J., Robinson, P.J.: Reasoning about real-time teleo-reactive programs. Technical Report SSE-2010-01, The University of Queensland (2010)
8. Gargantini, A., Morzenti, A.: Automated deductive requirements analysis of critical systems. ACM Trans. Softw. Eng. Methodol. 10, 255–307 (2001)
9. Gubisch, G., Steinbauer, G., Weiglhofer, M., Wotawa, F.: A Teleo-Reactive Architecture for Fast, Reactive and Robust Control of Mobile Robots. In: Nguyen, N.T., Borzemski, L., Grzech, A., Ali, M. (eds.) IEA/AIE 2008. LNCS (LNAI), vol. 5027, pp. 541–550. Springer, Heidelberg (2008)
10. Guelev, D.P., Hung, D.V.: Prefix and projection onto state in duration calculus. Electr. Notes Theor. Comput. Sci. 65(6), 101–119 (2002)
11. Hayes, I.J., Burns, A., Dongol, B., Jones, C.: Comparing models of nondeterministic expression evaluation. Technical Report CS-TR-1273, Newcastle University (2011)
12. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE Computer Society, Washington, DC (1996)
13. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Assume-Guarantee Refinement Between Different Time Scales. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 208–221. Springer, Heidelberg (1999)
14. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
15. Manna, Z., Pnueli, A.: Temporal Verification of Reactive and Concurrent Systems: Specification. Springer-Verlag New York, Inc. (1992)
16. Montanari, A., Ratto, E., Corsetti, E., Morzenti, A.: Embedding time granularity in logical specifications of real-time systems. In: Euromicro 1991, pp. 88–97 (June 1991)
17. Moszkowski, B.C.: Compositional reasoning about projected and infinite time. In: ICECCS, pp. 238–245. IEEE Computer Society (1995)
18. Nilsson, N.J.: Teleo-reactive programs and the triple-tower architecture. Electronic Transactions on Artificial Intelligence 5, 99–110 (2001)
19. Rönkkö, M., Ravn, A.P., Sere, K.: Hybrid action systems. Theor. Comput. Sci. 290, 937–973 (2003)
20. Wei, K., Woodcock, J., Burns, A.: Formalising the timebands model in timed Circus. Technical report, University of York (June 2010)
21. Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Form. Methods Syst. Des. 33, 45–84 (2008)
22. Zhou, C., Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. EATCS: Monographs in Theoretical Computer Science. Springer (2004)
23. Zhou, C., Ravn, A.P., Hansen, M.R.: An Extended Duration Calculus for Hybrid Real-Time Systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 36–59. Springer, Heidelberg (1993)

Safety and Line Capacity in Railways – An Approach in Timed CSP

Yoshinao Isobe[1], Faron Moller[2],
Hoang Nga Nguyen[2], and Markus Roggenbach[2,⋆]

[1] AIST, Japan
[2] Swansea University, UK

Abstract. Railways need to be safe and, at the same time, should offer high capacity. While the notion of safety is well understood in the railway domain, the meaning of capacity is understood only on an intuitive and informal level. In this study, we show how to define and analyse capacity in a rigorous way. Our modelling approach builds on an established modelling technique in the process algebra CSP for safety alone, provides an integrated view on safety as well as capacity, and offers proof support in terms of (untimed) model checking.

1 Introduction

Overcoming the constraints on railway capacity caused by nodes (stations and junctions) on the rail network is one of the most pressing challenges to the rail industry. In 2007, the UK governmental White Paper "Delivering a Sustainable Railway" [9] stated: *"Rail's biggest contribution to tackling global warming comes from increasing its capacity."* High capacity, however, is but one design aim within the railway domain. Railways are safety-critical systems. Their malfunction could lead to death or serious injury to people, loss or severe damage to equipment, or environmental harm. This work, carried out in cooperation with our industrial partner Invensys Rail, aims to develop an integrated view of rail networks within which capacity can be investigated and enhanced without compromising safety.

The process algebra CSP [11,18] has successfully been applied to modelling, analysing and verifying railways for safety aspects, see e.g. [20,21]. Solely concerned with safety, these approaches have ignored any aspect of time. However, the capacity of a rail network node is highly dependent on time: moving a point or moving a train through a node takes time, and sighting and braking distance are both functions of time. Thus, rather than using CSP, we apply Timed CSP [19,17] in order to achieve an integrated view on safety and capacity. While, e.g., [20,21] model safety within CSP, to the best of our knowledge we are the first to consider railway capacity within Timed CSP or any other similar formalism. One of the benefits of using (Timed) CSP is its naturalness; it takes little

⋆ This work was supported by RSSB/EPSRC under the grant EP/I010807/1.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 54–68, 2012.

effort to explain our formal models to our industrial partners who have assisted us throughout the process in ensuring that our models remain faithful to their engineering designs.

Of the various capacity notions within the railway domain, we deal here with so-called theoretical line capacity. "Theoretical capacity" denotes the capacity (i.e., throughput) that in principal can be scheduled (as opposed to the capacity actually used). By "line capacity" we refer to a situation in which all trains are of the same characteristic (e.g., all trains have the same braking behaviour and the same maximal speed) and take the same path through a network. It remains future work to capture the more complex notion of "network capacity" (the number of trains that can operate on a rail network in a given time period).

The literature on railway capacity classifies the various approaches into analytical, optimisation and simulation methods. Analytical methods, e.g., [3,7], model the railway infrastructure by means of mathematical expressions where a preliminary solution can be easily determined. This measures theoretical capacity and helps to identify bottlenecks. In contrast to this, optimisation methods, e.g., [4,16], utilise theoretical capacity by providing optimal time tables. Finally, simulation methods, see [6] for a survey, imitate the operation of the real world railway network over time. They present the dynamic behaviour of the network as moving from state to state with respect to well-defined rules. Our approach is closest to simulation methods. We differ from them as we take all possible system runs into account and therefore obtain a more concise result concerning capacity. Taking into account the whole behaviour of the system allows us also to consider safety. Overall, this leads to an integrated method.

Concerning safety, we build on the work of [21]. In general, other approaches outside of the Csp world, e.g., [10,13] verify the safety of interlocking programs with logical approaches and SAT-based model checking as the underlying proof technique.

Our paper is organized as follows. In Section 2, we discuss basic railway concepts in terms of a realistic double junction example provided as a real-world challenge by our industrial partner Invensys Rail, and use this example to motivate the question of capacity. In Section 3, we review the approach advocated by Winter [21] to modelling safety in the railway domain using Csp. In Section 4, the language Timed Csp and the idea of timed traces is introduced. In Section 5, we describe how to extend Winter's approach in order to capture the timing of events on a railway. Given such a timed behaviour, we ask in Section 6 what capacity it has by defining capacity as a function on timed traces which we then encode as a Timed Csp refinement. In Section 7 we apply these results to our original example, before concluding with an outline of future work in Section 8.

This paper is a significantly improved variant of [12], which was presented in an informal workshop setting without proper proceedings. We would like to thank Simon Chadwick and Dominic Taylor from Invensys Rail for their encouraging support and continuous invaluable feedback.

2 Railway Terminology and the Double Junction Example

We explain typical railway concepts in terms of the track plan shown in Figure 1. Engineers from Invensys Rail proposed this plan of a realistic *double junction* for our study as it exhibits typical challenges for safety and capacity. Their plan consisted of the elements in normal font; we added the components in boldface in order to facilitated the analysis and verification of railway protocols working over this junction. The double junction connects a *side line* with a *main line*. Concerning safety, its challenges include avoiding train collisions and preventing derailments. Concerning capacity, one is interested in optimising single paths through the junction as well as in reducing the time that one path blocks another.

The track plan depicted in Figure 1 consists of various elements. There are a number of individual *tracks*, which in the plan are named with two characters, e.g., AA; there are two *points*, namely point 101 and point 102; and finally there is the diamond crossing BW. A point may be in one of two positions: *normal* or *reverse*. If point 101 is in *normal position*, a train can pass from track AB to track AD; if it is in *reverse position*, a train can pass from track AB to track BW. The diamond crossing BW can be passed in two ways: it connects the tracks BV and BX, and it also connects the tracks AC and CM. The double junction is designed in such a way that trains can travel through it along four paths. There are two paths on the *main line*, paths $\overrightarrow{AB}$ and $\overrightarrow{DC}$. Path $\overrightarrow{AE}$ leaves the main line and enters the *side line* on track CM. Finally, path $\overrightarrow{FC}$ leaves the side line after track DR and enters the main line on track BY. On the main line, trains can travel at a speed of 120mph, whereas on the side line (i.e., on tracks CM, CL, DR, and DP) there is a speed restriction of 70mph. There is a further speed restriction of 40mph on the points 101 and 102 when they are in reverse position. These speed restrictions, which are as provided as a realistic scenario by Invensys Rail, are not shown on the track plan. There are six signals in the original plan, labelled 2, 3, 4, 5, 16 and 17, which show either *proceed* or *stop*. Train drivers are only allowed to enter a track, say AB, when its protecting signal, in our example signal 3, shows proceed; otherwise, the train driver has to stop and to wait until the signal changes to proceed.

Railway signals are controlled by an interlocking system which aims to guarantee safety. In general, train movements are considered safe if there is no risk of collision (through allowing two trains on the same track at the same time) nor any risk of derailment (through allowing a point to move while there is a train passing over the point, or by allowing a train to pass too fast over a point). In our study, we concentrate on capacity and consider only the collision-freedom aspect of safety.

An interlocking system gathers inputs from the physical railway, such as train locations with respect to tracks, and sends out commands to control signal aspects and point positions. To this end, it implements so-called *control tables* which dictate its behaviour. The control table for our double junction example appears in the lower left corner of Figure 1 and restricts the behaviour of the railway according to current UK regulations. For each signal, there is one row

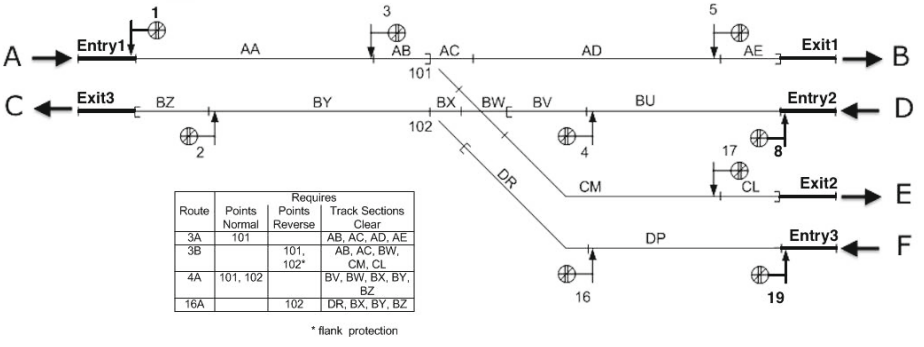


Fig. 1. The track plan of the *double junction*

describing the condition under which the signal can show proceed. There are two rows for signal 3: one for when the train stays on the main line (Route 3A) and one for when the train moves to the side line (Route 3B). Signal 3 for the main line can only show proceed when point 101 is in normal position and tracks AB, AC, AD and AE all are clear. The track AE is called an *overlap*. This rule ensures that the driver will always be able to stop the train before entering the track following AE, even if signal 5 is seen too late (e.g., just as the train is passing it). Besides the condition shown in the control table, signal 3 for the side line can only show proceed if an approaching train on track AA is slow enough. This is controlled by measuring the time that the train occupies track AA (approach control). This forces the train to slow down to 40mph even before it reaches signal 3. We refer to this version of control as Scenario 1.

Scenario 2 makes the assumption that all trains are equipped with an Automatic Train Protection (ATP) system [14]. ATP ensures that trains brake when needed and reduce their speed as required. Thanks to ATP, trains are guaranteed to stop at or before signal 5. Therefore the overlap AE can be removed from the clear part of the control table of signal 3. ATP guarantees that trains slow down in time to 40mph when passing a point. Thus, approach control is not needed. For Scenario 2, we remove all overlaps from the control table and work without approach control. Under current UK regulations, Scenario 2 is not allowed.

In the railway domain, capacity is regarded as an elusive concept which is not easy to define and measure. In general, it can be described as below:

> "Capacity determines the maximum number of trains that would be able to operate on a given railway infrastructure, during a specific time interval, given the operational conditions." [5].

Returning to our double junction example, the general view in the railway industry – shared by our industrial partner Invensys Rail – is the following. Removing overlaps such as track AE from control tables and removing approach control increases capacity. Our scientific questions are: can safety still be guaranteed? And how can the expected effect be measured? Based on the answers to these

questions, the political question would be: does the resulting capacity increase justify changes to regulations?

3 Modelling Railways for Safety in CSP

The process algebra Csp [11,18] is an established formalism for describing concurrent systems. While there is still ongoing research on foundations, Csp has many applications, e.g., in train controllers, in avionics, and in security protocols. We describe here only the basic constructs of Csp that we shall exploit.

Csp describes reactive systems in terms of abstract, discrete events such as "train 12 enters track 1". The events of a system are collected together in an *alphabet of communications* Σ. All such communications are atomic. In Csp terminology, a reactive system is referred to as a *process*. The most basic process is *Stop*, which represents the system that does not do anything. Another basic process is *Skip*, which represents the system that performs the termination event ✓ (pronounced as tick) and then behaves like *Stop*. Given an event a and a process P, the Csp process $(a \rightarrow P)$ represents the system that engages in the event a and then behaves like P. Csp provides two operators which allow a choice between processes: the process $(P \sqcap Q)$ is the *internal* choice operation which represents a system which will behave as either P or Q with the choice made nondeterministically by the system; while the process $(P \,\Box\, Q)$ is the *external* choice operation in which the choice between behaving as either P or Q is made by the environment. Csp provides various operators to combine two processes P and Q in parallel, but the only such operator of interest to us is $(P \,|[\, X \,]|\, Q)$ which requires the processes P and Q to cooperate on the events in the set X. Finally, the process $(P \setminus X)$ behaves like P but makes the events of X invisible to the environment. Semantically, Csp describes a process P by the set of all its traces $\mathcal{T}[\![P]\!]$, i.e., all finite sequences of events that the process can perform. A process *SPEC* is refined by a process *IMP*, written as $SPEC \sqsubseteq_T IMP$, iff $\mathcal{T}[\![IMP]\!] \subseteq \mathcal{T}[\![SPEC]\!]$. This refinement preserves safety: if a forbidden sequence of events s is excluded from *SPEC*, then s cannot be a trace of *IMP* if $SPEC \sqsubseteq_T IMP$. In practice, one usually works with Csp_M, a machine-readable version of Csp which also includes concepts of functional programming which handle data.

Winter [21] describes how to model railway systems in Csp for proving safety. We illustrate Winter's approach by modelling the double junction shown in Figure 1. Here we include the bold elements: without signal 1 trains could collide on track AA. The first step is to formalise the track plan as a graph:

```
datatype TrackIDs  = AA | AB | AC | AD | AE | ...
datatype SignalIDs = S2 | S3A | S3B | S4 | S5 | S16 | S17 | ...
datatype PointIDs  = P101 | P102
next(AA) = {AB}  next(AB) = {AC}  next(AC) = {AD,BW} ...
```

Then, trains are modelled as processes. Here, it is necessary to change Winter's definitions, as tracks can be shorter than trains (track AE is 50m long, Invensys suggested to work with a train length of 200m). We associate every track t (and

train *id*) with its length, denoted as *tracklength*(t) (and *trainlength*(*id*)) and define a process, namely *RearBehaves*, which refrains rear moves if the front track is shorter than the train length. Then, a train process is characterized by its identifier *id*, by the position of its *front* and by a list of *rearmoves* as follows:

```
TrainBehave(id,front,rearmoves) =
 if (front==Exit and null(rearmoves)) then TrainBehave(id,entry(id),<>)
 else ([] n1 : next(front) @ moveff.front.n1 ->
  RearBehaves(trackLength(n1),id,n1,
              rearmoves^<(moverr.front.n1,trainLength(id))>))
```

Next, control tables are modelled. We give here only the basic idea as presented in [21]. The signalling in the double junction requires a slightly more involved approach. When the front of a train enters the protected area, the signal state becomes *Red* indicating "halt". Similarly, when the rear of a train leaves the protected area, the signal state becomes *Green* indicating "proceed".

```
SignalBehave(id.aspect) =
   (aspect == Green & [] n : next(signalhome(id)) @
       moveff.signalhome(id).n -> SignalBehave(id.Red))
[] (aspect == Red & [] n : next(signalend(id)) @
      moverr.signalend(id).n -> SignalBehave(id.Green))
```

Finally, the whole train system comprises trains and signals which interact through a set of synchronized events:

```
TrainSystem = Trains [| union(
      Union({{ moveff.signalhome(id).n |
               n<- next(signalhome(id)) } | id <- SignalIDs }),
      Union({{ moverr.signalend(id).n |
               n<- next(signalend(id)) } | id <- SignalIDs }))
    |] Signals
```

We formalise the property `NoCollision` following Winter [21]. The difference is that we exclude the entry and exit tracks from the safety analysis:

```
P(F,R) =
([] on:union(F,R) @
(not(member(next(on),union(F,R))) or member(next(on),Exits) &
 (moveff.on.next(on) ->
   P(union(diff(F,{on}),union({next(on)},Entries)),R)))
  []
  (moverr.on.next(on) ->
  P(F,union(F,union(diff(R,{on}),union({next(on)},Entries))))))
SafeMove = P(Entries,Entries)
NoCollision = SafeMove ||| CHAOS(diff(Events,{|moveff,moverr|}))
```

A railway is safe iff it can perform only safe moves. This can equivalently be formulated in Csp as the refinement statement over the traces of the respective processes: $NoCollision \sqsubseteq_T TrainSystem$.

4 Timed CSP and Timed Traces

Timed Csp [19] conservatively extends the process algebra Csp with timing primitives, modelling the passage of time with reference to a single, conceptually global clock. Syntactically, the core extension of Csp to Timed Csp is modest. There are only three new operators, including timeout after d time units: $(P \triangleright^d Q)$. Based on these, Timed Csp adds many operators as syntactic sugar. Most prominent are $(Wait\ d) = (Stop \triangleright^d Skip)$ – the process, which waits for d time units before it terminates – and a delayed event prefix $(a \xrightarrow{d} P) = (a \rightarrow (Stop \triangleright^d P))$ which performs a and then behaves as P after a delay of d time units.

Semantically, processes in Timed Csp perform *timed events* $(r, e) \in \mathbb{R}_{\geq 0} \times \Sigma$: r is the time at which event e occurs. Events are instantaneous, i.e., they do not take any time. The execution of a system leads to a *timed trace*. We write $\langle\rangle$ for the empty trace and $t = \langle(r_1, e_1), \ldots, (r_n, e_n)\rangle$ for a finite observation with $\forall\, j > i \geq 1 : r_i \leq r_j$ and $\forall\, n > i \geq 1 : e_i \neq \checkmark$. Given a non-empty timed trace t, the time stamp of its first visible event is given by $begintime(t) = r_1$; that of its last visible event is given by $endtime(t) = r_n$; and its duration is given by $duration(t) = endtime(t) - begintime(t)$. We define $duration(\langle\rangle) = 0$. $\#t$ denotes the number of timed events occurring within a timed trace t. Given a set of events A, $t \upharpoonright A$ denotes the projection of t onto A, i.e., the subsequence of timed events from t which consists only of events from A. Using these notations, $t \downarrow A = \#(t \upharpoonright A)$ is the number of timed events from A in t. Given two timed traces t_1 and t_2, $t_1 \frown t_2$ denotes their concatenation; if the time stamps do not match, this concatenation is undefined.

We denote the set of all timed traces by TT, and write $\mathcal{T}_{\mathbb{R}}[\![P]\!] \subseteq TT$ for the set of all timed traces of a Timed-CSP process P. Given two Timed Csp processes IMP and $SPEC$, we say that $SPEC$ refines to IMP, denoted by $SPEC \sqsubseteq_{TT} IMP$, iff $\mathcal{T}_{\mathbb{R}}[\![SPEC]\!] \supseteq \mathcal{T}_{\mathbb{R}}[\![IMP]\!]$.

For the case that $SPEC$ is independent of time, i.e., $SPEC$ does not include any timed operator, and IMP is an integer-wait Timed Csp process, Roscoe [18] provides the following proof technique: $SPEC \sqsubseteq_{TT} IMP$ over Timed Csp is equivalent to $time(SPEC) \sqsubseteq_{T} time(IMP)$ over (untimed) Csp. The latter proof obligation can be discharged using standard CSP tools such as the model checker FDR [1]. The function $time$ adds a special event called $tock$ to the alphabet of the processes in order to indicate the passage of time, e.g., $time(\,Wait\ 0) = SKIP$ and $time(\,Wait\ (n{+}1)) = tock \rightarrow time(\,Wait\ n)$. For the external choice operator, we use Schneider's construction [19]: $time((?x : A \rightarrow P) \,\Box\, (?y : B \rightarrow Q)) = time(?x : A \rightarrow P) \,\Box\, time(?y : B \rightarrow Q) \,\Box\, tock \rightarrow time((?x : A \rightarrow P) \,\Box\, (?y : B \rightarrow Q))$ which provides a correct translation if $A \cap B = \emptyset$; this can be seen, e.g., by comparison with the semantics of $\Box_{tock}$ (see [19]). The automatic verification of real-time systems with FDR is competitive with other approaches [15].

We use the timed refusal trace semantics of Timed Csp as defined in [17], which guarantees a semantics for recursion even if the processes involved fail to be timed-guarded (see [19]). The timed traces of a process P can be extracted straightforwardly from the timed refusal traces of P [17].

5 Modelling Timed Behaviours of Railway System

In the following, we make two assumptions concerning time in railways. Firstly, we assume signalling to be instantaneous. In the real world, the cycle time of an interlocking is in the region of two seconds. This time is (nearly) negligible compared to the time a train needs to move from one track to another, so for the current study we disregard this delay rather than require the interlocking to await confirmation that the signal has been changed. Slightly more critical is our second assumption; for this study we assume that trains accelerate and brake immediately. The consequence is that we overestimate capacity (as trains move faster than in reality). The second assumption is clearly an over simplification to be remedied in future work.

In order to model time, we enrich Winter's model [21] of a track plan. To this end, we record the time it takes to travel a distance l at a speed limit s in a table $delay(l, s)$. The second change to the untimed model is that trains get one more parameter: besides their identity *id* and the position *front*, *rearmoves*, they also have the speed allowed on track *front*. The following Timed Csp code summarizes the essential part of these changes; all other processes remain as described in Section 3.

```
TrainBehave(id,front,rearmoves,curspeed) =
  if (front==Exit and null(rearmoves))
  then TrainBehave(id,entry(id),<>,0)
  else ([] n1 : next(front) @
    moveff.front.n1?speed ->
    if (tracklength(n1)>trainlength(id))
      then Wait(delay(trainlength(id),speed))
      else Wait(delay(tracklength(n1),speed));
    RearBehaves(tracklength(n1),id,n1,
                rearmoves^<(moverr.front.n1,trainlength(id))>),speed)
```

6 Modelling Railway Capacity

We now develop a semantic concept of capacity based on timed traces, and characterise a railway's capacity via time-wise refinement in Timed Csp.

6.1 Capacity Semantically

In this section, we present a formal definition of railway capacity which is compliant with the quotation given in Section 2 and compatible with existing analytical methods, for example [3]. Informally speaking, we want to count the number of trains appearing and operating within the railway. This number depends on two parameters: namely, (i) when we start counting and (ii) how long we observe. Thus, we speak of an *observation window* characterised by a starting time and a duration. There are two kinds of trains that we can observe in such a window: those trains which are already present at the starting time of the window, and those trains, which appear in the window while it is open.

Initially, we assume that there are no trains in the railway. As trains enter, travel through and leave a railway, their movements are recorded in a timed trace. Relative to a given track plan, we define *Entering* and *Leaving* as the sets of timed events which indicate the entering and leaving of trains, respectively. In our example, $moveff.Entry1.AA$ is an element of *Entering*, the set *Leaving* includes, e.g., the element $moverr.AE.Exit$.

Let s be a timed trace of a railway model. The number of trains in the railway after s is given by the number of trains entering the railway reduced by the number of trains leaving the railway:

$$storage(s) \;=\; s \downarrow Entering - s \downarrow Leaving$$

The number of trains entering the railway during s is given by

$$increase(s) \;=\; s \downarrow Entering$$

Relative to the duration δ of an observation window, we define the capacity of a Train System TS by

$$capacity(TS, \delta) \;=\; \max \left\{ \begin{array}{l} storage(s_1) + increase(s_2) \;: \\ s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TS]\!] \text{ and } duration(s_2) \leq \delta \end{array} \right\}.$$

Each decomposition $s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TS]\!]$ of a timed trace yields a value to be considered for capacity. We determine how many trains are in the system after the set-up phase s_1 and how many trains enter the system during the observation window s_2, and maximise the sum $storage(s_1) + increase(s_2)$ over all timed traces $s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TS]\!]$ in which $duration(s_2) \leq \delta$. The following result shows that this definition of capacity nicely fits with refinement:

Theorem 1 (Capacity and Refinement). *If* $TS_2 \sqsubseteq_{TT} TS_1$ *then* $\forall\, \delta \geq 0 :$ $capacity(TS_1, \delta) \leq capacity(TS_1, \delta)$.

For our purposes the decomposition of a timed trace into a set-up phase and an observation window gives a good insight into a railway system (see especially the paragraph on simulation later in Section 7). However, we note in passing that a notion of capacity for a Train System TS which is independent of observation duration, giving a long-term rate of "trains per unit time", could be defined by

$$\lim_{\delta \to \infty} \frac{capacity(TS, \delta)}{\delta}.$$

6.2 Capturing Storage and Increase in Timed Csp

In this section, we provide a construction in Timed Csp which turns capacity into an observable event. To this end, we run the process *TrainSystem* in a two-layered environment. The first layer consists of an observer process, while the second layer controls the whole set-up. The observer process synchronises

with *TrainSystem* over events indicating the entering and leaving of trains with respect to the railway. The controller process synchronises with the observer process on the duration of the observation window and the observed capacity. The observer process works in two phases: the process *Storage* (see below) realises the function *storage* (see above); after a *startObs* signal from the control layer, control goes over to the second phase in which the process *Increase* (see below) realises the function *increase* (see above).

The process *Storage* counts the entering trains and reduces this number by one for every leaving train:

```
Storage(n) = ([] n1 : next(Entry) @ moveff.Entry.n1?_  -> Storage(n+1))
          [] ([] n1 : pre(Exit)   @ moverr.n1.Exit  -> Storage(n-1))
          [] startObs?delta -> Increase(n,0,delta)
```

In addition, the process listens on the channel *startObs*. When it receives a value *delta*, it passes control to the process $Increase(n, 0, delta)$. Here, n is the number of trains which are on the railway already, 0 is the duration since the observation started, and *delta* is the size of the observation windows.

The process *Increase* counts the entering trains as long as the observation window is open. When the window is closed, it informs the controller on the channel *infocap* on the observed capacity. After this, it goes to an idle state *EndCapObserver*. The process *Increase* is defined as follows:

```
Increase(n,d,delta) =
d<=delta & ([] n1 : next(Entry) @  moveff.Entry.n1?_ @ u ->
                if d+u<=delta then Increase(n+1,d+u,delta)
                              else Infocap(n))
         [] ([] n1 : pre(Exit) @  moverr.n1.Exit @ u ->
                if d+u<=delta then Increase(n,d+u,delta)
                              else Infocap(n))
Infocap(n) =  infocap.n -> EndCapObserver
         [] ([] n1 : next(Entry) @  moveff.Entry.n1?_ -> Infocap(n))
         [] ([] n1 : pre(Exit) @  moverr.n1.Exit ->  Infocap(n))
EndCapObserver = ([] n1:next(Entry)@moveff.Entry.n1?_ -> EndCapObserver)
              [] ([] n1:pre(Exit)@moverr.n1.Exit ->  EndCapObserver)
```

The process *Controller* decides when the observation window starts, and later receives the value of the observed capacity through the channel *infocap*. This process is defined as follows:

```
Controller(delta) = startObs.delta -> infocap?n -> Stop
```

The overall set-up is given by the process *TrainSystemWithCapacity*:

```
TrainSystemWithCapacity =
    (TrainSystem
     [|union(
       { moveff.Entry.n._ | n <- next(Entry) },
       { moverr.n.Exit | n <- pre(Exit) })|]
     Storage(0)) [| {|startObs, infocap|} |] Controller(delta)
```

We define that a process Q does not block a process P with alphabet Σ_P over a synchronization set X if $(\mathcal{T}_{\mathbb{R}}[\![P]\!] = \mathcal{T}_{\mathbb{R}}[\![P \,|[\, X \,]|\, Q]\!]) \upharpoonright \Sigma_P$. We establish the following result for the coupling of *TrainSystem* and *Storage* in the definition of the process *TrainSystemWithCapacity*:

Theorem 2. *Storage*(0) *does not block TrainSystem.*

Proof (Sketch). In *Storage*(0), every event in $\{moveff.Entry.n.\_ \mid n \in next(Entry)\} \cup \{moverr.n.Exit \mid n \in pre(Exit)\}$ is always ready to engage.

This insight provides the following result.

Theorem 3. *The following are equivalent:*

- $capacity(TrainSystem, \delta) = n$.
- $n' \leq n$ *iff there exists a timed trace* $t \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ *such that* $(r, infocap.n') \in t$ *for some* $r \in \mathbb{R}$.

Proof (Sketch). The following two correspondences hold between the timed traces of the process *TrainSystem* and the process *TrainSystemWithCapacity*:

- If $t_1 = s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$ then $t_1' = s_1 \frown \langle (begintime(s_2), startObs.\delta) \rangle \frown s_2 \frown \langle (endtime(s_2), infocap.n) \rangle \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$.
- If $t_2 = s_1 \frown \langle (r_1, startObs.\delta) \rangle \frown s_2 \frown s_3 \frown \langle (r, infocap.n) \rangle \in \mathcal{T}_{\mathbb{R}}[\![TrainSystemWithCapacity]\!]$ then $t_2' = s_1 \frown s_2 \in \mathcal{T}_{\mathbb{R}}[\![TrainSystem]\!]$.

6.3 Capacity via Refinement

We formulate a process which allows at most n trains operating within an observation window of duration *delta*. Here, we use only events of the interface between the observer process and the controller process:

```
CapacityFrom(n,delta)=|~|n':{0..n}@ startObs.delta -> infocap.n' -> Stop
```

With regards to this process, we have the following result:

Theorem 4. *Given a length* δ *of observation,* $capacity(TrainSystem, \delta) = n$ *iff*

- *for all* $k \geq n$ *holds:* $CapacityFrom(k, \delta) \sqsubseteq_{TT} TrainSystemWithCapacity \setminus MoveEvents$, *and*
- *for all* $0 \leq l < n$ *holds:* $CapacityFrom(l, \delta) \not\sqsubseteq_{TT} TrainSystemWithCapacity \setminus MoveEvents$,

where $MoveEvents = \{moveff.x.y.\_, moverr.x.y \mid x, y \in Tracks\}$

Proof (Sketch). By Theorem 3 and the definition of $CapacityFrom(n, \delta)$.

7 Studying Safety and Capacity in the Context of the Double Junction

In this section, we study *safety and capacity in one model* formulated in Timed Csp. To this end, we consider each of the paths $\overrightarrow{AB}$, $\overrightarrow{DC}$, $\overrightarrow{AE}$ and $\overrightarrow{FC}$ shown in Figure 1 in isolation. It remains future work to study the double junction as a whole. For each of the four paths, we encode Scenarios 1 and 2 from Section 2. It turns out that both scenarios are safe and that capacity increases when signalling is changed from Scenario 1 to Scenario 2.

For the model of each path, we determine the minimal amounts of time a train travels from one end to the other of each track. Here, we use data suggested by our industrial partner Invensys Rail about the lengths of trains and tracks. We take the length of trains to be 200m, the length of tracks where there is either a point or a diamond crossing to be 50m, the length of overlap tracks to be 200m, the length of other tracks to be 1500m, and the track lengths on path $\overrightarrow{AE}$ to be summarised in the following table:

Track	AA	AB	AC	BW	CM	CL
Length	1500m	200m	50m	50m	1500m	200m

The minimal amounts of time to travel such distances in different speed limits can be easily calculated. For example, it takes at least $4s$ for a train to travel on AA at a speed of 120mph. These constants are incorporated into the Timed Csp models as presented in Section 5. These result in processes $\mathit{TrainSystem}_{p,s}$, where p ranges over $\{\overrightarrow{AB}, \overrightarrow{AE}, \overrightarrow{DC}, \overrightarrow{FC}\}$ and $s \in \{1, 2\}$.

These models are collision-free iff $\mathit{NoCollision} \sqsubseteq_{TT} \mathit{TrainSystem}_{p,s}$. Since the processes $\mathit{TrainSystem}_{p,s}$ contain only integer-wait operators and $\mathit{NoCollision}$ does not include any timed operator, we utilise FDR to prove the refinements $\mathit{time}(\mathit{NoCollision}) \sqsubseteq_{T} \mathit{time}(\mathit{TrainSystem}_{p,s})$. FDR shows that all these refinements hold. Thus, all paths are safe in both scenarios.

In order to deal with capacity, we simulate both scenarios with our Timed Csp Simulator tool [8]. This is possible as the processes $\mathit{TrainSystem}_{p,s}$ involve only rational numbers for time. To this end, we apply the *automatic simulation* available in the Timed Csp Simulator, which randomly chooses between events and prioritises events over the evolution of time. Simulating $\mathit{TrainSystem}_{p,s}$ in the Timed Csp Simulator yields one of its timed traces.

We determine capacity in a three step process. First, we make estimates on the length of the set-up phase and on the minimal length δ of an observation window. We choose these numbers in such a way that we can expect a difference in the capacities of Scenarios 1 and 2. Next, we validate this estimation. Both these steps are based on simulation with the Timed Csp Simulator. Finally, given a good estimate for the length δ, Theorem 4 allows us to determine $\mathit{capacity}(\mathit{TrainSystem}_{p,s}, \delta)$ for each $\mathit{TrainSystem}_{p,s}$. Here, we discharge the involved proof obligations with FDR. This is possible as the process $\mathit{CapacityFrom}(n)$ does not have any timed operator and there are only integer-waits in the process $\mathit{TrainSystemWithCapacity}$.

Step 1: Simulation with the Timed Csp Simulator suggests that it takes a fixed time μ from one train entering $TrainSystem_{p,s}$ until this train leaves. Furthermore, it suggests that it takes a fixed time d from one train entering $TrainSystem_{p,s}$ to the next train entering $TrainSystem_{p,s}$. Let μ_1, d_1 and μ_2, d_2 be the estimates from the simulation of Scenarios 1 and 2, respectively. For the length of the observation window we select $\delta = d_1 x$ for some x such that $d_1 x = d_2(x+1)$. For total run-times we choose $\mu_1 + \delta$ ($\mu_2 + \delta$) for Scenario 1 (Scenario 2). This "guarantees" (based on the simulation data) that the two scenarios show different capacities. For the path $\overrightarrow{AE}$, we obtain $\mu_1 = 113s$, $\mu_2 = 105s$, $d_1 = 85s$ and $d_2 = 70s$. Thus, we choose $\delta = 397s$ and simulate Scenario 1 for $\mu_1 + \delta = 510s$ and Scenario 2 for $\mu_2 + \delta = 502s$.

Step 2: Automatic simulation of Scenario 1 for $\mu_1 + \delta$ and of Scenario 2 for $\mu_2 + \delta$ yields two timed traces. The capacity observed on these timed traces gives lower bounds for the capacity of $TrainSystem_{p,s}$. For path $\overrightarrow{AE}$, we obtain a capacity of 7 in Scenario 1 and a capacity of 8 in Scenario 2.

Step 3: Finally, we verify with FDR that these numbers are indeed the capacity. For path $\overrightarrow{AE}$, we obtain:

$\overrightarrow{AE}$	6	7	8	Running time in FDR
Scenario 1	x $(1, 12 \times 10^6)$	✓ $(1, 14 \times 10^6)$	-	25s
Scenario 2	-	x $(1, 67 \times 10^6)$	✓ $(1, 73 \times 10^6)$	33s

Each row in the table provides the result for one scenario. Column n expresses if the refinement $CapacityFrom(n) \sqsubseteq_{TT} TrainSystemWithCapacity \setminus MoveEvents$ holds. "✓" stands for successful verification, "x" indicates that the refinement does not hold, "-" says that this check was not carried out. We associate the round number of states that are checked by FDR in each refinement. The last column shows how long FDR spends for running all checks needed for determining capacity in each scenario[1]. The results for the other paths are obtained in the same way as for $\overrightarrow{AE}$. We summarise these in the following table:

Path	Window length	Capacity in Scenario 1	Capacity in Scenario 2
$\overrightarrow{AB}$	379s	12	13
$\overrightarrow{DC}$	399s	12	13
$\overrightarrow{FC}$	328s	7	8

Interpreting our results within the railway domain, we can state: under optimal conditions, we expect one more train approximately every 6.5 minutes in Scenario 2 compared with Scenario 1 without compromising safety. It takes about 2 minutes of set-up time to observe this difference.

For windows of length larger than 6.5 minutes, $TrainSystem_{p,2}$ has at least the capacity of $TrainSystem_{p,1}$. This holds by Theorem 1. We observe that $TrainSystem_{p,1}$ and $TrainSystem_{p,2}$ are identical but for the value x in the

[1] On a machine with a 2GHz 64 bit processor with 4GByte memory running Mac OS.

Wait x processes involved. Here, $x' \leq x$ for the corresponding *Wait* processes, where x is the value in $TrainSystem_{p,1}$ and x' is the value in $TrainSystem_{p,2}$. Thus, any delay y between two timed events of a timed trace of $TrainSystem_{p,1}$ can be reproduced in $TrainSystem_{p,2}$ as for the corresponding *Wait* x', we have $x' \leq x \leq y$.

Lessons learnt from using tools for Csp and Timed Csp for our performance analysis in the railway domain include:

1. Obviously, one would like to study safety for the whole junction and not, as we do above, for single paths in isolation. However, it turns out that handling the complete junction (even in the untimed case) is beyond the proof support given by FDR, at least for our current modelling approach.
2. Reducing proof obligations over Timed Csp to proof obligations over (untimed) Csp works well. Tool support for the translation (which we carry out manually) would be welcome.
3. Concerning capacity, it might be possible to determine it by "optimal" simulation in the Timed Csp Simulator. To this end, one would have to argue which simulation strategy leads to a timed trace showing the capacity of the railway system.
4. Proper time-wise refinement, as in $TrainSystem_{p,2} \sqsubseteq_{TT} TrainSystem_{p,1}$, still lacks convincing tool support. On our examples, the PAT system [2] was running out of memory for the refinement checks.

8 Summary and Future Work

In close cooperation with railway industry, we have provided a formal definition of line capacity based on the timed traces that one can observe in a natural, timed model of railway systems. This definition can equivalently be characterized as a refinement statement in Timed Csp. By adapting the safety formulation of Winter [21], we are able to study both safety and capacity in one formal model in Timed Csp. As the refinements for safety and capacity only require the checking of qualitative properties, both refinement statements can be discharged by translation into untimed Csp. This approach has the advantage that one can re-use established tool support for Csp alone.

To illustrate our approach, we have applied it to a standard double junction from the (UK) railway domain. For this junction, we can answer fundamental questions from railway industry: changing control tables in the way suggested by railway engineers yields a capacity increase without compromising safety. This increase can be quantified: under optimal conditions, after the change there can be one more train every six minutes in our example. Having shown the increases that can be gained via changing signalling rules through the trusted use of ATP, this encourages changes to be proposed to the current UK railway regulation.

The double junction example demonstrates the limitations of the current proof support in terms of the model checker FDR. For complex examples, e.g., with more tracks, 3-aspect or 4-aspect signalling, long or different-length delays, the translational approach is inefficient. Dedicated proof support, e.g., in the form of a Timed Csp-Prover (currently under construction) is necessary.

It remains future work to include further timing aspects into our modelling, such as the cycle time of signalling and point movements or braking and accelerating curves of trains. Finally, we intend to develop our definition further, so that it also captures the more complex notion of network capacity.

Acknowledgement. The authors would like to thank Erwin R. Catesbeiana (Jr) for pointing out that immobility is the enemy of capacity.

References

1. FDR2, http://www.fsel.com/software.html
2. PAT, http://www.comp.nus.edu.sg/~pat/
3. UIC Leaflet 405 OR. Links between Railway Infrastructure Capacity and the Quality of Operations. International Union of Railways (1996)
4. UIC Leaflet 406. Capacity. International Union of Railways (2004)
5. Abril, M., Barber, F., Ingolotti, L., Salido, M., Tormos, P., Lova, A.: An assessment of railway capacity. Transportation Research Part E: Logistics and Transportation Review 44(5), 774–806 (2008)
6. Barber, F., Abril, M., Salido, M., Ingolotti, L., Tormos, P., Lova, A.: Survey of automated systems for railway management. Technical Report. TU Valencia (2007)
7. Burdett, R.L., Kozan, E.: Techniques for absolute capacity determination in railways. Transportation Research Part B: Methodological 40(8), 616–632 (2006)
8. Dragon, M., Gimblett, A., Roggenbach, M.: A Simulator for Timed CSP. In: AVoCS 2011. Technical Report. Newcastle University (2011)
9. Department of Transport. Delivering a Sustainable Railway. White Paper CM 7176 (2007)
10. Fokkink, W., Hollingshead, P.: Verification of interlockings: from control tables to ladder logic diagrams. In: Proceedings of FMICS 1998, pp. 171–185 (1998)
11. Hoare, T.: Communicating Sequential Processes. Prentice Hall (1985)
12. Isobe, Y., Nguyen, H.N., Roggenbach, M.: Towards safe capacity in the railway domain – an experiment in Timed-CSP. In: DSW 2011 (2011)
13. James, P., Roggenbach, M.: Automatically Verifying Railway Interlockings using SAT-based Model Checking. In: AVoCS 2010. EASST (2011)
14. Kerr, D., Rowbotham, T.: Introduction To Railway Signalling. Institution of Railway Signal Engineers (2001)
15. Khattri, M., Ouaknine, J., Roscoe, A.: Automated translation of timed automata to Tock-CSP. In: AVoCS 2010. Technical Report. Düsseldorf University (2010)
16. Landex, A., Kaas, A., Schittenhelm, B., Schneider-Tilli, J.: Practical use of the UIC 406 capacity leaflet by including timetable tools in the investigations. In: Proceedings of the 10th International Conference on Computers in Railways (2006)
17. Ouaknine, J., Worrell, J.: Timed CSP = closed timed ε-automata. Nordic Journal of Computing 10, 1–35 (2003)
18. Roscoe, B.: Understanding Concurrent Systems. Springer (2010)
19. Schneider, S.: Concurrent and Real-time systems. Wiley (2000)
20. Simpson, A., Woodcock, J., Davies, J.: The mechanical verification of solid-state interlocking geographic data. In: Formal Methods Pacific 1997. Springer (1997)
21. Winter, K.: Model checking railway interlocking systems. Australian Computer Science Communications 24(1) (2002)

Refinement-Based Development of Timed Systems

Jesper Berthing[1], Pontus Boström[2], Kaisa Sere[2], Leonidas Tsiopoulos[2], and Jüri Vain[3]

[1] Danfoss Power Electronics A/S, Denmark
jbe@danfoss.com
[2] Åbo Akademi University, Finland
{pontus.bostrom,kaisa.sere,leonidas.tsiopoulos}@abo.fi
[3] Tallinn University of Technology, Estonia
vain@ioc.ee

Abstract. Refinement-based development supported by Event-B has been extensively used in the domain of embedded and distributed systems design. For these domains timing analysis is of great importance. However, in its present form, Event-B does not have a built-in notion of time. The theory of refinement of timed transition systems has been studied, but a refinement-based design flow of these systems is weakly supported by industrial strength tools. In this paper, we focus on the refinement relation in the class of Uppaal Timed Automata and show how this relation is interrelated with the data refinement relation in Event-B. Using this interrelation we present a way how the Event-B and Uppaal tools can complement each other in a refinement-based design flow. The approach is demonstrated on a fragment of an industrial case study.

1 Introduction

The Correct-by-Construction Design (CCD) workflow has proven its importance with motivating facts from recent industrial practice. Peugeot Automobiles has developed the model of the functioning of subsystems (lightings, airbags, engine, etc) for Peugeot after sales service; RATP (Paris Transportation) used the model of automatic platform doors to equip an existing metro line to verify the consistency of System Specification. Event-B [1] as one such CCD supporting formalism has proven its relevance in data intensive development while lacking sufficient support for timing analysis and refinement of timed specifications. Uppaal Timed Automata (UPTA) [2] address timing aspects of systems providing efficient data structures and algorithms for their representation and analysis but are less focusing on supporting refinement-based development, especially data refinement. The goal of this paper is to advocate the model-based transformational design method where these two approaches are combined to mutually complement each other.

The transformational design flow discussed in the paper consists of alternation between data and timing refinement steps (see Fig. 1). The result of a data

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 69–83, 2012.

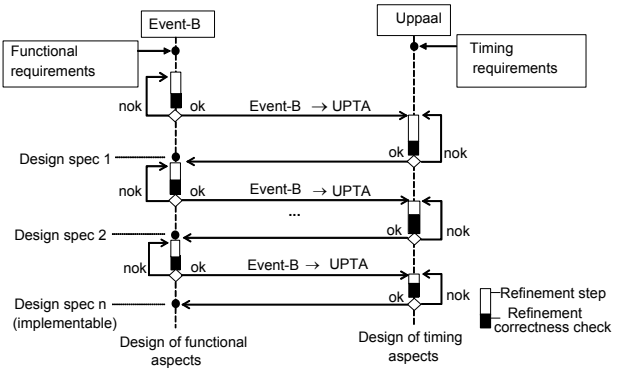


Fig. 1. CCD workflow with interleaving data and timing refinement steps

refinement step, performed within Event-B, after being proved correct, serves as an input to timing refinement step. The timing refinement means mapping the constraints of the previous level timing specification onto the UPTA model that is derived from the UPTA model of the previous design step and from the Event-B refinement of the current step. Then the newly introduced refinement of Event-B model must be decorated with timing attributes so that timing correctness criteria are satisfied. The timing correctness of refinement is verified using Uppaal [2] tool in two steps. At first, the consistency of the model being the result of timing refinement step is verified internally. Here the properties, e.g. deadlock and non-Zenoness are checked. Second, the preservation of timing properties introduced in the previous refinement step are verified. The design flow depicted in Fig. 1 is not complete since the real design flow may include also backtracking over several earlier design steps. For instance, when no feasible timing refinement is possible, it may require revision of much earlier functional refinement phases. The design backtracking issues and error diagnostics are not addressed also in the current paper.

2 Related Work

An extensive study of automata models for timed systems is presented in [12]. A general automaton model is defined as the context for developing a variety of simulation proof techniques for timed systems. These techniques include refinements, forward and backward simulations, hybrid forward-backward and backward-forward simulations, and other relations. Relationships between the different types of simulations are stated and proved. To improve model checking performance of timed systems, timing constraint refinement methods such as the efficient forward algorithm based on zones for checking reachability [3] and the counter example guided automatic timing refinement technique [8] have been studied. The refinement of timing has been addressed as part of specification

technique recently in [7] where the constructs for refinement of Timed I/O specifications were defined for development of compositional design methodology.

In works [3,7,8] the motivation behind timing refinement has been rather model checking or automated design verification than stepwise transformation-based design development. The way how timing refinement steps are constructed in the course of practical design flow has deserved relatively little attention. For systematic and modular co-use of refinement transformations both data and timing refinement transformations must be specified explicitly in terms of syntactic constraints, i.e., the domain of refinement transformations must consist of well-defined syntactic constructs of the modelling language.

In [4,11,13] attempts have been made to incorporate discrete time directly into formalisms without having a native notion of time. However, the clocks are not an integrated part, but are modelled as ordinary variables. Hence, continuous time specific problems such as Zeno behaviour cannot be addressed directly in, for example, the Event-B proof system. Furthermore, timing can be seen as an extra functional property and adding this to a functional Event-B model will make it cluttered with non-functional aspects. This will, apart from making the model less readable, make the proofs harder to automate.

An earlier attempt to integrate stepwise development in Event-B with model checking in Uppaal is given in [10] where the events are grouped into more coarse grained processes with timing properties. We aim to provide a framework where the timing refinements can be addressed by reusing the model constructs of Event-B introduced in the course of data refinement steps. That provides opportunity to verify data refinement steps also from the timing feasibility point of view.

3 Case-Study: Safety Related Controller Design

The CCD methodology introduced above will be deployed on an industrial redundant safety controller design case study by Danfoss A/S. The case study concerns an emergency shutdown module for a frequency converter that is used to control the speed of an electrical motor. An emergency switch and a safe field bus are used to activate the safety functions. The emergency shutdown module provides two safety functions, the Safe Torque Off (STO) and the Safe Stop 1 (SS1). In this paper we focus on the activation of the safety functions through the emergency switch only. The safety functions are activated if at least one of the two emergency switches (ES) is pushed. STO will remove the torque on the electrical motor. If SS1 is configured active, on activation of SS1 a timer with a configurable delay shall be started and the frequency converter is requested to start a non-safe ramp down. After the timer expires, STO shall be activated. In addition to these functional requirements we need to take into account timing requirements. Specifically, the reaction time from user terminal (pushing of ES) to active STO or active SS1 shall be less than 10 ms. STO or SS1 shall not be activated if the duration of the ES signal is shorter than or equal to 3 ms.

The deployment of our CCD methodology starts with a specification which captures an abstract description of the behaviour of the whole system. Stepwise refinements should introduce the algorithms needed for the implementable

system to behave according to the specification. The abstract specification and refinements should be done in such a manner that we can prove all (safety and liveness) properties stated in the requirements as invariant properties or refinements. Because of the space limit, in this paper we will present only one refinement step introducing the needed redundancy. The safety integrity requirements of this safety critical system require the safety controller to be mapped onto a redundant architecture (see 1oo2 architecture of IEC 61508-6 [9]). Before we proceed with the actual modelling of the emergency shutdown module we first present in the next section preliminaries of Event-B and UPTA as well as the mapping between the two formalisms.

4 Preliminaries

4.1 Preliminaries of Event-B

Consider an Event-B model M with variables v, invariant $I(v)$ and events $\mathcal{E}_1, \ldots, \mathcal{E}_m$. All events can be written in the form

$$\mathcal{E}_i = when\ G_i(v)\ then\ v : |S_i(v, v')\ end$$

where $G_i(v)$ is a predicate called the guard and $v : |S_i(v, v')$ is a statement that describes a nondeterministic relationship between variable valuations before and after executing the event. Event-B models do not have a fixed semantics [1], but correctness of a model is defined by a set of proof obligations. We can use these proof obligations to prove correctness of many transition systems. In order to guarantee that $\mathcal{E}_i$ preserves invariant $I(v)$ we need to show that [1]:

- $I(v) \wedge G_i(v) \wedge S_i(v, v') \Rightarrow I(v')$ (INV).

In order to be able to relate Event-B with UPTA in the following sections we interpret an Event-B model as a Labelled Transition System (LTS) $(\Sigma, init, T, i)$, where Σ is the set of states, $init$ is the set of initial states, $T \subseteq \Sigma \times \Sigma$ is the set of transitions and i is the set of legal states $i \subseteq \Sigma$. The set of states g_i where the guard of a transition σ_i holds is given as $g_i = \{v|G_i(v)\}$ and the relation s describing the before after relation for states corresponding to the update statement $v : |S_i(v, v')$ is given as $\{v \mapsto v'|S_i(v, v')\}$. The relation describing the before-after states for each transition T_i is then given as[1] $g_i \lhd s_i$. We can now describe the Event-B model as a transition system $(\Sigma, init, T, i)$ where the state space Σ is formed from the variables $v_1, \ldots, v_n$, $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$, Σ_i is the type of v_i. The initial states are formed as $init = \{v|Init(v)\}$. The transitions T_i are given as $T_i = g_1 \lhd s_1 \cup \ldots \cup g_m \lhd s_m$. The set of legal states are the ones where the invariant holds $i = \{v|I(v)\}$.

[1] The domain restriction operator $\lhd$ is defined as: $g \lhd s = \{\sigma \mapsto \sigma' \in s|\sigma \in g\}$.

4.2 Preliminaries of UPTA

An UPTA is given as the tuple $(L, E, V, Cl, Init, Inv, T_L)$, where L is a finite set of locations, E is the set of edges defined by $E \subseteq L \times G(Cl, V) \times Sync \times Act \times L$, where $G(Cl, V)$ is the set of constraints allowed in guards. $Sync$ is a set of synchronisation actions over channels. An action *send* over a channel h is denoted by $h!$ and its co-action *receive* is denoted by $h?$. Act is a set of sequences of assignment actions with integer and boolean expressions as well as with clock resets. V denotes the set of integer and boolean variables. Cl denotes the set of real-valued clocks ($Cl \cap V = \varnothing$). $Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks. $Inv : L \rightarrow I(Cl, V)$ is a function that assigns an invariant to each location, $I(Cl, V)$ is the set of invariants over clocks Cl and variables V. $\mathrm{T}_L : L \rightarrow \{$*ordinary, urgent, committed*$\}$ is the function that assigns the type to each location of the automaton.

We introduce the semantics of UPTA as defined in [2]. A clock valuation is a function $val_{cl} : Cl \rightarrow \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. A variable valuation is a function $val_v : V \rightarrow \mathbb{Z} \cup BOOL$ from the set of variables to integers and booleans. Let $\mathbb{R}^{Cl}$ and $(\mathbb{Z} \cup BOOL)^V$ be the sets of all clock and variable valuations, respectively. The semantics of an UPTA is defined as an LTS $(\Sigma, init, \rightarrow)$, where $\Sigma \subseteq L \times \mathbb{R}^{Cl} \times (\mathbb{Z} \cup BOOL)^V$ is the set of states, the initial state$init = Init(cl, v)$ for all $cl \in Cl$ and for all $v \in V$, with $cl = 0$, and $\rightarrow \subseteq \Sigma \times \{\mathbb{R}_{\geq 0} \cup Act\} \times \Sigma$ is the transition relation such that:

$$(l, val_{cl}, val_v) \xrightarrow{\mathtt{d}} (l, val_{cl} + \mathtt{d}, val_v) \text{ if } \forall \mathtt{d}' : 0 \leq \mathtt{d}' \leq \mathtt{d} \Rightarrow val_{cl} + \mathtt{d}' \models Inv(l), \text{ and}$$

$$(l, val_{cl}, val_v) \xrightarrow{act} (l', val'_{cl}, val'_v) \text{ if } \exists \mathtt{e} = (l, act, G(cl, v), r, l') \in E \text{ s.t.}$$
$$val_{cl}, val_v \models G(cl, v), val'_{cl} = [re \mapsto 0]val_{cl}, \text{and } val'_{cl}, val'_v \models Inv(l'),$$

where for delay $\mathtt{d} \in \mathbb{R}_{\geq 0}$, $val_{cl} + \mathtt{d}$ maps each clock cl in Cl to the value $val_{cl} + \mathtt{d}$, and $[re \mapsto 0]val_{cl}$ denotes the clock valuation which maps (resets) each clock in re to 0 and agrees with val_{cl} over $Cl \setminus re$.

We have now obtained the correspondence between Event-B and UPTA models through their semantics definition as LTS.

4.3 Mapping from Event-B Models to UPTA

The goal in this paper is to extend the Event-B based stepwise CCD to timed systems. In the subsequent description of CCD model transformations the following notions are used:

- $\mathcal{R}^{\mathcal{T}}$ - Refinement transformation of type $\mathcal{T} \in \{\sqsubseteq_{evt}, \sqsubseteq_{\mathtt{e}}, \sqsubseteq_l\}$, where $\sqsubseteq_{evt}$ stands for *event* refinement, $\sqsubseteq_{\mathtt{e}}$and $\sqsubseteq_l$ stand for *edge* and *location* refinement respectively (elaborated in Section 5).
- $M_i^{\mathtt{T}}$- Model of type $\mathtt{T}$ resulting in i^{th} refinement step, where $\mathtt{T}$ is given as $\mathtt{T} \in \{^{B}, ^{EFSM}, ^{UPTA}\}$.
- $M_0^{\mathtt{T}}$- The initial specification of type $\mathtt{T}$.
- $\mathsf{T}^{kl} : M^{\mathtt{T}_k} \mapsto M^{\mathtt{T}_l}$ - Syntactic map of model $M^{\mathtt{T}_k}$ to $M^{\mathtt{T}_l}$, where $\mathtt{T}_k$, $\mathtt{T}_l \in \{^{B}, ^{EFSM}, ^{UPTA}\}$and $\mathtt{T}_k \neq \mathtt{T}_l$.

- $dom(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}})$ - domain of the refinement $\mathcal{R}^{\mathcal{T}}$ in the model $M_i^{\mathtt{T}}$ (Note that $dom(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}}) = \emptyset$, if $\mathcal{R}^{\mathcal{T}}$ is not defined in $M_i^{\mathtt{T}}$).
- $ran(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}})$ - co-domain of the refinement $\mathcal{R}^{\mathcal{T}}$ in the model $M_i^{\mathtt{T}}$.

Let us now state some properties of the transformations:

- For any $M_i^{\mathtt{T}_l} = ran(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}_k}) \Rightarrow \mathtt{T}_k = \mathtt{T}_l$ ($\mathcal{R}$ is conservative regarding the type of argument model)
- Submodel: $M_j^{\mathtt{T}_i} \subseteq M_i^{\mathtt{T}_i}$ iff $\forall el \in M_j^{\mathtt{T}_i} \Rightarrow el \in M_i^{\mathtt{T}_i}$
- $M_j^{\mathtt{T}} = dom(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}}) \Rightarrow M_j^{\mathtt{T}} \subseteq M_i^{\mathtt{T}}$, where $dom(\mathcal{R}^{\mathcal{T}}, M_i^{\mathtt{T}})$ is submodel of $M_i^{\mathtt{T}}$, and $M_k^{\mathtt{T}_i}$ is model complement of $M_j^{\mathtt{T}_i}$ in $M_i^{\mathtt{T}_i}$: $M_i^{\mathtt{T}_i} \setminus M_j^{\mathtt{T}_i} = \{el \mid el \in M_i^{\mathtt{T}_i},\ el \notin M^{\mathtt{T}_j},\ M_j^{\mathtt{T}_i}\}$
- $M_i^{\mathtt{T}_i} \mid M_j^{\mathtt{T}_j}$- projection of the model $M_i^{\mathtt{T}_i}$ on the model $M_j^{\mathtt{T}_j}$

Due to the fact that $M^B \mapsto M^{UPTA}$ mappings depicted in Fig. 1 preserve locality of M^B changes introduced by refinements, only those model fragments that are introduced by Event-B refinements need to be mapped to the corresponding UPTA fragments. The rest of the refined UPTA model remains same.

The Event-B to UPTA mapping proceeds in following steps:

Step 1: The Event-B model M^B is transformed to a flat Extended Finite State Machine (EFSM) model M^{EFSM} that serves as an UPTA skeleton to be decorated in the next step with UPTA specific timing attributes. The step can be implemented in two sub-steps:

Step 1.1: The Event-B specification is transformed to a Hierarchical Abstract State Transition Machine (HASTM) representation by the algorithm described in [5]. HASTM is a subclass of UML state charts without AND-parallelism.

Step 1.2: The HASTM representation is flattened by means of the algorithm introduced in [6]. The transformation result is an EFSM model M^{EFSM} with operational semantics that is equivalent to the one of the original Event-B model M^B. Thus, the proposed $M^B \mapsto M^{EFSM}$ transformation implements a total injective syntactic map. The derivation of a partially defined (i.e., without timing) UPTA model $M^{UPTA\#}$ from the M^{EFSM} is straightforward:

Let a EFSM model $M^{EFSM} = (\Sigma,\ T,\ V,\ s_0)$, be a syntactic representation of the LTS of the Event-B model M^B, where Σ is a finite set of states, V is a finite set of variables, V may include distinguished subsets I and O - of inputs and outputs respectively, T is the set of transitions, and s_0 is the initial state.

Let $M^{UPTA\#} = (L^{\#},\ E^{\#},\ V,\ \emptyset,\ l_0)$ be an UPTA model without the elements of timing specification, i.e., $L^{\#}$ is a set of locations without clock invariants and committed or urgent types, $E^{\#}$ is a set of edges without clock resets and clock conditions in the edge guards. Then we can establish the correspondence between the elements of M^{EFSM} and $M^{UPTA\#}$ as follows. $L^{\#} = \Sigma$, $E^{\#} = T$, V is the set of variables of the Event-B model M^B (as well as of M^{EFSM} due to **Step 1.2**), $l_0 = s_0$.

Step 2: In this step the partially defined UPTA model $M^{UPTA\#}$ is extended to full UPTA model M^{UPTA}. The timing constraints to be added to the newly

introduced by Event-B refinement $M^{UPTA\#}$ fragment, $M_i^{UPTA\#} = \mathsf{T}^{B,UPTA\#}[ran(\mathcal{R}^T,\ M_{i-1}^B)]$, have to satisfy also the timing constraints of the resultant model M_{i-1}^{UPTA} of the previous timing refinement step. Thus, the arguments of the timing refinement operator $\mathcal{R}_i^{\mathcal{T}}$ to be applied in i^{th} step are untimed $M_i^{UPTA\#}$ and the timing constraints of $dom(\mathcal{R}^{\mathcal{T}},\ M_{i-1}^B) \mid M_{i-1}^{UPTA})$. The rest of the timing specification M_i^{UPTA}i.e., the parts that do not concern the i^{th} refinement step remain the same as in M_{i-1}^{UPTA}. The formal definition of UPTA timing refinement and its correctness properties are defined in section 5.1.

4.4 Abstract Event-B and UPTA Specifications of Safety Controller

We can now proceed with the formal modelling of the shutdown module exemplifying the mapping from Event-B to UPTA proposed in the previous subsection. We start the modelling by defining only three externally observable variables, namely *ES*, *STO* and *SS1*, in the abstract Event-B model. The abstract Event-B specification M_0^B named *Safe* is presented in Fig. 2a. We have a configuration parameter defined in *SafeCTX* named *ss1_status* which can be set either to *active* or *nonactive*. If *ss1_status* is set to *active* then activation of *STO* should occur after the delay when *SS1* is activated. All the variables of this abstract model are of the boolean type *BOOL*. Value *TRUE* corresponds to *STO* or *SS1* being active and value *FALSE* corresponds to *STO* or *SS1* being nonactive.

At initialisation *STO* is active. Event *ES_ReleasedReact2* takes care of reseting *STO*. Events *ES_Pushed* and *ES_Released* separately model the physical act of pushing and releasing *ES* and they can be considered as input events from the environment to the system. Events *ES_React1* and *ES_React2* model the eventual reaction of the system to the pushing of *ES*; *ES_React1* corresponds to the case when *ss1_status=active* and *ES_React2* corresponds to the case when *ss1_status=nonactive*. Similarly, we distinguish between these two cases for the reaction of the system to the release of *ES* with events *ES_ReleasedReact1* and *ES_ReleasedReact2*. Already at this abstraction level we need to consider the redundancy of the system. Thus, these events are non-deterministic because both redundant systems need to first react to the pushing of *ES* and then they can be disabled. Moreover, the redundant STO outputs are disabled asynchronously. The last event *SS1_DelayReact* models the activation of *STO* after the timer triggered by the activation of *SS1* has expired. For the same reasons as above this event is non-deterministic too at this point. The initial model describes the desired functionality of the system in such a manner that is easy to get an overview of the intended behaviour.

Let us now introduce the corresponding UPTA abstract specification resulting from M_0^B *Safe* incorporating the timing requirements. We map the events of M_0^B *Safe* to edges of M_0^{UPTA} that is defined as parallel composition of automata *Safe* and *Environment*. The parallel composition of automata is needed to avoid explicit modelling of system and environment events' interleaving. Similarly, interleaving of system and environment events is implicit in Event-B. Events *ES_Pushed* and *ES_Released* of M_0^B *Safe* are modelled in automaton M_0^{UPTA} *Environment* illustrated in Fig. 2c and all other events of M_0^B *Safe* in

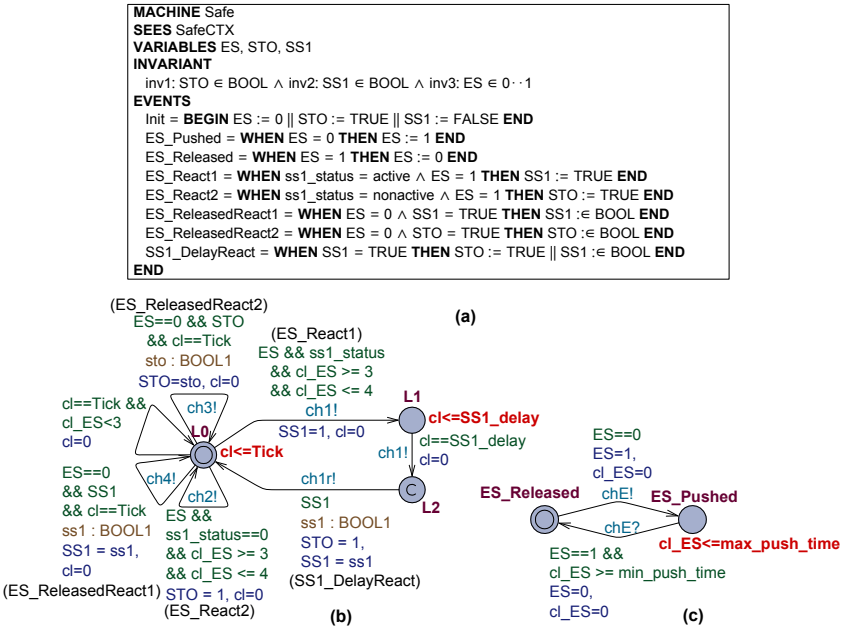


Fig. 2. (a) Event-B model M_0^B *Safe*, (b) UPTA model M_0^{UPTA} *Safe* and (c) UPTA model M_0^{UPTA} *Environment*

automaton M_0^{UPTA} *Safe* illustrated in Fig. 2b. Since the only causally dependent event pair in M_0^B is *ES_React1* and *SS1_DelayReact* their sequencing is mapped in M_0^{UPTA} to the pair of edges connected via location $L1$. Second reason for$L1$ is to model *SS1_delay*. $L2$ is introduced for technical reasons to limit the number of channels per edge in UPTA. In fact, channels depicted in Fig. 2b and 2c are obsolete for M_0^{UPTA}, they are shown only because of specifying synchronization constraints with the edges of refined model M_1^{UPTA} in Fig. 4b and 4c respectively.

The timing constraints added to M_0^B events are specified in M_0^{UPTA} by means of model clocks, clock guards and clock resets. To avoid the interference of clock constraints of parallel automata there is one local clock defined for each automaton. An extra local clock is needed only when there is simultaneously timeout bounded waiting for an external event and periodic time bound actions executed during that timeout period. For instance, the invariant $cl \leq Tick$ of location $L0$ guarantees that each time after the state variables are updated some of the enabled transitions in $L0$ will be executed latest at time instant specified by constant $Tick$. The clock guard $cl == Tick$ of edge *Idle time pass* ensures that if there is not any transition enabled within $Tick$ then at least the clock cl reset action has to be taken by executing edge *Idle time pass* with update $cl = 0$. The alternative clock *cl_ES* is needed for specifying the timing of simultaneous events *ES_Pushed* and *ES_Released*. One clock can be used also for specifying alternative time delays when used at different locations, e.g. the invariant

$cl \leq SS1\_delay$ of location $L1$ and clock guard $cl == SS1\_delay$ of edge outgoing $L1$ model the SS1 delay.

5 Proving Refinement of Timed Systems

We now have an interpretation of Event-B models as UPTA. The goal is to ensure correctness of the abstract model and its refinements. Let us first introduce the refinement definition for timed systems presented by Lynch and Vaandrager [12]. We adapt this definition in order to correspond to the UPTA semantics of section 4.2. Let cl_c and cl_a be the concrete and abstract clocks of refinement and specification model N and M respectively.

Definition 1. *A specification M is refined by a specification N, written $M \sqsubseteq N$, iff there exists a binary relation $\mathsf{R} \subseteq \Sigma^N \times \Sigma^M$ such that for each pair of states $(n, m) \in \mathsf{R}$ we have:*

1. *whenever* $n_{(l_{ref}, val_{cl_c}, val_w)} \xrightarrow{act^N} n'_{(l'_{ref}, val'_{cl_c}, val'_w)}$ *for some* $n' \in \Sigma^N$ *then* $m_{(l_{abs}, val_{cl_a}, val_v)} \xrightarrow{act^M} m'_{(l'_{abs}, val'_{cl_a}, val'_v)}$ *and* $(n', m') \in \mathsf{R}$ *for some* $m' \in \Sigma^M$
2. *whenever* $n_{(l_{ref}, val_{cl_c}, val_w)} \xrightarrow{\mathsf{d}^N} n'_{(l_{ref}, val_{cl_c}+\mathsf{d}, val_w)}$ *for* $\mathsf{d} \in \mathbb{R}_{\geq 0}$ *then* $m_{(l_{abs}, val_{cl_a}, val_v)} \xrightarrow{\mathsf{d}^M} m'_{(l_{abs}, val_{cl_a}+\mathsf{d}, val_v)}$ *and* $(n', m') \in \mathsf{R}$ *for some* $m' \in \Sigma^M$

Let us introduce the original Event-B proof obligations that are considered in this paper and needed to be discharged in order to ensure correctness of refinements. Let M and N be Event-B models, where $M \sqsubseteq N$ with refinement transformation $\mathcal{R}^{\mathcal{T}}$ of type $\mathcal{T} \in \{\sqsubseteq_{evt}\}$, where $\sqsubseteq_{evt}$ stands for event refinement. Let w be the concrete variables of model N and $J(v, w)$ be the concrete invariant stating properties on variables w and the gluing invariant between the abstract and concrete state space. For a concrete transition r of model N with guards $H(w)$ and before-after predicate $R(w, w')$ refining the abstract transition e defined in section 4.1 the following proof obligations [1] need to be discharged:

- $I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v)$ (GRD),
- $I(v) \wedge J(v, w) \wedge H(w) \wedge R(w, w') \Rightarrow \exists v'.\, S(v, v') \wedge J(v', w')$ (INV-SIM),

where obligation GRD states the guard strengthening of event e by event r, obligation INV-SIM states the preservation of the invariant after updates on variables v and w and the simulation of the abstract event e by the concrete event r. It is easy to see that condition 1 of Definition 1 can be checked by the Event-B proof obligations given above [1]. Since the UPTA models are constructed from the Event-B models we need to extend the original refinement proof obligations GRD and INV-SIM to also check the timing obligations of all conditions of Definition 1.

Let $J_t(cl_a, cl_c)$ be the concrete invariant stating properties on abstract and concrete clocks cl_a and cl_c in a timed system. Based on the untimed Event-B proof obligations given above we propose the following proof obligations elaborated with timing conditions for a concrete event r of model N with guards

$H(w)$ and $H_t(cl_c)$, before-after predicate $S(w, w')$ and clocks reset on cl_c refining the abstract event e defined in section 4.1:

- $I(v) \wedge J(v, w) \wedge I_t(cl_a) \wedge J_t(cl_a, cl_c) \wedge H(w) \wedge H_t(cl_c) \Rightarrow G(v) \wedge G_t(cl_a)$ (GRD+),
- $I(v) \wedge J(v, w) \wedge H(w) \wedge R(w, w') \wedge I_t(cl_a) \wedge J_t(cl_a, cl_c) \wedge H_t(cl_c) \wedge cl_c^{'} = 0 \Rightarrow \exists v', cl_a^{'} \cdot S(v, v') \wedge J(v', w') \wedge cl_a^{'} = 0 \wedge J_t(cl_a^{'}, cl_c^{'})$ (INV-SIM+).

Note that if there are no clock resets on the event the clock reset updates are omitted. Let us elaborate more on obligation GRD+. Invariant $J_t(cl_a, cl_c)$ can be defined as:

- $J_t(cl_a, cl_c) = J1_t(cl_a, cl_c) \wedge J2_t(cl_c)$,

where $J1_t$ is the gluing invariant expressing that (i) resets of cl_a and cl_c are synchronous when entering source locations $pre(e)$ and $pre(r)$ of events e and r, respectively, and (ii) $J_t(cl_c) \wedge H_t(cl_c) \Rightarrow I_t(cl_a) \wedge G_t(cl_a)$. $J2_t$ is the invariant of the source location of r.

The proposed INV-SIM+ proof obligation can be decomposed by logic rules to two substatements (for each pair of concrete and abstract clocks $cl_c$and cl_a):

- $I(v) \wedge J(v, w) \wedge H(w) \wedge R(w, w') \Rightarrow \exists v' \cdot S(v, v') \wedge J(v', w')$,
- $I_t(cl_a) \wedge J_t(cl_a, cl_c) \wedge H_t(cl_c) \Rightarrow J_t(0, 0)$.

The first condition corresponds to the original proof obligation within Event-B, while the second condition is the correctness condition for resetting of clocks. The general functional and timing statements introduced in this subsection are instantiated by UPTA syntax related refinement conditions introduced in the next subsection.

5.1 Superposition Refinement of UPTA

To check the timing refinement conditions in UPTA, we use an approach where the abstract UPTA model is composed in parallel with models that describe the refined parts of the abstract model. Thus, the refinements are added to the abstract model incrementally in the course of design development process and to support compositional solving of model checking tasks. For composition the refinements have a wrapping construct "context frame" that allows their uniform and easy injection into the abstract model. To keep the clear correspondence between syntactic units of the abstract model and refinement we define the refinement transformations syntactic element wise, i.e., by locations and edges of UPTA calling them *location* and *edge* refinements respectively.

Also, the course of refinement is made explicit in the model by introducing each refinement step as an increment to the resultant model of the previous refinement. It means that without changing the semantics of the abstract model we add the refinement of its syntactic element el as new automaton M^{el} that is composed with the original model M. For composition we use synchronized

parallel composition $\|_{sync}$, i.e. $M \sqsubseteq M \|_{sync} M^{el}$. Synchronization of M and M^{el} is needed to preserve the contract of el with its context after refinement. Technically, it means decorating the primary automaton M with auxiliary channel labels to synchronize the entry and leave points to/from the element el of M.

For further elaboration of the technique, we define *location refinement* ($\sqsubseteq_l$) and *edge refinement* ($\sqsubseteq_{\mathtt{e}}$) relations separately. Notice that the *event* (*guard* and *update*) *refinement* introduced in Event-B [1] can be considered as special case of edge refinement where the guards of edges are strengthened and new updates are added respectively in the refining model $M^{\mathtt{e}}$ consisting of exactly one edge.

Edge Refinement. We say that a synchronous parallel composition of automata M and $M^{\mathtt{e}_i}$ is an edge refinement for edge $\mathtt{e}_i$ of M, ($M \sqsubseteq_{\mathtt{e}} M \parallel M^{\mathtt{e}_i}$) iff:

$$\mathtt{e}_i \in E(M), \text{ and there exists } M^{\mathtt{e}_i} \textit{ such that } P_1 \wedge P_2 \wedge P_3 \wedge P_4,$$

where P_1(*interference free new updates*): No variable of M is updated in $M^{\mathtt{e}_i}$, i.e. no variable of M occurs in the left-hand side of any update in $M^{\mathtt{e}_i}$.

- P_2(*guard splitting*): Let $<l_0^{'}, l_F^{'}>$ denote a set of all feasible paths from the initial location $l_0^{'}$ to final location $l_F^{'}$ in $M^{\mathtt{e}_i}$ and $<l_0^{'}, l_F^{'}>_k \in <l_0^{'}, l_F^{'}>$ be k^{th}path in that set. Then,
  - $\forall k \in [1, |<l_0^{'}, l_F^{'}>|]. \wedge_{j\in[1,Length(k)]} G(\mathtt{e}_j^{'}) \Rightarrow G(\mathtt{e}_i)$,

i.e., the conjunction of edge guards of any path in $<l_0^{'}, l_F^{'}>$ is not weaker than the guard of the edge $\mathtt{e}_i$ refined.

- P_3(0-*duration unwinding*): $\forall l_i^{'} \in (L_{M^{\mathtt{e}_i}} \setminus l_0^{'}). \mathrm{T}(l_i^{'}) = \textit{committed}$,

i.e., all edges in the refinement $M^{\mathtt{e}_i}$ must be atomic and locations *committed*.

- P_4(*non-divergency*): $G(\mathtt{e}_i) \Rightarrow M^{\mathtt{e}}, l_0^{'} \models \mathtt{A}\Diamond l_F^{'}$,

i.e., validity of $G(\mathtt{e}_i)$ implies the existence of a feasible path in $M^{\mathtt{e}_i}$.

The context frame needed to implement the edge refinement is depicted in Fig. 3a. It includes auxiliary locations $l_0^{'}$ and $l_F^{'}$.

Location Refinement. We say that a synchronous parallel composition of automata M and M^{l_i} is a location refinement for location l_i of M, ($M \sqsubseteq_l M \parallel M^{l_i}$) iff $l_i \in L_M$, and exists M^{l_i} s.t. $P_1^{'} \wedge P_5 \wedge P_4^{'}$, where:

- $P_1^{'}$(*interference free new updates* - same as P_1 above).
- P_5(*preservation of non-blocking invariant*):
  - $[(M \parallel M^{l_i}), (l_0, l_0^{'}) \models \mathtt{E}\Diamond \mathit{deadlock}] \Rightarrow [M, l_0 \models \mathtt{E}\Diamond \mathit{deadlock}]$.
- $P_4^{'}$(*non-divergency*):
  - $inv(l_i) \equiv x \leq \mathtt{d}$ for $x \in Cl_M$, $\mathtt{d} < \infty \Rightarrow [M^{l_i}, l_0^{'} \models l_0^{'} \rightsquigarrow_{\mathtt{d}} l_F^{'}]$,

where "$\rightsquigarrow_{\mathtt{d}}$" denotes bounded reachability operator with non-negative integer time bound, Cl_M is the set of clocks of M, locations $l_0^{'}$ and $l_F^{'}$ denote respectively auxiliary pre- and post-locations in the context frame of the refinement.

P_5 and $P_4^{'}$ are specified as Uppaal model checking queries expressed in TCTL. "*deadlock*" denotes a standard predicate in Uppaal about the existence of deadlocks in the model. $P_4^{'}$ requires that the invariant of l_i is not violated due to accumulated delays of M^{l_i} runs. A graphical representation of the model fragment that schematically represents location refinement is depicted in Fig. 3b.

The auxiliary context frame of $\sqsubseteq_l$ consists of the following elements (denoted by dashed line in Fig. 3b):

- Synchronizing channel *ch* is needed to synchronize the executions of entering to and departing from location l_i transitions with those entering and departing to and from refining model M^{l_i}.
- Auxiliary initial location l_0' and final location $l_F^{'}$ of M^{l_i} are introduced to model waiting before the synchronization via channel *ch* arrives and after the execution of M^{l_i} terminates.

Location refinement can be applied when the refinement M^{l_i} specifies non-instantaneous time bounded behaviours that are represented in abstract model as location l_i.

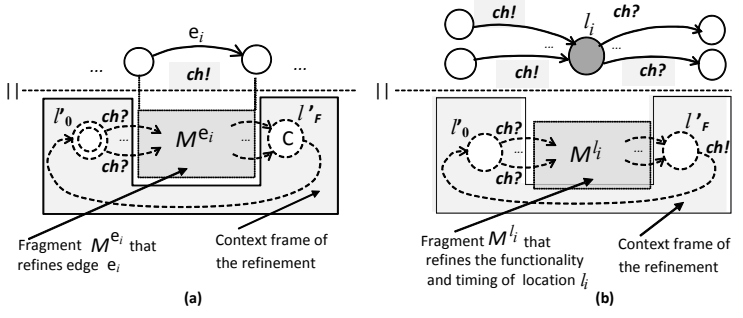


Fig. 3. (a) A fragment of the primary model M and refinement $M^{\mathtt{e}_i}$ of edge $\mathtt{e}_i$. (b) A fragment of the abstract model M with location l_i and its refinement M^{l_i}.

5.2 Event-B and UPTA Refinement of Safety Controller

Let us now exemplify the refinement theory proposed above by performing a refinement on the abstract safety controller specification. The Event-B refinement presented in this subsection introduces the required redundancy for the system. The excerpt of the Event-B refined model M_1^B *Safe1* is presented in Fig. 4a. Each (redundant) variable becomes a function from the set of CPUs (including two instances, *cpu1* and *cpu2*) to some boolean value. The new set is specified in a refined context. The refined variables are named *ES_ Redundant*, *STO_ Redundant* and *SS1_ Redundant*.

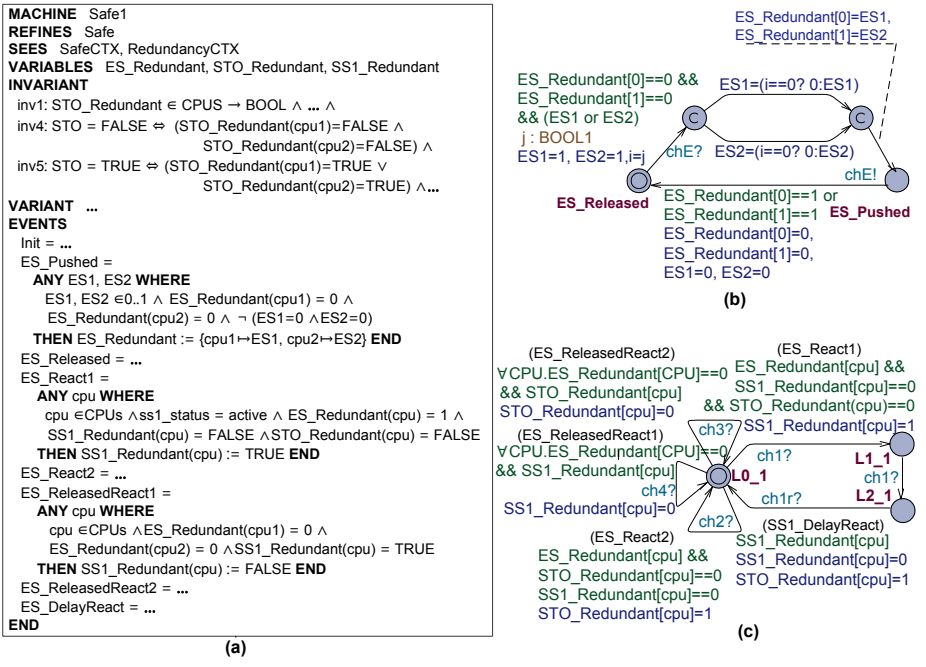


Fig. 4. (a) Event-B refinement M_1^B *Safe*1, (b) UPTA refinement M_1^{UPTA} *Safe1* and (c) UPTA model M_0^{UPTA} *Environment*

Some gluing invariants are required to relate the abstract state with the more concrete state. Considering the deactivation and activation of *STO*, we need two different invariants. If *STO* is deactivated, then none of the redundant outputs are activated (*inv4* in Fig. 4a). If *STO* is activated, then at least one of the redundant outputs is activated (*inv5* in Fig. 4a). The reason for such invariants is that it is enough for only one of the redundant inputs and outputs to work in order for the system to be safe. Similar pairwise gluing invariants exist for the other two redundant variables.

Let us focus on some of the refined events. Event *ES_Pushed* is refined as follows in order to model the redundant pushing of *ES*. Either *ES* can activate its corresponding safety function. The refined events *ES_React1* and *ES_React2* model the reaction of each CPU to its corresponding *ES*. The refinement of the previously non-deterministic events *ES_ReleasedReact1* and *ES_ReleasedReact2* takes into account the redundant CPUs. In order to allow reseting of an activated redundant safety function it is required that both redundant *ES* are released (handled by event *ES_Released*). This can handle failure of an *ES* which would cause the deactivation of its corresponding safety function.

Since only new variables and their updates are introduced by Event-B refinement $\mathcal{R}^{\sqsubseteq_{evt}} : M_0^B \rightarrow M_1^B$ the mapping $\mathsf{T}^{B,UPTA\#} : M_1^B \mapsto M_1^{UPTA\#}$ copies the

control structure of model $M_0^{UPTA\#}$ by introducing two structurally identical parallel instances $M_1^{UPTA(0)}$ and $M_1^{UPTA(1)}$ to model the redundancy. Note that both instances need their own context frame. Technically, copying the control structure of M_0^{UPTA}in M_1^{UPTA}can be considered as an aggregate model resulting from refinement steps of all edges with simple variable renaming. Since the timing of M_1^{UPTA} does not differ from that of M_0^{UPTA} the edge *Idle time pass* and location $L2$ of type *committed* added to $M_0^{UPTA\#}$ when specifying timing, do not need duplication. The edge refinement preserves the timing behaviour of M_0^{UPTA} by construction. Regardless the aggregation of several parallel refining models $M_1^{UPTA(i)}$into one in Fig. 4b, the synchronization defined by $\mathcal{R}^{\sqsubseteq_\bullet}$needs to be preserved between the edges of M_0^{UPTA} and their refinements in M_1^{UPTA}.

The correctness of $\mathcal{R}^{(\sqsubseteq_\bullet)} : M_0^{UPTA} \rightarrow M_1^{UPTA}$ follows trivially from the proof obligations of the definition of edge refinement given in Section 5.1. Both the consistency of abstract timing specification of $M_0^{UPTA}$and the correctness of timing refinement are verified by means of the Uppaal model checker. The proof obligations of this refinement are expressed in TCTL (query language of Uppaal). For instance, the reaction time requirement in M_0^{UPTA} is expressed as bounded (with time bound t) liveness property $\varphi \rightarrow_{\leq t} \psi$. The query:

– $(ES \;\&\&\; cl\_ES \geq 3) \rightarrow ((STO \; or \; SS1) \;\&\&\; gclock \leq 10)$

is satisfied for M_1^{UPTA} if and only if after starting pushing *ES* longer than 3 time units the state where *STO* or *SS1* is active is reached always within 10 time units. After timing refinement (in our example applying edge refinement as described above) one needs to model check the properties P_2 and P_4. For P_2 it suffices from checking implication between guards of refined and abstract model edges. Checking P_4 reduces to checking the queries of form $\mathtt{A}\Diamond post(r)$ where $post(r)$ denotes the post location of edge r in refined model M_1^{UPTA}. Properties P_1 and P_3 are subject to simple syntactic checks.

6 Conclusion and Future Work

We propose a correct-by-construction design workflow where model-based design transformations combine alternating data and timing constraints refinement steps. The goal is to benefit from mutually complementing formalisms Event-B and Uppaal automata and related verification techniques. For bridging the data and timing refinement steps the Event-B to UPTA map and its timing refinement transformations have been defined. That allows to verify the data refinement correctness also from its timing feasibility point of view. The approach is demonstrated on a fragment of an industrial case study of a safety critical system. The approach does not guarantee the fully incremental design flow, backtracking is needed when there is no feasible timing refinement possible for a given data refinement result. The design backtracking and error diagnostics are not addressed in the current paper. Also the automation of the proposed design transformations remain for future work.

Acknowledgement. This work has been partially funded by RECOMP project within the ARTEMIS joint undertaking (Grant agreement no. 100202). We would also like to thank the reviewers for their useful comments.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. 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)
3. Bouyer, P., Laroussinie, F., Reynier, P.-A.: Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 112–126. Springer, Heidelberg (2005)
4. Cansell, D., Méry, D., Rehm, J.: Time Constraint Patterns for Event B Development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 140–154. Springer, Heidelberg (2006)
5. Chaudhari, D.L., Damani, O.P.: Generating hierarchical state based representation from Event-B models. In: Proceedings of the B 2011 Workshop. ENTCS, vol. 280 (2011)
6. Chimisliu, V., Wotawa, F.: Abstracting timing information in UML state charts via temporal ordering and LOTOS. In: AST 2011, pp. 8–14. ACM (2011)
7. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O Automata: A complete specification theory for real-time systems. In: HSCC 2010. ACM (2011)
8. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic Abstraction Refinement for Timed Automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007)
9. International Electrotechnical Commission (IEC). IEC 61508-6: Functional safety of electrical/electronic/programmable electronic safety-related systems, 2nd edn. (2010)
10. Iliasov, A., Laibinis, L., Troubitsyna, E., Romanovsky, A., Latvala, T.: Augmenting Event-B modelling with real-time verification. Technical Report 1006, TUCS (2011)
11. Lamport, L.: Real-Time Model Checking Is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
12. Lynch, N., Vaandrager, F.: Forward and backward simulations - Part II: Timing-Based systems. Information and Computation 128 (1995)
13. Sarshogh, M.R., Butler, M.: Specification and refinement of discrete timing properties in Event-B. Technical report, Electronic and Computer Science, University of Southampton (2011)

Analysing and Closing Simulation Coverage by Automatic Generation and Verification of Formal Properties from Coverage Reports

Tim Blackmore[1], David Halliwell[1], Philip Barker[1], Kerstin Eder[2], and Naresh Ramaram[2]

[1] Infineon Technologies, Infineon House
Great Western Court, Hunts Ground Road, Bristol BS34 8HP, UK
firstname.surname@infineon.com
[2] Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
Kerstin.Eder@bristol.ac.uk

Abstract. A significant amount of time during simulation-based hardware design verification is spent analysing coverage reports in order to identify which uncovered cases are coverable and which are not, *ie* indicating areas of dead code. This dead-code analysis is typically left until the code is stable because changes to the code can mean having to start the analysis again. Some formal tools offer a push-button functionality allowing this process to be automated to some extent. This paper extends this capability of formal tools. A method is presented that automatically extracts candidates for dead code analysis from coverage reports, turns these into formal assertions and uses a formal property checker to determine whether or not the code can be reached. The core principle of the method is based on temporal induction. The method is fully automatic and generic in that it can be implemented with any state-of-the-art formal property checker; it also does not need code stability. The major benefits of employing this method in practice are a saving of engineering effort and earlier coverage closure which can avoid late discovery of bugs and schedule slips.

1 Introduction

We present a methodology that uses a formal property checker to analyse coverage holes left by module-level simulation in order to achieve early coverage closure. Coverage [1] is used to assess completeness of simulation-based verification. Coverage models are either based on the code being verified, *ie* code coverage, or on a more abstract view of the functionality of the design, *ie* functional coverage. The most common code coverage models are statement and branch coverage. It is normally regarded as a minimum requirement to achieve 100% statement and branch coverage as well as 100% functional coverage to complete verification. Having other coverage targets, such as 100% focused-expression coverage, can ensure a more thorough simulation-based verification, which may be desirable *eg* for safety critical designs.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 84–98, 2012.

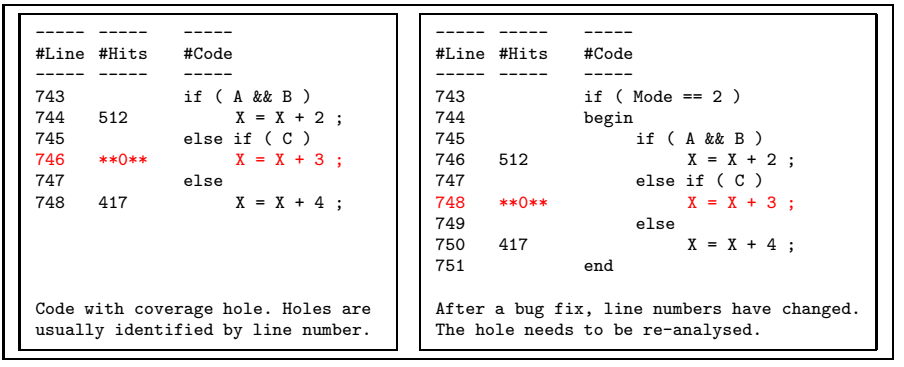


Fig. 1. Code Stability Example

Coverage targets need to be carefully defined and justified since it is rarely possible to exercise all statements, take all branches or cover all expression cases. This is for various reasons, some are given in Section 2.1. For example, it may only be possible to take 95% of the branches in the code for the configuration being verified, perhaps leaving several hundred branches uncovered. These several hundred branches have to be analysed to decide whether they can in fact be covered, if so then how, and if not, justification must be given as to why not, potentially identifying a coding error.

There are two major challenges in practice. Firstly, analysis of the missing coverage can consume a large amount of valuable engineering time. Secondly, recording of code that cannot be covered is typically based on line number or pattern recognition, both of which are susceptible to changes to the code. Indeed, changes to the code may mean that code that previously could not be covered can now be covered. To avoid this, coverage closure is typically left to near the end of the project, after sufficient code stability is achieved. This makes schedules more difficult to predict and means that bugs in the design and testbench may only be found late, both increasing the likelihood of schedule slips. An example of this code stability issue is shown in Figure 1 and is addressed by our methodology as detailed in Section 3.2.

Our methodology uses a formal property checker to overcome these two challenges by automating the analysis of coverage holes. To illustrate the fundamental idea, let us consider an uncovered branch. We observe that, to enter a branch during simulation a certain condition must be satisfied. Now, if it is possible to formally prove that this condition can never be satisfied, then clearly this branch is not coverable and can be discounted wrt branch coverage. If, however, the proof fails, then a counter example is produced by the property checker. This counter example provides an indication (rather than proof) of whether the branch really can be covered and if so how to cover it.

A similar approach, termed Coverability Analysis, has been developed at IBM to address statement coverage [2,3]. It was later extended to handle a form of expression coverage [4]. This approach requires source code instrumentation before formal analysis can be performed. The instrumentation is very similar in

style to that implemented in coverage tools to enable coverage collection during simulation. Our approach does not require any such source code modifications, it purely works with the coverage report generated by a standard coverage collection tool after simulation. Nevertheless, we review this approach in detail in Section 2 and compare it to our method.

Some formal tools have dead code checks built in, *eg* [5,6], and we compare our methodology with such built-in checks indicating why our method is both more efficient and effective, as well as being more generally applicable. Finally we report the results of using this methodology on the development of the Infineon TriCore 1.6 microcontroller.

Although the methodology includes formal verification, the development of the properties does not require formal verification expertise beyond that described in Section 3, *The Methodology*. In addition, interaction with the formal tool is encapsulated within the process thus automating away any direct interaction with this tool for a user.

This paper is organised as follows. Section 2 briefly reviews the background including coverage models, formal property checking and related work. Section 3 presents our new methodology detailing the core principle of temporal induction and how it is applied in the case of branch coverage. Section 4 gives brief details of the work flow and implementation. Section 5 contains the results of our experiments which were conducted on the TriCore 1.6 microprocessor. Section 6 presents a detailed discussion including a comparison to off-the-shelf tools. Section 7 concludes this paper with an outlook on next steps.

2 Background

2.1 Coverage

To assess the completeness of simulation-based verification engineers measure coverage. Models used for this purpose can be classified into code and functional coverage models [1]. Code coverage models are based on the code structure of the design to be verified, while functional coverage models are based on a more abstract view of the functionality of the design such as given in a functional design specification. Code coverage models are often further refined into statement, branch, expression and toggle coverage. Statement coverage reports which statements in the code have and have not been exercised during simulation. Branch coverage, sometimes referred to as decision coverage, is more detailed in that it reports on the control flow transfer in the code, *ie* on branches that have been taken or not during simulation. Expression coverage, sometimes also referred to as multiple condition coverage, is an extension of branch coverage which reports the number of times each permutation of the elements in a branch condition has made the result true or false. Finally, toggle coverage reports on which signals in the design have been toggled.

Note that 100% branch coverage implies 100% statement coverage, since if all branches have been taken then all statements have been exercised. The converse is, however, not true, *eg* an `if` statement may not have an explicit `else` branch

```
----- -----       -----
#Line #Hits       #Code
----- -----       -----
343               if ( ready && sel_mast )
344               begin
345                    if ( burst <= 3 )
346                    begin
347                         case(Mast_num)
348   1003                       Mast'h1 : master = 1;
349   ** 0 **                    Mast'h2 : master = 3;
350   5000                       default : master = 0;
351                         endcase
352                    end
353                    else if ( burst > 4 )
354                    begin
355                         case(Mast_num)
356   507                        Mast'h0 : master = 2;
357   432                        Mast'h3 : master = 4;
358   872                        default : master = 0;
359                         endcase
360                    end
361               end
```

Fig. 2. Coverage report fragment

coded, and hence no statements are associated with the `else` branch. Hence it is possible to get 100% statement coverage without ever seeing the *if* condition being false, but 100% branch coverage requires that this condition be hit true and false.

The most common code coverage models used in practice are statement and branch coverage. It is normally regarded as a minimum requirement to achieve 100% statement and branch coverage to complete verification. Because full expression coverage requires too many simulation runs to be of practical value, a scaled down version of expression coverage, called Focused Expression Coverage (FEC), is typically used in practice. FEC is a variant of Modified Condition/Decision Coverage [7]. Broadly speaking, it requires that each single element in a branch condition independently affects the result.

As mentioned earlier, coverage targets need to be carefully defined and justified. In practice it is rarely possible to exercise all statements, take all branches or cover all expression cases. The three main reasons for not being able to obtain full coverage we encountered in practice as as follows.

One is the coding style. In [4] examples are given where coding style, such as exhaustive enumeration and redundancy in expressions, can lead to uncoverable expressions. For branches, it may be quite natural to code `if .. elseif .. elseif ..` thereby enumerating all the possible cases exhaustively. However, this coding style leaves the implicit else unreachable. In fact, this type of coding style has long been known to unnecessarily increase code complexity and impede effective testing in the software domain [8]. Nevertheless, in practice this is often tolerated as a compromise to facilitate readability of code. In fact, Figure 2 illustrates this example. The implicit else branch to line `353` has a condition, shown in Figure 3 with label `branch_path_n353`, that is not reachable.

Label	Control Flow Conditions
branch_path_343	ready && sel_mast
branch_path_345	ready && sel_mast && burst <= 3
branch_path_348	ready && sel_mast && burst <= 3 && Mast_num == 1
branch_path_349	ready && sel_mast && burst <= 3 && Mast_num == 2
branch_path_350	ready && sel_mast && burst <= 3 && Mast_num != 1 && Mast_num != 2
branch_path_353	ready && sel_mast && !(burst <= 3) && burst > 4
branch_path_356	ready && sel_mast && !(burst <= 3) && burst > 4 && Mast_num == 0
branch_path_357	ready && sel_mast && !(burst <= 3) && burst > 4 && Mast_num == 3
branch_path_358	ready && sel_mast && !(burst <= 3) && burst > 4 && Mast_num != 0 && Mast_num != 3
branch_path_n353	ready && sel_mast && !(burst <= 3) && !(burst > 4)
branch_path_n343	!(ready && sel_mast)

Fig. 3. Control flow conditions extracted from the code fragment depicted in Figure 2

The second factor is configurability of designs. Designs have become so configurable that it may only be desirable to completely verify them for the configuration being used in the current system. Failure to recognize this may result in engineers spending a lot of effort verifying for configurations that are never used, *eg* various local memory sizes for microcontrollers.

The third reason for unreachable coverage encountered in our case study was reverification of legacy designs using new coverage models. Because designers tend to have less thorough knowledge of these designs they may be unwilling to remove code that appears to not be coverable, 'just in case'.

2.2 Model Checking

Formal property checkers are tools that are used to formally verify whether or not a design satisfies user-specified properties expressed as formulae in a temporal logic language such as Linear Time Temporal Logic, Computational Tree Logic or the Accellera Property Specification Language (PSL) [9]. They typically implement model checking methods which, given a design description and a temporal logic formula, fully automatically and exhaustively determine whether the formula holds for the state machine derived from the design description. If the proof fails, a counter example is provided. The counter example presents an execution trace that illustrates under what conditions the property is violated. More detailed information on model checking can be found in [10].

2.3 Related Work

The coverability of a coverage model refers to the degree to which the model can be covered during simulation [2]. Intuitively, a coverage goal, such as a statement or a branch depending on the coverage model, is coverable if and only if a test that covers this coverage goal exists. Coverability analysis is the process used to determine for all coverage goals in a given coverage model whether or not they are coverable. In [2] a first method is presented that implements coverability

analysis using symbolic model checking. The focus of this method is firstly on checking whether a variable can take all its defined values. This is determined by introducing, for each variable *var*, a set of auxiliary properties $!EF(var = V_i)$[1], one for each value V_i. The conjunction of these properties is then fed to a formal property checker. Secondly, the method is applied to statement coverage, *ie* checking whether all statements in a program are coverable. This is achieved by instrumenting the program separately for each statement S_i in two steps. First, an auxiliary variable V_i is created and initialized to 0. Then, the statement S_i is replaced with the assignment $V_i = 1$. The formula $!EF(V_i = 1)$ is then presented to a model checker. If the proof succeeds then indeed the assignment, and hence also the original statement S_i, is not reachable. Because this type of code instrumentation significantly changes the program behaviour only one statement can be checked for coverability at a time.

Because the code is modified for each statement it is necessary to recompile the code, as well as run a property, for each statement. This is likely to impact run times considerably. Our methodology requires only one compilation step to run properties for all coverage goals. Indeed, the properties produced by our methodology are so simple, they run in seconds, while compilation takes minutes. It is also difficult to see how the methodology developed in [2] can be generalised from statement coverage to branch coverage, at least not without considerable parsing of the code. The methodology presented in this paper works on branch coverage, and indeed can be generalised to any of the types of code coverage discussed in Section 2.1 above in a straightforward manner.

Overall, the coverability analysis method presented in [2] is computationally expensive and in [3] optimizations are presented to increase performance. A major modification in [3] is that the statement S_i is now replaced with the block $\{V_i = 1; S_i\}$, effectively retaining the original program behaviour. Performance improvements have been achieved by using a variety of dynamic and static techniques including inflation, static analysis and the creation of drastically reduced programs that retain the same coverability properties as the original, but are faster to compile and model check. These techniques are then combined into one algorithm for efficient coverability analysis. It was also noted that by running random simulations before starting the coverability analysis many cases, namely the ones covered during simulation, could be removed from the analysis, thereby drastically reducing the number of cases to be analysed.

In [4] the above method is extended to focused expression coverability analysis. This is achieved by adding auxiliary variables initialized to *false* to the code and also assignment statements immediately before each expression case. One variable and one assignment is needed for each FEC goal. The model checker is then called to determine whether the modified design satisfies a property that requires each of the auxiliary variables to become *true*, which corresponds to reaching each FEC goal. The experimental evaluation on two commercial designs by Motorola show the effectiveness of the extended method. It detected

[1] Read as "It is not the case that there exists a path on which eventually variable *var* has value V_i."

an impressive number of intrinsically uncoverable cases many resulted from the coding style employed at Motorola.

The methodology presented in [4] overcomes the need to recompile the design for each statement to check each property. In doing so, it does mean that greater care must be taken when instrumenting the code to ensure that the intention of the original code is not changed. *Eg* for a branch in Verilog with a single associated statement it is not necessary to bracket the statement in a begin-end block. If other statements are added during code instrumentation then the begin-end block markers must be added.

Both the statement coverability as well as the expression coverability analysis method reviewed above require source code instrumentation before formal analysis can be performed. This we view as a drawback of these methods. Our approach does not require any such source code modifications. Properties to be verified formally are generated from the coverage report obtained from a standard coverage collection tool after simulation. In addition, we note that while statement and expression coverage have been addressed, branch coverability has not been. The particular difficulty with branch coverability is the detection and correct handling of implicit branches, such as implicit `else` branches.

3 Methodology

3.1 The Principle of Temporal Induction

The core principle of our methodology is based on the notion of *temporal induction*, a form of inductive proof carried out over the time steps in a design. Like traditional inductive proof, temporal induction requires two parts: the base case and the inductive step. The behaviour of a design will vary with time as the state of the design changes. Temporal induction states that to show B is true for any state of the design it is sufficient to prove the following two properties:

(1) B holds at reset and (Base Case)
(2) $B \Rightarrow next(B)$ holds (Inductive Step)

A formal definition of the $next()$ operator can be found in *eg* the PSL Language Reference Manual [9]. We note also that it may be that B only holds after an initialisation sequence rather than at reset, and this can be dealt with similarly. Intuitively, temporal induction works as follows. The first property, Equation (1), shows B holds at reset; it is called *Base Case* or *Reset Property*. The second property, Equation (2), called *Inductive Step* or *Step Property*, shows that B holds at cycle reset+1, another application of the second property shows that B holds at cycle reset+2, and so on. Figure 4 gives a graphical illustration of the basic principles of temporal induction.

In particular, if there is a branch in the code such as

```
if X then ...
```

temporal induction can be used to show that the branch is never entered by substituting B in Equations (1) and (2) above with $not(X)$, *ie* showing that

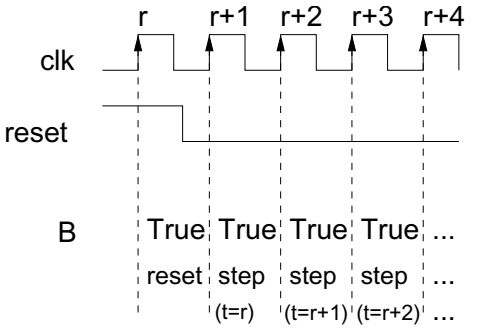


Fig. 4. The basic principle of Temporal Induction

1. $not(X)$ at reset and
2. $not(X) \Rightarrow next(not(X))$

both hold. These properties are simple in that they can be easily handled by any formal property checker in seconds and they can be derived from a coverage report via scripting. The properties are conservative in that if they pass then the branch (and any statements within the branch) definitely cannot be covered, but if they fail it may or may not be possible to cover the branch.

The properties can be strengthened while still remaining simple by analysing code structure. Thus for priority coding of branches *eg* an `else if`,

```
if X then A
else if Y ...
```

it can be proven that the `else if` branch is never entered by substituting B in Equations (1) and (2) above with $not(not(X)\ and\ Y)$, *ie* showing that

1. $not(not(X)\ and\ Y)$ at reset and
2. $not(not(X)\ and\ Y) \Rightarrow next(not(not(X)\ and\ Y))$

both hold. Similarly for nested branches:

```
if X then
    if Y then ...
```

it can be proven that the nested branch is never entered by substituting B in Equations (1) and (2) above with $not(X\ and\ Y)$, *ie* showing that

1. $not(X\ and\ Y)$ at reset and
2. $not(X\ and\ Y) \Rightarrow next(not(X\ and\ Y))$

both hold. With some understanding of the Hardware Description Language (HDL) used, these rules can be easily generalised to branches with any level of priority coding or nesting to give the highest possible chance of proving a branch (and its corresponding statements) cannot be covered.

```
macro branch_path_349
        ( ready && sel_mast && burst <= 3 && Mast_num == 2 )
endmacro;

property branch_not_covered_349_base_case =
        !(branch_path_349) @ reset;
endproperty;

property branch_not_covered_349_inductive_step =
        !(branch_path_349) => next (!(branch_path_349)) @ posedge(clk);
endproperty;

assert branch_not_covered_349_base_case;
assert branch_not_covered_349_inductive_step;
```

Fig. 5. Stepwise generation of the formal property

Essentially, temporal induction allows the proof that the design *always* behaves in a particular way by considering very small time windows. The time windows considered above are of 1 and 2 cycles. These can be lengthened to increase the likelihood of the properties passing, although this will increase run times and in practice it is unlikely to have much effect, especially when extending the time windows above 3 or 4 cycles. The small time windows have two definite advantages over a property that considers an infinite time window, such as $!EF(V_i = 1)$. Firstly the property can be run on a bounded model checker. Secondly the property will typically run in seconds on any model checker. A disadvantage is that reachable state information independent of the property must be considered separately.

3.2 Application of the Methodology

In this section we use branch coverage to illustrate how our methodology is applied to achieve branch coverage closure. Note, however, that the application of this methodology is not restricted to branch coverage; it can be applied to many coverage models including the code coverage models discussed in Section 2. We are now transferring these principles to close functional coverage.

The methodology, applied to close branch coverage, starts from analysing coverage reports after simulation. Figure 2 gives an example fragment of a coverage report consisting of three columns. The first column refers to the line number in the HDL file. The second column indicates the number of times each statement has been executed during simulation. The third column refers to the actual HDL code. Note that for line number 349 there are zero hits, *ie* there is a coverage hole at this line and coverage closure will focus on this line.

Coverage analysis is performed to extract control flow conditions from the code. Figure 3 shows the control flow conditions extracted from the code fragment depicted in Figure 2. Note that the expressions on the last two lines, the ones labeled `branch_path_n353` and `branch_path_n343`, are generated from the implicit `else` branches that complement line `353` and line `343` respectively.

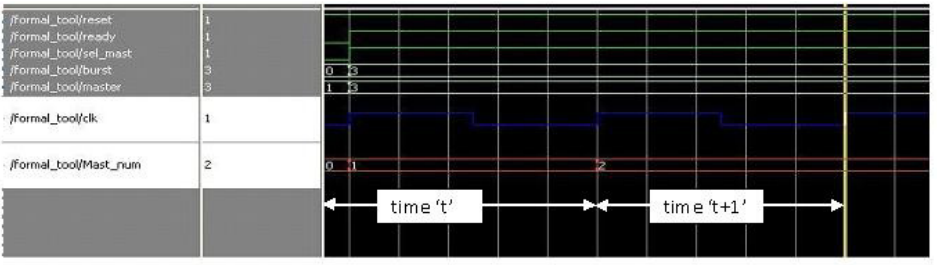


Fig. 6. Counter example waveform

Once these conditions have been extracted, a set of properties is generated for each uncovered branch. Figure 5 shows an example translation of the control flow expression for line 349 in Figure 3 to a formal property. The property is defined in three steps, namely macro generation, property definition based on the macro and property assertion. The macro generation encapsulates the control flow condition in formal syntax and assigns a name to the macro. The property definition then refers to this macro and claims that the specified condition is **not** reachable, hence the negation. The final step is asserting the property so that it can be executed by the formal property checker.

If these properties pass then the branch can be filtered from the coverage report as it is not reachable. By using a naming convention, *eg* one that contains the line number of the code, this filtering can easily be scripted. Thus the full process of reading the coverage report, generating and proving the properties, and filtering code that cannot be covered becomes push button.

If a property fails then the property checker provides a counter example that indicates how the property can be violated. Figure 6 presents a small fraction of such a counter example. Observe that the signal `Mast_num` in Figure 6 has assumed a value of `1` at time 't'. The formal tool inductively tries to prove that it holds the same value in the next time interval 't+1'. However, in this case the property fails and the property checker provides this counter example in form of a waveform. The waveform shows that the signal `Mast_num` can take a value of `2`, which indicates that the code is reachable. Further analysis of this counter example is now needed to determine how the code has been reached; this is beyond the scope of this paper.

Similarly the code stability issue can be easily handled by applying the above discussed principles to Figure 1 as shown in Figure 7. As the methodology depends on coverage holes, this assures that irrespective of any amount of changes to the code, if a particular code is found to be uncovered then it sure to be picked up for formal consideration.

Rather than running separate properties for statement coverage, this can be derived from the results of branch coverage. Thus any statement within a branch that cannot be covered can in turn not be covered. Statements outside of branches can usually be hit in a trivial way. If this is not the case (*eg* statements within a particular instance of a function) then this can normally be dealt with statically by scripts but without the need to use formal verification.

```
branch_path_746_reset;
branch_path_746_step_t;

macro branch_746
!( A && B ) && C ;
endmacro;

property
branch_path_746_step_t =
 ! branch_path_746
 => next ( ! branch_path_746 );
endproperty
```

```
branch_path_748_reset;
branch_path_748_step_t;

macro branch_748
Mode == 2 && !(A && B) && C;
endmacro;

property
branch_path_748_step_t =
! branch_path_748
=> next ( ! branch_path_748 );
endproperty
```

Fig. 7. Properties for Code Stability Example as depicted in Figure 1

3.3 Further Considerations

The methodology described above can be modified and enhanced in various ways. In this section a few are described. Firstly, the methodology can be extended in a straightforward way to deal with other coverage models, such as expression, focused expression and toggle coverage. This can all be done within the context of branching. Thus, *eg* for an expression

```
x <= A or B;
```

if the case $A = 1$ and $B = 0$ has not been simulated then the properties

1. $not(A = 1\ and\ B = 0)$ at reset and
2. $not(A = 1\ and\ B = 0) \Rightarrow next(not(A = 1\ and\ B = 0))$

can be run in an attempt to see if this case cannot be covered. If the expression is within a branch

```
if (Y) X <= A or B;
```

then the properties can be modified to

1. $not(Y\ and\ A = 1\ and\ B = 0)$ at reset and
2. $not(Y\ and\ A = 1\ and\ B = 0) \Rightarrow next(not(Y\ and\ A = 1\ and\ B = 0))$

There is a subtlety here. The properties encapsulate more information and so are intuitively more likely to pass. Indeed, this is always true of the reset property. However, the step property has both a weaker RHS and a weaker LHS. Thus it is feasible that the properties without consideration of the branching context will pass and the properties with the extra branching information will fail. It can thus be beneficial to run both sets of properties (and since they run very quickly this is not a great overhead). In practice it happens extremely rarely that the simpler properties pass and those that consider branching context fail.

Secondly, the properties can be made more likely to pass by the addition of extra (valid) assumptions. For instance, some code will not be covered because the testbench is not intended to cover it. This information can be encapsulated

in the properties quite simply in the form of extra assumptions. For example, if an input is intentionally never driven high by the testbench, or if inputs never violate a bus protocol, then this can easily be added to the properties as extra assumptions. Such assumptions can be derived from consideration of why an individual property is failing, but once derived can be used on all properties without considerably extending run times. Assumptions can also specify values of internal signals, *eg* configuration registers. As well as reflecting specified behaviour or documented restrictions, these extra assumptions should be checked as assertions during simulation to ensure that they correctly reflect testbench behaviour and that they have been coded correctly.

Going one step further, for someone with formal expertise, unreachable state information about the design can be proved and added as an assumption to all of these properties. In this way, it is possible to *formally prove* that all uncoverable code cannot be covered, although the effort for this may be considered too great (unless required for *eg* safety accreditation), but this is not the primary purpose of this methodology.

Thirdly, just the reset property can be run, but over a number of cycles *ie*

$$B \text{ holds } n \text{ numbers of cycles from reset}$$

for *eg* $n = 5$, $n = 10$, $n = 20$, etc. This may give some confidence that a certain coverage is not reachable. More importantly it can direct the user to code that is easy to cover and how to cover it. Thus coverage that fails 5 cycles from reset is likely to be easier to cover than coverage that fails for the first time 20 cycles from reset. The counter example will be easier to understand and will give a good idea of how to reach the coverage. It may be that analysis of the counter example shows that more assumptions are needed on the environment. As noted above these assumptions can then be used on all properties.

4 Implementation

The work flow of the implementation of our methodology is depicted in Figure 8.

The process of property generation and the invocation of the formal property checker are completely automated in a series of scripts, so is the filtering of unreachable coverage. Manual post processing is required for the analysis of failed properties where the tool generated a counter example.

5 Experiments and Results

This methodology was applied at Infineon Technologies during the TriCore 1.6 microcontroller verification; results are presented in Table 1.

Statement, branch and focused expression coverage (FEC) models were considered, all in the context of branch prioritisation and nesting. For statement coverage, 331 of 41074 statements were not covered during simulation and of these, 309 were proved not coverable. For branch coverage, 353 of 12341 branches

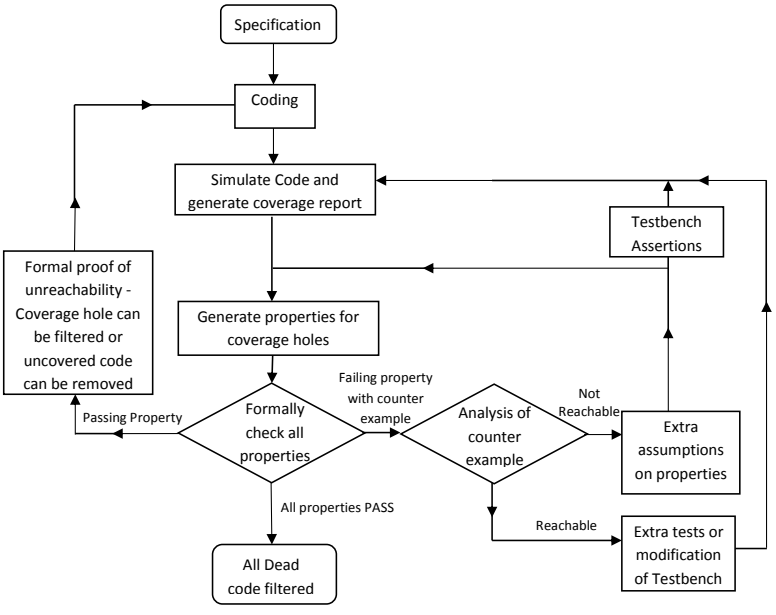


Fig. 8. Work Flow

Table 1. Experimental Results on TriCore 1.6 Design

Coverage Model	Total Coverage Goals	Coverage Holes after Simulation	Filtered Coverage Holes	% Filtered Coverage Holes
Statement	41074	331	309	93.4
Branch	12341	353	334	94.6
FEC	27230	1581	1080	68.3

were not covered during simulation and of these, 334 were proved not coverable. For FEC, 1581 of 27230 FEC goals were not covered during simulation and of these, 1080 were proved not coverable. Using this methodology meant that statement and branch coverage were achieved early in the HDL verification phase, and the number of FEC goals left for consideration could be prioritised. This directly led to the discovery of several bugs in the code, testbench and random constraints. These bugs may not have been found otherwise.

The scripts developed for use on TriCore 1.6 have been re-used with very little modification on a subsequent TriCore development, showing that after the initial effort involved in script development it is possible to re-use the scripts in an almost push-button manner. We note that in comparison to the results published in [2,3] and [4] the number of coverage goals processed in our experiments is considerably higher than the ones presented there.

6 Discussion

In this section we discuss the advantages of using our methodology and compare it to off-the shelf solutions. Firstly, because the entire process, once scripted, is fully automatic, the methodology saves considerable engineering effort. In addition, the scripts are re-usable between projects, with modifications only needed in case the HDL, coverage tool or formal tool changes. Secondly, this methodology can be applied early in the project since it does not rely on code stability. Code changes are automatically reflected in the generated properties. Engineering effort can be invested into adding assumptions to the properties as described above in order to increase the number of holding properties. Hence, code that is not covered but can be covered can be identified much earlier, allowing tests or constraints to be written, regressions improved and bugs found earlier avoiding significant late code changes. Thirdly, it has been *formally* shown that any code excluded from coverage in this manner cannot be covered. This contrasts with the standard approach, often based on informal arguments. In the context of safety-critical applications, or when re-verifying legacy code with new coverage models, this is particularly significant.

Formal tools with built-in dead code analysis provide the above advantages to some extent. However, there are some major benefits to using this methodology in terms of performance and effectiveness. The main reason that performance will be significantly better is that properties are only generated for uncovered code. This means that, even very early in a project, more than 90% of the code will not be considered which makes the difference between an overnight run and a run taking several weeks. Without any scripting a built-in solution will only be able to take advantage of the coverage information if the formal tool is integrated with the coverage tool and specifically designed to do this.

The fact that the user has control over the properties greatly improves the effectiveness of the proposed methodology compared to built-in tools. Thus, *eg* including priority coding or nesting of branches, is only possible because the user is writing the scripts that extract the properties. This alone, in our experience, already identified a greater amount of unreachable code than a built-in solution. For the TriCore 1.6 microcontroller, this methodology found 93% of uncovered statements were indeed not coverable, while a built-in tool only found 55%. Also, the addition of extra assumptions to reflect testbench scope and input behaviour may or may not be available when using a built-in solution.

We have found that built-in solutions target a limited number of coverage models, often only statement and branch coverage. Our scripted solution can be tailored for any model, including FEC and even functional coverage, provided that the functional coverage is specified in a language understood by formal tools such as System Verilog Assertions or PSL.

7 Conclusion

We have presented a truly integrated method that automatically extracts candidates for dead code analysis from simulation coverage reports, turns these into

formal properties and uses a formal property checker to determine whether or not the code can be exercised during simulation. The core principle of the method is based on temporal induction. In comparison to existing methods developed for coverability analysis our method does not require source code modifications, it purely works with the coverage report generated by a standard coverage collection tool after simulation. We are currently working on an extension of our method to functional coverage.

References

1. Piziali, A.: Functional Verification Coverage Measurement and Analysis. Springer (2004)
2. Ratzaby, G., Ur, S., Wolfsthal, Y.: Coverability Analysis Using Symbolic Model Checking. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 155–160. Springer, Heidelberg (2001)
3. Ratsaby, G., Sterin, B., Ur, S.: Improvements in Coverability Analysis. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 41–56. Springer, Heidelberg (2002)
4. Cunningham, G., Jackson, P., Dines, J.: Expression Coverability Analysis: Improving Code Coverage Analysis with Model Checking. In: Proceedings of the Design and Verification Conference (DVCon) (March 2004)
5. Andrews, M.: Tightening the Loop in Coverage Closure. Mentor Graphics, EDA Tech Forum (December 2008)
6. OneSpin Solutions GmbH: User Documentation: OneSpin 360TM, Version 4.0 (August 2006)
7. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A Practical Tutorial on Modified Condition/Decision Coverage. NASA, Technical Memorandum TM-2001-210876 (2001)
8. Watson, A.H., McCabe, T.J., Wallace, D.R.: Special Publication 500-235, Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. In: U.S. Department of Commerce/National Institute of Standards and Technology (1996)
9. Accellera: Property Specification Language Reference Manual (v1.1) (June 2004)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)

Model Checking as Static Analysis: Revisited

Fuyuan Zhang, Flemming Nielson, and Hanne Riis Nielson

DTU Informatics, Technical University of Denmark, DK-2800 Lyngby, Denmark
{fuzh,nielson,riis}@imm.dtu.dk

Abstract. We show that the model checking problem of the μ-calculus can be viewed as an instance of static analysis. We propose Succinct Fixed Point Logic (SFP) within our logical approach to static analysis as an extension of Alternation-free Least Fixed Logic (ALFP). We generalize the notion of stratification to weak stratification and establish a Moore Family result for the new logic as well. The semantics of the μ-calculus is encoded as the intended model of weakly stratified clause sequences in SFP.

1 Introduction

Both *model checking* [1, 5] and *static analysis* [7] are prominent approaches to detecting software errors. Model Checking is a successful formal method for verifying properties specified in modal logics with respect to transition systems. Static analysis is also a powerful method for validating program properties which can predict safe approximations to program behaviors.

The link between model checking and static analysis has been studied for many years. Recent research [13] takes the point of view that model checking problems can be reduced to static analysis and presents a flow logic approach to static analysis which encodes the model checking problem of *Action Computation Tree Logic* [14] in *Alternation-free Least Fixed Point Logic* (ALFP [15]). It is shown in [21] that model checking for the alternation-free μ-calculus can be encoded in ALFP as well. However, as is suggested in the negative result there, ALFP is not well-suited for the encoding of the full fragment of the μ-calculus, where nesting of the least and greatest fixed points are allowed.

Continuing these lines of work, we propose *Succinct Fixed Point Logic* (SFP) as an extension of ALFP within the framework of our logical approach to static analysis and show that the model checking problem of the μ-calculus [1, 6] can be encoded in SFP. We first propose the notion of *weak stratification* which allows a convenient specification of nested fixed points in the μ-calculus. Then, we give the definition of the *intended model* of SFP clause sequences. Unlike in ALFP, we explicitly introduce a least fixed point operator in SFP to facilitate our development. Last, we explain our approach to the analysis of the μ-calculus and show that the intended model of an SFP clause sequence specifying a μ-calculus formula exactly characterizes the set of states which satisfy this μ-calculus formula over a given Kripke structure.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 99–112, 2012.

The structure of this paper is as follows. In Section 2, we briefly introduce Kripke structure and the syntax and semantics of the μ-calculus. Section 3 explains our logical approach to static analysis, where we first review ALFP and then propose SFP, which is a main contribution of this paper. We show through an example that we cannot take the greatest lower bound of the set of models of an SFP clause sequence as the intended model, since this does not match the fixed point semantics of the μ-calculus. Section 4 is the other main contribution of our work, where we encode the model checking problem of the μ-calculus in SFP. We conclude our work in Section 5.

2 Modal μ-Calculus

2.1 Kripke Structures

The definition of *Kripke Structure* is modified slightly in comparison with [1] to distinguish different transitions in a system. Here, a Kripke structure over a set $\boldsymbol{P}$ of atomic propositions is a tuple $M = (S, T, L)$, where S is a set of states, T is a set of transition relations, and $L : S \to 2^{\boldsymbol{P}}$ labels each state with the set of true atomic propositions. Each element a in T is a transition relation and $a \subseteq S \times S$. As in [1] we also assume that the Kripke structure is total, although this is not necessary for our development.

2.2 Syntax and Semantics of the Modal μ-Calculus

Definition 1 (Syntax of the Modal μ-calculus). *Let Var be a set of variables, and $\boldsymbol{P}$ be a set of atomic propositions. The syntax of the modal μ-calculus is defined as follows:*

$$\phi ::= \ p \mid Q \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle \phi \mid [a]\phi \mid \mu Q.\phi \mid \nu Q.\phi$$

Here $p \in \mathbf{P}$, $Q \in Var$ and $a \in T$. The μ (resp. ν) operator is the least (resp. greatest) fixed point operator. For $\mu Q.\phi$ and $\nu Q.\phi$, it is required that all occurrences of Q in ϕ are under *an even number of negations* within ϕ. In this case, ϕ is said to be *syntactically monotone* in Q. A variable is *free* if it is not bound by any fixed point operator in a formula. A formula is *closed* if there are no free variables in it.

A formula ϕ is interpreted as the set of states, on a given Kripke structure, that make it true and this set of states is denoted by $[\![\phi]\!]_e$, where $e : Var \to 2^S$ is an environment. We use $e[Q \mapsto S]$ to denote the new environment updated from e by binding the relational variable Q to the set of states S. The semantics of μ-calculus formulas are defined as follows.

- $[\![p]\!]_e = \{\ s \mid p \in L(s)\ \}$
- $[\![Q]\!]_e = e(Q)$
- $[\![\neg\phi]\!]_e = S \setminus [\![\phi]\!]_e$
- $[\![\phi_1 \vee \phi_2]\!]_e = [\![\phi_1]\!]_e \cup [\![\phi_2]\!]_e$

- $[\![\phi_1 \wedge \phi_2]\!]_e = [\![\phi_1]\!]_e \cap [\![\phi_2]\!]_e$
- $[\![\langle a \rangle \phi]\!]_e = \{\ s \mid \exists s' :\ (s,\ s') \in a \texttt{ and } s' \in [\![\phi]\!]_e\}$
- $[\![[a]\phi]\!]_e = \{\ s \mid \forall s' :\ (s,\ s') \in a \texttt{ implies } s' \in [\![\phi]\!]_e\}$
- $[\![\mu Q.\phi]\!]_e$ is the least fixpoint of the function $\tau(S) = [\![\phi]\!]_{e[Q \mapsto S]}$
- $[\![\nu Q.\phi]\!]_e$ is the greatest fixpoint of the function $\tau(S) = [\![\phi]\!]_{e[Q \mapsto S]}$

The boolean operators have the usual meanings. If $(s,\ s') \in a$, we call s' an a-derivative of s. Due to the restricted use of negations in ϕ, monotonicity is guaranteed [1] for the function $\tau(S) = [\![\phi]\!]_{e[Q \mapsto S]}$. The dualities $\neg[a]\phi \equiv \langle a \rangle \neg\phi$, $\neg\langle a \rangle \phi \equiv [a]\neg\phi$, $\neg\mu Q.\phi \equiv \nu Q.\neg\phi[\neg Q/Q]$, and $\neg\nu Q.\phi \equiv \mu Q.\neg\phi[\neg Q/Q]$ are useful when transforming a formula to an equivalent form according to the semantics of the μ-calculus. The notation $\phi[\neg Q/Q]$ refers to a formula resulting from ϕ by substituting all occurrences of Q in ϕ with $\neg Q$. We give another syntax of the μ-calculus using only the μ operator as follows, which will facilitate our static analysis approach to the analysis of the μ-calculus.

Definition 2. *Let Var be a set of variables, $\boldsymbol{P}$ be a set of atomic propositions that is closed under negation. The syntax of the μ-calculus is defined as follows:*

$$\phi ::= \ p \mid Q \mid \neg Q \mid \ \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle \phi \mid [a]\phi \mid \mu Q.\phi \mid \neg\mu Q.\phi$$

where no variable is quantified twice and ϕ is syntactically monotone in Q in the cases of $\mu Q.\phi$ and $\neg\mu Q.\phi$.

3 Logical Approach to Static Analysis

In our logical approach to static analysis, we specify analysis constraints in *clause sequences*. Assume that we are given a fixed countable set $\mathcal{X}$ of variables and a finite alphabet $\mathcal{R}$ of predicate symbols. We define the syntax of clause sequences cls, together with basic values v, pre-conditions pre and clauses cl as follows:

$$\begin{array}{lll} v & ::= & c \mid x \\ pre & ::= & R(v_1, ..., v_n) \mid \neg R(v_1, ..., v_n) \mid pre_1 \wedge pre_2 \\ & & \mid pre_1 \vee pre_2 \mid \forall x :\ pre \mid \exists x :\ pre \\ cl & ::= & R(v_1, ..., v_n) \mid \textbf{true} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow R(v_1, ..., v_n) \mid \forall x :\ cl \\ cls & ::= & cl_1, ..., cl_n \end{array}$$

The pre-conditions, clauses and clause sequences are interpreted over a finite and non-empty universe $\mathcal{U}$. A constant c is an element of $\mathcal{U}$, a variable $x \in \mathcal{X}$ ranges over $\mathcal{U}$, and the n-ary relation $R \in \mathcal{R}$ denotes a subset of $\mathcal{U}^n$. We use $pre \Rightarrow R(v_1, ..., v_n)$ instead of $pre \Rightarrow cl$ which is used in [15] to simplify our development, but this does not restrict the expressiveness (merely the succinctness) of our approach.

Occurrences of $R(v_1, ..., v_n)$ and $\neg R(v_1, ..., v_n)$ in pre-conditions are called *positive queries* and *negative queries*, respectively. All other occurrences of relations

are *definitions* and often occur to the right of an implication. To deal with negations conveniently, we are often interested in some subsets of clause sequences defined by the above grammar.

Let $Int : \prod_k Rel_k \to \mathcal{P}(\mathcal{U}^k)$ be a mapping where Rel_k is a finite alphabet of k-ary predicate symbols and $\mathcal{P}(\mathcal{U}^k)$ is the powerset of $\mathcal{U}^k$. We define the satisfaction relations for pre-conditions, clauses and clause sequences (ρ, σ) $\underline{\texttt{sat}}$ pre, (ρ, σ) $\underline{\texttt{sat}}$ cl and (ρ, σ) $\underline{\texttt{sat}}$ cls in Table 1, where $\rho \in Int$ is an interpretation of relations which maps each k-ary predicate symbol R to a subset of $\mathcal{U}^k$ and σ is an interpretation of variables. We write $\rho(R)$ for the set of k-tuples $(a_1, ...a_k)$ from $\mathcal{U}$ associated with the k-ary predicate R, we use $\sigma(x)$ to denote the atom of $\mathcal{U}$ bound to x and $\sigma[x \mapsto a]$ stands for the mapping that is σ except that x is mapped to a. We also treat a constant c as a variable by setting $\sigma(c) = c$.

Table 1. Semantics of Pre-conditions, Clauses and Clause Sequences

(ρ,σ) $\underline{\texttt{sat}}$ $R(v_1,...,v_n)$	`iff` $(\sigma(v_1),...,\sigma(v_n)) \in \rho(R)$
(ρ,σ) $\underline{\texttt{sat}}$ $\neg R(v_1,...,v_n)$	`iff` $(\sigma(v_1),...,\sigma(v_n)) \notin \rho(R)$
(ρ,σ) $\underline{\texttt{sat}}$ $pre_1 \wedge pre_2$	`iff` (ρ,σ) $\underline{\texttt{sat}}$ pre_1 `and` (ρ,σ) $\underline{\texttt{sat}}$ pre_2
(ρ,σ) $\underline{\texttt{sat}}$ $pre_1 \vee pre_2$	`iff` (ρ,σ) $\underline{\texttt{sat}}$ pre_1 `or` (ρ,σ) $\underline{\texttt{sat}}$ pre_2
(ρ,σ) $\underline{\texttt{sat}}$ $\forall x:\ pre$	`iff` $(\rho,\sigma[x \mapsto a])$ $\underline{\texttt{sat}}$ pre `for all` $a \in \mathcal{U}$
(ρ,σ) $\underline{\texttt{sat}}$ $\exists x:\ pre$	`iff` $(\rho,\sigma[x \mapsto a])$ $\underline{\texttt{sat}}$ pre `for some` $a \in \mathcal{U}$
(ρ,σ) $\underline{\texttt{sat}}$ $R(v_1,...,v_n)$	`iff` $(\sigma(v_1),...,\sigma(v_n)) \in \rho(R)$
(ρ,σ) $\underline{\texttt{sat}}$ **true**	`iff` **true**
(ρ,σ) $\underline{\texttt{sat}}$ $cl_1 \wedge cl_2$	`iff` (ρ,σ) $\underline{\texttt{sat}}$ cl_1 `and` (ρ,σ) $\underline{\texttt{sat}}$ cl_2
(ρ,σ) $\underline{\texttt{sat}}$ $pre \Rightarrow R(v_1,...,v_n)$	`iff` (ρ,σ) $\underline{\texttt{sat}}$ $R(v_1,...,v_n)$ `whenever` (ρ,σ) $\underline{\texttt{sat}}$ pre
(ρ,σ) $\underline{\texttt{sat}}$ $\forall x:\ cl$	`iff` $(\rho,\sigma[x \mapsto a])$ $\underline{\texttt{sat}}$ cl `for all` $a \in \mathcal{U}$
(ρ,σ) $\underline{\texttt{sat}}$ $cl_1,...,cl_n$	`iff` (ρ,σ) $\underline{\texttt{sat}}$ cl_i `for all` i `where` $1 \le i \le n$

A clause sequence with no free variables is called *closed*, and in closed clause sequences the interpretation σ is of no importance. For a fixed interpretation σ_0, when cls is closed, we have that (ρ, σ) $\underline{\texttt{sat}}$ cls agrees with (ρ, σ_0) $\underline{\texttt{sat}}$ cls. We call an interpretation ρ a solution, or a model, of cls whenever (ρ, σ_0) $\underline{\texttt{sat}}$ cls holds.

Central to our approach to static analysis is the establishment of an *intended model* of cls. We often consider the least model of cls as a candidate, since that is the most precise analysis result. We briefly review ALFP in Section 3.1. ALFP restricts itself to the *stratified* fragment of clause sequences. The intended model of an ALFP formula is defined by the least model characterized by Moore Family properties. We propose *Succinct Fixed Point Logic* in Section 3.2. SFP restricts itself to the *weakly stratified* fragment of clause sequences. The Moore Family result of SPF is established in a slightly different way and the model of an SFP formula is defined as the least model characterized by Moore Family properties as well.

3.1 Alternation-Free Least Fixed Point Logic

Alternation-free Least Fixed Point Logic is more expressive than Datalog [19, 20] and has been used in a number of papers for specifying static analysis. It has proved to be very useful for obtaining efficient implementations of static analyses and there are a number of solvers available [17]. A clause sequence cls is called an ALFP formula iff it is stratified. The notion of *stratification* is given as follows.

Definition 3. *A clause sequence $cls = cl_1, ..., cl_n$ is stratified if there is a ranking function $\mathtt{rank} : \mathcal{R} \to \{0, ..., n\}$ such that the following holds for $0 \leq i \leq n$:*

- *if cl_i contains a definition of R then $rank(R) = i$;*
- *if cl_i contains a positive query of R then $rank(R) \leq i$; and*
- *if cl_i contains a negative query of R then $rank(R) < i$.*

Example 1. The following clause sequence is not in ALFP since it is ruled out by the notion of stratification:

$$cls = (\forall x : R_1(x) \Rightarrow R_2(x)), (\forall x : \neg R_2(x) \Rightarrow R_1(x))$$

This is because it is not possible that we have both $rank(R_1) \leq rank(R_2)$ and $rank(R_2) < rank(R_1)$.

According to the choice of ranks we have made, we define a lexicographic ordering, $\sqsubseteq$, for the interpretations of relations, ρ, as follows: $\rho_1 \sqsubseteq \rho_2$ if there exists a rank $i \in \{0, ..., r\}$ such that (1) $\rho_1(R) = \rho_2(R)$ whenever $rank(R) < i$, (2) $\rho_1(R) \subseteq \rho_2(R)$ whenever $rank(R) = i$, and (3) either $i = r$ or $\rho_1(R) \subset \rho_2(R)$ for some R with $rank(R) = i$. We define $\rho_1 \subseteq \rho_2$ to mean $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$.

The set of interpretations of relations constitutes a complete lattice with respect to $\sqsubseteq$. We know from [15] that the set of solutions to an ALFP formula constitutes a Moore Family. Recall that a Moore Family [7] is a subset Y of a complete lattice $L = (L, \sqsubseteq)$ that is closed under greatest lower bounds: $\forall Y' \subseteq Y : \bigsqcap Y' \in Y$. The Moore Family result of ALFP is given as follows:

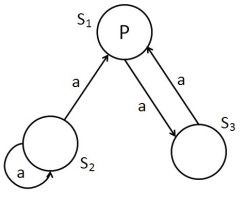
Proposition 1. *The set $\{\rho | (\rho, \sigma_0)$ $\underline{\mathtt{sat}}$ $cls\}$ is a Moore Family, i.e. is closed under greatest lower bounds, whenever cls is closed and stratified; the greatest lower bound $\sqcap \{\rho | (\rho, \sigma_0)$ $\underline{\mathtt{sat}}$ $cls\}$ is the least model of cls.*

More generally, given ρ_0 the set $\{\rho | (\rho, \sigma_0)$ $\underline{\mathtt{sat}}$ $cls \wedge \rho_0 \subseteq \rho\}$ is a Moore Family and $\sqcap \{\rho | (\rho, \sigma_0)$ $\underline{\mathtt{sat}}$ $cls \wedge \rho_0 \subseteq \rho\}$ is the least model.

The Moore Family result of ALFP formulas ensures the existence of a unique least model. We take the least model as the unique intended model of our analysis constraints specified by ALFP formulas.

ALFP suffices [21] to encode the alternation-free fragment of the μ-calculus, where nesting of least and greatest fixed points are prohibited. We give an example in the following.

Example 2. Consider a Kripke structure, given by the diagram to the left, where $S = \{s_1, s_2, s_3\}$, the transition relation $T = \{a\}$ is represented by edges labeled with a between states, and L labels s_1 with proposition p.



$\varrho(R_Q)$	$[\![\mu Q.[a](p \vee Q)]\!]$
$\{s_1, s_3\}$	$\{s_1, s_3\}$

We evaluate the formula $\mu Q.[a](p \vee Q)$ over the above Kripke structure using ALFP and the semantics of the μ-calculus respectively. The results are given in the table to the right.

In our static analysis approach, we first encode the above Kripke structure in ϱ_0 by defining $\varrho_0(P_p) = \{s_1\}$ and $\varrho_0(T_a) = \{(s_2, s_1), (s_2, s_2), (s_1, s_3), (s_3, s_1)\}$. Here, the universe is $\mathcal{U} = S$. The relation P_p specifies the set of states on which the atomic proposition p holds, and the relation T_a specifies the transition relation of the given Kripke structure. Then we specify the formula $\mu Q.[a](p \vee Q)$ with the clause sequence $cls = \forall s : \forall s' : \neg T_a(s, s') \vee P_p(s') \vee R_Q(s') \Rightarrow R_Q(s)$. The relation R_Q intends to characterize $[\![\mu Q.[a](p \vee Q)]\!]_{[]}$. The least solution ρ to cls subject to $\varrho_0 \subseteq \rho$ can be calculated by *Succinct Solver* [15].

3.2 Succinct Fixed Point Logic

The condition of stratification in ALFP requires that the definition of a relation R in cls only depends on relations with ranks less or equal to R. In particular, the requirement that a relation must be defined before they can be negatively queried is essential. This makes it inconvenient for ALFP to specify nested fixed points in the μ-calculus, where least and greatest fixed points are mutually dependent on each other.

In this section, we propose *Succinct Fixed Point Logic* (SFP) to encode nested fixed points in the μ-calculus. We first define the syntax of SFP, which include basic values v, pre-conditions pre, clauses cl, clause sequences cls and formulas f, as follows:

Definition 4 (Syntax of Succinct Fixed Point Logic)

$$
\begin{array}{lll}
v & ::= & c \mid x \\
pre & ::= & R(v_1, ..., v_n) \mid \neg R(v_1, ..., v_n) \mid pre_1 \wedge pre_2 \\
 & & \mid pre_1 \vee pre_2 \mid \forall x : \ pre \mid \exists x : \ pre \\
cl & ::= & R(v_1, ..., v_n) \mid \boldsymbol{true} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow R(v_1, ..., v_n) \mid \forall x : \ cl \\
cls & ::= & cl_1, ..., cl_n \\
f & ::= & \boldsymbol{LFP}(cls)
\end{array}
$$

where cls is weakly stratified.

Here, we require that clause sequences are weakly stratified. The definition of *weak stratification* will be given later. We introduce a least fixed point operator **LFP** and $f = \mathbf{LFP}(cls)$ is defined as SFP formulas. This is mainly to facilitate the definition of the intended model of weakly stratified clause sequences. Our intention is that ρ is the intended model of cls iff ρ satisfies the formula $\mathbf{LFP}(cls)$.

To formalize the notion of weak stratification, we first give the definition of *Dependency Graph* as follows.

Definition 5 (Dependency Graph). *The dependency graph DG_{cls} of $cls = cl_1, ..., cl_n$ is a directed graph where each edge is labeled with a sign. The nodes of DG_{cls} are cl_1,...,cl_n. We define a positive (resp. negative) edge from cl_i to cl_j iff a relation defined in cl_i is positively (resp. negatively) queried in cl_j, where $1 \leq i, j \leq n$.*

We say that *cl_j depends positively (resp. negatively)* on cl_i iff there exists a path in DG_{cls} from cl_i to cl_j with even (resp. odd) number of negative edges.

Definition 6 (Weak Stratification). *A clause sequence $cls = cl_1, ..., cl_n$ is weakly stratified iff the following conditions hold, where $1 \leq i, j \leq n$, $i \neq j$ and $R \in \mathcal{R}$:*

- *if R is defined in cl_i, then R is not defined in cl_j, and*
- *cl_i does not depend negatively on itself.*
- *if cl_i depends positively (resp. negatively) on cl_j, then cl_i does not depend negatively (resp. positively) on cl_j.*

The first condition in the above definition simply says that we use only one clause to define each relation. The second condition imposes *syntactic monotonicity* to the clause sequence. The last condition is actually used to facilitate the establishment of a Moore Family result for SFP.

Example 3. The following clause sequence satisfies the condition of weak stratification.

$$cls = (\forall x : \neg R_2(x) \Rightarrow R_1(x)), (\forall x : \neg R_1(x) \Rightarrow R_2(x))$$

Example 4. The following clause sequence is ruled out by the notion of weak stratification. We can see that the clause $(\forall x : R_2(x) \Rightarrow R_1(x))$ depends negatively on itself.

$$cls = (\forall x : R_2(x) \Rightarrow R_1(x)), (\forall x : \neg R_1(x) \Rightarrow R_2(x))$$

Let's consider the following example where we specify a μ-calculus formula of nested fixed points with a weakly stratified clause sequence.

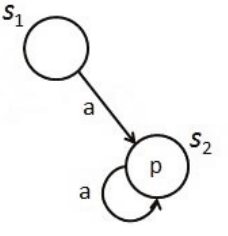
Example 5. Consider the μ-calculus formula $\phi = \mu Q_1.(\neg \mu Q_2.(Q_2 \vee (\neg Q_1 \wedge p)))$, which is semantically equivalent to $\mu Q_1.(\nu Q_2.(Q_2 \wedge (Q_1 \vee \neg p)))$ and therefore consists of nested fixed points. The formula ϕ can be specified by the following clause sequence cls.

$$cls = [\forall s : \neg R_{Q_2}(s) \Rightarrow R_{Q_1}(s)], [\forall s : [R_{Q_2}(s) \vee (\neg R_{Q_1}(s) \wedge P_p(s))] \Rightarrow R_{Q_2}(s)]$$

The clause sequence cls is weakly stratified. The relation P_p intends to specify the set of states, in a given Kripke structure, on which the atomic proposition p holds. The relation R_{Q_1} (resp. R_{Q_2}) intends to characterize $[\![\phi]\!]_{[]}$ (resp. $[\![\mu Q_2.(Q_2 \vee (\neg Q_1 \wedge p))]\!]_{[Q_1 \mapsto [\![\phi]\!]_{[]}]}$).

The next step is to define an intended model ρ of cls. In our setting, this amounts to define the semantics of formulas $f = \mathbf{LFP}(cls)$. Our intention is to use ρ to encode the fixed point semantics in the μ-calculus. Our first try is to define it in a similar way as we do in ALFP. Let's assume that all relations defined in a clause cl_i have the same rank and that all predefined relations have rank 0. However, we show through the following example that we cannot define the intended model ρ of cls as $\sqcap\{\rho|(\rho, \sigma_0)\ \underline{\mathtt{sat}}\ cls \wedge \rho_0 \subseteq \rho\}$, where ρ_0 defines all predefined relations, with respect to $\sqsubseteq$, since it does not capture the fixed point semantics.

Example 6. Consider the Kripke structure $M = (S, T, L)$, given by the diagram to the left, where $S = \{s_1, s_2\}$, $T = \{a\}$, $a = \{(s_1, s_2), (s_2, s_2)\}$, and L labels s_2 with the proposition p. We encode the μ-calculus formula $\phi = \mu Q_1.(\neg\mu Q_2.(Q_2 \vee (\neg Q_1 \wedge p)))$ in the same clause sequence $cls = [\forall s : \neg R_{Q_2}(s) \Rightarrow R_{Q_1}(s)], [\forall s : [R_{Q_2}(s) \vee (\neg R_{Q_1}(s) \wedge P_p(s))] \Rightarrow R_{Q_2}(s)]$ as we do in Example 5. We evaluate ϕ over M using SFP and the semantics of the μ-calculus respectively.



	ρ_1	ρ_2	ρ_3
R_{Q_2}	$\{s_1, s_2\}$	$\emptyset$	$\{s_2\}$
R_{Q_1}	$\emptyset$	$\{s_1, s_2\}$	$\{s_1\}$
P_p	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$

Assume we have an initial interpretation ρ_0, where $\rho_0(P_p) = \{s_2\}$ and $\rho_0(R_{Q_1}) = \rho_0(R_{Q_2}) = \emptyset$. We now consider the set of interpretations $I = \{\rho|(\rho, \sigma_0)\ \underline{\mathtt{sat}}\ cls \wedge \rho_0 \subseteq \rho\}$ according to the semantics in Table 1. There are at least three solutions ρ_1, ρ_2 and ρ_3, given in the table to the right, in the set I.

We can take at most two essentially different ranking functions $rank_1$ and $rank_2$, where $rank_1(P_p) = 0, rank_1(R_{Q_1}) = 1$ and $rank_1(R_{Q_2}) = 2, rank_2(P_p) = 0, rank_2(R_{Q_1}) = 2$ and $rank_2(R_{Q_2}) = 1$. Let $e = [Q_1 \mapsto [\![\phi]\!]_{[]}, Q_2 \mapsto [\![\mu Q_2.(Q_2 \vee (\neg Q_1 \wedge p))]\!]_{[Q_1 \mapsto [\![\phi]\!]_{[]}]}]$. According to the semantics of the μ-calculus, we know that $[\![Q_1]\!]_e = \{s_1\}$ and $[\![Q_2]\!]_e = \{s_2\}$. We can see that ρ_3 exactly characterizes the semantics of the μ-calculus in our example. However, due to the existence of ρ_1 and ρ_2, the solution ρ_3 is not the least model in I for either $rank_1$ or $rank_2$.

The method of establishing an intended model of cls in the above example can be summarized as follows. First, we calculate all the models that satisfy cls. Second, we make a choice of ranks for all those relations defined in cls. Last, we choose the least model as the intended model of cls, according to the lexicographic

ordering with respect to the choice of ranks we have made. This method applies well when we approximate an analysis where analysis information only flows from the lowest rank to the highest rank. Therefore, ALFP successfully characterizes the semantics of the alternation-free μ-calculus, where information flows from inner fixed points to outer fixed points since nesting of fixed points operators of different types are prohibited.

In the following, we define the semantics of formulas f. We assume that $cls = cl_1, ..., cl_n$ and write $\rho = \varrho_0, \varrho_1, ..., \varrho_n$ to mean that ϱ_0 is an interpretation for some predefined relations and ϱ_i $(1 \leq i \leq n)$ is an interpretation of relations defined in cl_i. We use $\rho[\varrho_i'/\varrho_i]$ to denote a new interpretation updated from ρ by substituting ϱ_i with ϱ_i'. Let ϱ_i and ϱ_i' be two interpretations of relations defined in cl_i. We define that $\varrho_i \subseteq \varrho_i'$ iff for all relations R defined in cl_i, $\varrho_i(R) \subseteq \varrho_i'(R)$ holds. The set of interpretations defined in cl_i constitute a complete lattice with respect to $\subseteq$. The satisfaction relation (ρ, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_1, ..., cl_n)$ is defined in the following.

Definition 7 (Semantics of SFP formulas). *Let $\rho = \varrho_0, ..., \varrho_n$ be an interpretation and $cls = cl_1, ..., cl_n$ a weakly stratified clause sequence. The satisfaction relation (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_1, ..., cl_n)$ is defined inductively as follows:*

- (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_n)$ *iff* $\varrho_n = \sqcap\{\varrho_n' \mid (\rho[\varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $cl_n\}$
- (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_i, ..., cl_n)$ *iff*
  1. (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_{i+1}, ..., cl_n)$, *and*
  2. $\varrho_i = \sqcap\{\varrho_i' \mid \exists \varrho_{i+1}', ..., \varrho_n' : (\rho[\varrho_i'/\varrho_i, ..., \varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $cl_i \wedge$ $(\rho[\varrho_i'/\varrho_i, ..., \varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_{i+1}, ..., cl_n)\}$

The Moore Family properties for weakly stratified clause sequence $cls = cl_1, ..., cl_n$ is established as follows.

Theorem 1. *Let $\rho = \varrho_0, ..., \varrho_n$ be an interpretation, $cls = cl_1, ..., cl_n$ a weakly stratified clause sequence and $1 \leq i \leq n$. Then, we have the followings:*

- *The set of interpretations $\{\varrho_n' \mid (\rho[\varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $cl_n\}$ is a Moore Family*
- *The set of interpretations $\{\varrho_i' \mid \exists \varrho_{i+1}', ..., \varrho_n' : (\rho[\varrho_i'/\varrho_i, ..., \varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $cl_i \wedge$ $(\rho[\varrho_i'/\varrho_i, ..., \varrho_n'/\varrho_n], \sigma)$ $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_{i+1}, ..., cl_n)\}$ is a Moore Family.*

We define the intended model of a weakly stratified clause sequence below.

Definition 8. *Assume that $cls = cl_1, ..., cl_n$ is a weakly stratified clause sequence. The model ρ is an intended model of cls iff (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_1, ..., cl_n)$.*

The Moore Family properties of SFP leads to the following theorem which guarantees the existence and the uniqueness of the intended model of cls.

Theorem 2. *Let $cls = cl_1, ..., cl_n$ be a weakly stratified clause sequence. The model ρ such that (ρ, σ) $\underline{\texttt{sat}}$ $\boldsymbol{LFP}(cl_1, ..., cl_n)$ exists and is unique.*

Example 7. Let's reconsider the problem in Example 6 again and show how to find the model $\rho = \varrho_0, \varrho_1, \varrho_2$ to the formula $\mathbf{LFP}(cls)$. Let's write $cls = cl_1, cl_2$ where $cl_1 = [\forall s : \neg R_{Q_2}(s) \Rightarrow R_{Q_1}(s)]$ and $cl_2 = [\forall s : [R_{Q_2}(s) \vee (\neg R_{Q_1}(s) \wedge P_p(s))] \Rightarrow R_{Q_2}(s)]$. According to Definition 7, (ρ, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_1, cl_2)$ iff (ρ, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_2)$ and $\varrho_1 = \sqcap\{\varrho_1' \mid \exists \varrho_2' : (\rho[\varrho_1'/\varrho_1, \varrho_2'/\varrho_2], \sigma)$ $\underline{\texttt{sat}}$ $cl_1 \wedge (\rho[\varrho_1'/\varrho_1, \varrho_2'/\varrho_2], \sigma)$ $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_2)\}$.

We first calculate the set of interpretations such that (ρ, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_2)$. To this end, we first list all the interpretations such that (ρ, σ) $\underline{\texttt{sat}}$ cl_2 in Table 2. In this case, relations P_p and R_{Q_1} are predefined relations for the clause cl_2.

Table 2. (ρ, σ) $\underline{\texttt{sat}}$ cl_2

	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8	ρ_9	ρ_{10}	ρ_{11}	ρ_{12}
R_{Q_2}	$\{s_2\}$	$\{s_1, s_2\}$	$\{s_2\}$	$\{s_1, s_2\}$	$\emptyset$	$\{s_1$	$\{s_2\}$	$\{s_1, s_2\}$	$\emptyset$	$\{s_1\}$	$\{s_2\}$	$\{s_1, s_2\}$
R_{Q_1}	$\emptyset$	$\emptyset$	$\{s_1\}$	$\{s_1\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$
P_p	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$

The next step is to select those interpretations which satisfy $\mathbf{LFP}(cl_2)$ from Table 2. From all those interpretations which coincide on predefined relations, we choose the one with the best analysis result for R_{Q_2}. Let's take ρ_1 and ρ_2 as an example. The models ρ_1 and ρ_2 coincide on their interpretations for P_p and R_{Q_1}. However, $\rho_1(R_{Q_2}) = \sqcap\{\rho_1(R_{Q_2}), \rho_2(R_{Q_2})\}$. Therefore, (ρ_1, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_2)$. The result of our selection are $\{\rho_1, \rho_3, \rho_5, \rho_9\}$. These are the interpretations which satisfy $\mathbf{LFP}(cl_2)$.

We now select those interpretations which satisfy cl_1 from $\{\rho_1, \rho_3, \rho_5, \rho_9\}$ and see that only ρ_3 and ρ_9 do. The last step is to select from ρ_3 and ρ_9 the one which satisfies $\mathbf{LFP}(cl_1, cl_2)$. Since $\rho_3(R_{Q_1}) = \sqcap\{\rho_3(R_{Q_1}), \rho_9(R_{Q_1})\}$, we know that (ρ_3, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cl_1, cl_2)$. Notice that ρ_3 exactly characterized the fixed point semantics here.

4 Model Checking as Static Analysis

Here, we use Definition 2 to give the syntax of the μ-calculus. Given a μ-calculus formula ϕ, for each variable Q in ϕ, a relation R_Q is defined. We specify our analysis with a pair $\langle cls_\phi, pre_\phi \rangle$, where cls_ϕ is a weakly stratified clause sequence and pre_ϕ is a pre-condition.

Assume that $\rho = \varrho_0, ..., \varrho_n$ such that (ρ, σ) $\underline{\texttt{sat}}$ $\mathbf{LFP}(cls_\phi)$, where ϱ_0 is an initial interpretation which encodes a given Kripke structure and defines relations $R_{Q_1}, ..., R_{Q_n}$, where $Q_1, ..., Q_n$ are all the free variables in ϕ. The intention of our development is that $s' \in [\![\phi]\!]_{e[Q_1 \mapsto S_1, ..., Q_n \mapsto S_n]}$ iff $(\rho, \sigma[s \mapsto s'])$ $\underline{\texttt{sat}}$ pre_ϕ, and that when ϕ takes the form $\mu Q.\phi$, we have that $[\![\mu Q.\phi]\!]_{e[Q_1 \mapsto S_1, ..., Q_n \mapsto S_n]}$ equals $\rho(R_Q)$.

We encode a Kripke structure $M = (S, T, L)$ into SFP by defining the corresponding relations in ϱ_0 as follows. Assume that the universe is $\mathcal{U} = S$,

Table 3. μ-calculus in Succinct Fixed Point Logic

$$
\begin{array}{lll}
p & \longmapsto & \langle \mathbf{true},\ P_p(s)\rangle \\
Q & \longmapsto & \langle \mathbf{true},\ R_Q(s)\rangle \\
\neg Q & \longmapsto & \langle \mathbf{true},\ \neg R_Q(s)\rangle \\
\phi_1 \vee \phi_2 & \longmapsto & \langle (cls_{\phi_1}, cls_{\phi_2}),\ pre_{\phi_1} \vee pre_{\phi_2}\rangle \\
 & & \textbf{whenever } \phi_1 \longmapsto \langle cls_{\phi_1},\ pre_{\phi_1}\rangle \textbf{ and } \phi_2 \longmapsto \langle cls_{\phi_2},\ pre_{\phi_2}\rangle \\
\phi_1 \wedge \phi_2 & \longmapsto & \langle (cls_{\phi_1}, cls_{\phi_2}),\ pre_{\phi_1} \wedge pre_{\phi_2}\rangle \\
 & & \textbf{whenever } \phi_1 \longmapsto \langle cls_{\phi_1},\ pre_{\phi_1}\rangle \textbf{ and } \phi_2 \longmapsto \langle cls_{\phi_2},\ pre_{\phi_2}\rangle \\
\langle a\rangle\phi & \longmapsto & \langle cls_\phi,\ \exists s' : T_a(s,s') \wedge pre_\phi[s'/s]\rangle \\
 & & \textbf{whenever } \phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle \\
[a]\phi & \longmapsto & \langle cls_\phi,\ \forall s' : \neg T_a(s,s') \vee pre_\phi[s'/s]\rangle \\
 & & \textbf{whenever } \phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle \\
\mu Q.\phi & \longmapsto & \langle ([\forall s : pre_\phi \Rightarrow R_Q(s)], cls_\phi),\ R_Q(s)\rangle \\
 & & \textbf{whenever } \phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle \\
\neg\mu Q.\phi & \longmapsto & \langle cls_{\mu Q.\phi},\ \neg R_Q(s)\rangle \\
 & & \textbf{whenever } \mu Q.\phi \longmapsto \langle cls_{\mu Q.\phi},\ pre_{\mu Q.\phi}\rangle
\end{array}
$$

- for each atomic proposition p we define a predicate P_p such that $s \in \varrho_0(P_p)$ if and only if $p \in L(s)$,
- for each element a in T, we define a binary relation T_a such that $(s,t) \in \varrho_0(T_a)$ if and only if $(s,t) \in a$.

The mapping rules for $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$ is given in Table 3. The clause sequence cls_ϕ is used to define all the relations R_Q where Q is a bounded variable in ϕ. We use $pre_\phi[s'/s]$ to denote a pre-condition resulting from pre_ϕ by substituting the free variable s in pre_ϕ with s'.

In Table 3, the choice of the ordering of clauses in cls_ϕ is essential in our approach. Assume that $cls_\phi = cl_1, ..., cl_n$. We define only one relation in each clause cl_i $(1 \leq i \leq n)$. Assume that we are given a μ-calculus formula ϕ. We call a subformula of ϕ a μ-subformula iff its main connective is μ. Assume that $\mu Q_i.\varphi_1$ and $\mu Q_j.\varphi_2$ are two μ-subformulas in ϕ and we define R_{Q_i} (resp. R_{Q_j}) in cl_i (resp. cl_j), our intention is to ensure that $i < j$ if $\mu Q_j.\varphi_2$ is a subformula of $\mu Q_i.\varphi_1$. Therefore, in the case of $\mu Q.\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$, for example, we have that $cls_{\mu Q.\phi} = ([\forall s : pre_\phi \Rightarrow R_Q(s)], cls_\phi)$ instead of $cls_{\mu Q.\phi} = (cls_\phi, [\forall s : pre_\phi \Rightarrow R_Q(s)])$.

We first explain the case of $\mu Q.\phi$. Here, Q is a bounded variable. Under the assumption that $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$ holds, we define $cls_{\mu Q.\phi}$ as $([\forall s : pre_\phi \Rightarrow R_Q(s)], cls_\phi)$. The clause $[\forall s : pre_\phi \Rightarrow R_Q(s)]$ defines the relation R_Q and the clause sequence cls_ϕ defines all those relations $R_{Q'}s$ where Q' is a bounded variable in ϕ. We define $pre_{\mu Q.\phi}$ as $R_Q(s)$.

For atomic proposition p, we simply define cls_p as **true** since there are no bounded variables in p. We make use of the predefined predicate P_p and define pre_p as $P_p(s)$. For a variable Q, we also define cls_Q as **true** since the Q is a free variable here. We define pre_Q as $R_Q(s)$. For $\neg Q$, we define $cls_{\neg Q}$ as **true** and define $pre_{\neg Q}$ as $\neg R_Q(s)$.

For $\phi_1 \vee \phi_2$, we assume that $\phi_1 \longmapsto \langle cls_{\phi_1},\ pre_{\phi_1}\rangle$ and $\phi_2 \longmapsto \langle cls_{\phi_2},\ pre_{\phi_2}\rangle$. This means that for each subformula $\mu Q.\phi$ in ϕ_1 (resp. ϕ_2), the relation R_Q is defined in cls_{ϕ_1} (resp. cls_{ϕ_2}) and that pre_{ϕ_1} and pre_{ϕ_2} are also defined as expected. We define $cls_{\phi_1\vee\phi_2}$ as $(cls_{\phi_1}, cls_{\phi_2})$. This ensures that for each bounded variable Q in $\phi_1 \vee \phi_2$, R_Q is defined in $(cls_{\phi_1}, cls_{\phi_2})$. It's natural to define $pre_{\phi_1\vee\phi_2}$ as $pre_{\phi_1} \vee pre_{\phi_2}$. The case for $\phi_1 \wedge \phi_2$ follows the same pattern.

For $\langle a\rangle\phi$, we assume that $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$. We simply define that $cls_{\langle a\rangle\phi} = cls_\phi$ and this suffices to guarantee that for each bounded variable Q in $\langle a\rangle\phi$, the relation R_Q is defined in $cls_{\langle a\rangle\phi}$. We define $pre_{\langle a\rangle\phi}$ as $\exists s' : T_a(s, s') \wedge pre_\phi[s'/s]$. This means for any state s if $pre_\phi[s'/s]$ holds on any of the a-derivative s' of s, then $pre_{\langle a\rangle\phi}$ holds on state s. This matches the semantics for $\langle a\rangle\phi$.

For $[a]\phi$, we also assume that $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$. For a similar reason as in the case for $\langle a\rangle\phi$, we define that $cls_{[a]\phi} = cls_\phi$. We define $pre_{[a]\phi}$ by $\forall s' : \neg T_a(s, s') \vee pre_\phi[s'/s]$. This means for any state s if $pre_\phi[s'/s]$ holds on all of the a-derivative s' of s, then $pre_{[a]\phi}$ holds on state s.

For $\neg\mu Q.\phi$, we assume that $\mu Q.\phi \longmapsto \langle cls_{\mu Q.\phi},\ pre_{\mu Q.\phi}\rangle$. We define that $cls_{\neg\mu Q.\phi} = cls_{\mu Q.\phi}$. We simply define $pre_{\neg\mu Q.\phi}$ as $\neg R_Q(s)$.

We have the following lemma which ensures that our specification of the μ-calculus formulas is within SFP.

Lemma 1. *Given a closed μ-calculus formula ϕ, assume that $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$ holds according to Table 3, the clause sequence cls_ϕ is closed and weakly stratified.*

The following theorem shows that the pre-condition pre_ϕ in our mapping $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$ correctly characterizes the semantics of ϕ.

Theorem 3. *Let ϕ be a μ-calculus formula with $Q_1, ..., Q_n$ being all the free variables in it. Assume that $\phi \longmapsto \langle cls_\phi,\ pre_\phi\rangle$. Let $\rho = \varrho_0, ..., \varrho_n$ be an interpretation such that (ρ, σ) $\underline{\mathtt{sat}}$ $\boldsymbol{LFP}(cls_\phi)$, where $\varrho_0(R_{Q_1}) = S_1, ..., \varrho_0(R_{Q_n}) = S_n$ and ϱ_0 defines P_p and T_a. Then, $s' \in [\![\phi]\!]_{e[Q_1\mapsto S_1,...,Q_n\mapsto S_n]}$ iff $(\rho, \sigma[s \mapsto s'])$ $\underline{\mathtt{sat}}$ pre_ϕ.*

We focus on closed μ-calculus formulas of the form $\mu Q.\phi$. This is not a restriction since $[\![\phi]\!] = [\![\mu Q.\phi]\!]$ when Q is not a free variable in ϕ. From Theorem 3, we have the following corollaries saying that the model of SFP formulas for the analysis of the μ-calculus coincides with the solution for the corresponding model checking problem.

Corollary 1. *Let $\mu Q.\phi$ be a closed μ-calculus formula. Assume that $\mu Q.\phi \longmapsto \langle cl_{\mu Q.\phi},\ pre_{\mu Q.\phi}\rangle$ holds. Let $\rho = \varrho_0, ..., \varrho_n$ be an interpretation such that (ρ, σ) $\underline{\mathtt{sat}}$ $\boldsymbol{LFP}(cls_{\mu Q.\phi})$, where ϱ_0 defines P_p and T_a. Then, we have that $[\![\mu Q.\phi]\!] = \rho(R_Q)$.*

5 Conclusion

Early works [9–12] have taken the view that static analysis problems can be reduced to model checking. In the other research direction, we have generalized the work in [13, 21] by showing that the model checking problem of the μ-calculus

can also be reduced to static analysis as well. We first propose Succinct Fixed Point Logic as a specification language which allows convenient specifications of nest fixed points in the μ-calculus and then present a mapping which can encode the full fragment of the μ-calculus to SFP. We show that μ-calculus formulas of nested fixed points can be characterized as the intended model of SFP clause sequences.

A number of previous papers (surveyed in [8, 18]) have developed a uniform approach to static analysis using ALFP as the specification language. On top of the many theoretical results established for this approach also a number of solvers have been developed [17] to calculate the least model of ALFP. ALFP can be encoded in SFP by showing that the least model of an ALFP formula can be characterized as the model of a corresponding SFP formula. This encoding is conceptually obvious and we didn't give it here.

The link between model checking and logic programming has been investigated in [22–26], where model checkers based on logic programming have been implemented. In our future work, we are interested in developing an efficient solver to calculate the model for SFP formulas so that a model checker for the μ-calculus is also implicitly implemented.

Acknowledgements. The research presented in this paper has been supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
2. Emerson, E.A., Lei, C.-L.: Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract). In: LICS 1986, pp. 267–278 (1986)
3. Cleaveland, R., Steffen, B.: A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. Formal Methods in System Design 2(2), 121–147 (1993)
4. Andersen, H.R.: Model Checking and Boolean Graphs. Theor. Comput. Sci. 126(1), 3–30 (1994)
5. Baier, C., Katoen, J.-P.: Principles of model checking, pp. I-XVII, 1-975. MIT Press (2008)
6. Kozen, D.: Results on the Propositional mu-Calculus. Theor. Comput. Sci. 27, 333–354 (1983)
7. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis (2. corr. print), pp. I-XXI, 1-452. Springer (2005)
8. Nielson, H.R., Nielson, F.: Flow Logic: A Multi-paradigmatic Approach to Static Analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
9. Steffen, B.: Data Flow Analysis as Model Checking. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 346–365. Springer, Heidelberg (1991)
10. Steffen, B.: Generating Data Flow Analysis Algorithms from Modal Specifications. Sci. Comput. Program. 21(2), 115–139 (1993)

11. Schmidt, D.A., Steffen, B.: Program Analysis *as* Model Checking of Abstract Interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
12. Schmidt, D.A.: Data Flow Analysis is Model Checking of Abstract Interpretations. In: POPL 1998, pp. 38–48 (1998)
13. Nielson, F., Nielson, H.R.: Model Checking *Is* Static Analysis of Modal Logic. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 191–205. Springer, Heidelberg (2010)
14. De Nicola, R., Vaandrager, F.W.: Action Versus State Based Logics for Transition Systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
15. Nielson, F., Seidl, H., Nielson, H.R.: A Succinct Solver for ALFP. Nord. J. Comput. 9(4), 335–372 (2002)
16. Nielson, F.: Two-Level Semantics and Abstract Interpretation. Theor. Comput. Sci. 69(2), 117–242 (1989)
17. Filipiuk, P., Nielson, H.R., Nielson, F.: Explicit Versus Symbolic Algorithms for Solving ALFP Constraints. Electr. Notes Theor. Comput. Sci. 267(2), 15–28 (2010)
18. Nielson, H.R., Nielson, F., Pilegaard, H.: Flow Logic for Process Calculi. ACM Comput. Surv. 44(1), 3 (2012)
19. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148 (1988)
20. Chandra, A.K., Harel, D.: Computable Queries for Relational Data Bases. J. Comput. Syst. Sci. 21(2), 156–178 (1980)
21. Zhang, F., Nielson, F., Nielson, H.R.: Fixpoints vs. Moore Families. Student Research Forum at SOFSEM 2012 (2012)
22. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient Model Checking Using Tabled Resolution. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
23. Ramakrishnan, C.R.: A Model Checker for Value-Passing Mu-Calculus Using Logic Programming. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 1–13. Springer, Heidelberg (2001)
24. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.N.: XMC: A Logic-Programming-Based Verification Toolset. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 576–580. Springer, Heidelberg (2000)
25. Delzanno, G., Podelski, A.: Model Checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
26. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. STTT 3(3), 250–270 (2001)

Formal Verification of Compiler Transformations on Polychronous Equations

Van Chan Ngo[1], Jean-Pierre Talpin[1], Thierry Gautier[1],
Paul Le Guernic[1], and Loïc Besnard[2]

[1] INRIA Rennes-Bretagne Atlantique, 35042 Rennes cedex, France
{Chan.Ngo,Jean-Pierre.Talpin,Thierry.Gautier,Paul.LeGuernic}@inria.fr
[2] IRISA/CNRS, 35042 Rennes cedex, France
Loic.Besnard@irisa.fr

Abstract. In this paper, adopting the translation validation approach, we present a formal verification process to prove the correctness of compiler transformations on systems of polychronous equations. We encode the source programs and the transformations with *polynomial dynamical systems* and prove that the transformations preserve the abstract clocks and clock relations of the source programs. In order to carry out the correctness proof, an appropriate relation called *refinement* and an automated proof method are presented. Each individual transformation or optimization step of the compiler is followed by our validation process which proves the correctness of this running. The compiler will continue its work if and only if the correctness is proved positively. In this paper, the highly optimizing, industrial compiler from the synchronous language SIGNAL to C is addressed.

Keywords: Formal Verification, Translation Validation, Validated Compiler, Multi-clocked Synchronous Programs, Polychronous Model.

1 Introduction

In the synchronous approaches, synchronous data-flow languages such as LUSTRE [9], SIGNAL [7] have been introduced and used successfully for the design and implementation of embedded and critical real-time systems. For the critical, high-assurance systems, the design and realization highly require an efficient and reliable implementation. Thus the systems must be verified using formal methods (e.g. model checking, etc). We want that when the compiler does not claim bugs in the formally verified source code, the generated executable code behaves as abstract clock relations semantics of the source program. However, compilation is complex and compilers involve many phases where they perform transformations over the data structures of the source program. Some transformations might be optimizations based on static analyses to eliminate inefficiencies, subexpressions in the code. Thus, bugs in the compilers can happen, making wrong executable code to be generated from correct source programs. The software industry is aware of these issues and applies many techniques to deal with them, such as

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 113–127, 2012.

manual reviews of the generated code, or testing. These techniques are not fully automated, and are expensive in terms of time and performance. An automated formal approach is applied to verify the compiler in order to prove that the semantic of the source program is preserved during the compilation is needed.

In this paper, adopting the *translation validation* approach in [15], we present an automated verification process to prove the correctness of a multi-clocked synchronous language compiler. As a part of the VERISYNC project [18], due to the very important role of abstract clock and clock relations, we are interested in proving that abstract clocks and clock relations semantics of source programs are preserved during the compilation phases of the compiler. Each individual transformation or optimization step of the compiler is followed by our verification process which proves the correctness of this running. The compiler will continue its work if and only if the correctness is proved positively. This approach avoids the disadvantage of proving in advance that the compiler always do correctly since every small change to the compiler requires reproving. Our verification framework uses polynomial dynamical systems (PDS) over a finite field, as common semantics for both source and compiled programs and a syntactic simulation-based proof which automatically proves the semantic preservation. This automated proof is implemented within the existing model checker SIGALI in the Polychronly toolset [12].

The remainder of this paper is organized as follows. Section 2 introduces the formal model of synchronous program behaviors and the automatic translation from a SIGNAL program to its formal model. In Section 3, we present our approaches to formally verify the compilation and formalize the notion of "correct translation" by means of a refinement relation between PDSs. Section 4 addresses the application of our verification approaches to the highly optimizing, industrial compiler from the synchronous language SIGNAL with the implementation which is integrated in the Polychrony toolset. Section 5 describes some related works, concludes our work and describes future work.

2 An Equational Model of Synchronous Programs

2.1 An Equational Model of the Synchronous Program Behavior

We denote by $\mathbb{Z}/p\mathbb{Z}[Z]$ the set of polynomials over variables $Z = \{z_1, ..., z_k\}$ whose coefficients range over $\mathbb{Z}/p\mathbb{Z}$, where $\mathbb{Z}/p\mathbb{Z}$ is the finite field modulo p, with p prime. For a polynomial $P \in \mathbb{Z}/p\mathbb{Z}[Z]$, the solutions of the polynomial equation $P(Z) = 0$ is denoted by $Sol(P)$. We say that $P_1 \equiv P_2$ whenever $Sol(P_1) = Sol(P_2)$. And the representative of $Sol(P)$ of each $\equiv$-equivalence class is called the *canonical generator*. In the following, we shall use some notations:

$$
\begin{aligned}
\overline{P} &\triangleq 1 - P^{p-1}. \text{ Thus } (\mathbb{Z}/p\mathbb{Z})^k \backslash Sol(P) = Sol(\overline{P}) \\
P_1 \oplus P_2 &\triangleq (P_1^{p-1} + P_2^{p-1})^{p-1} \\
P_1 \Rightarrow P_2 &\triangleq \{Z \in (\mathbb{Z}/p\mathbb{Z})^k | P_1(Z) = 0 \Rightarrow P_2(Z) = 0\} \equiv \overline{P_1} * P_2 \\
\exists z_i P &\triangleq P|_{z_i=1} * P|_{z_i=2} * ... * P|_{z_i=p} \\
\forall z_i P &\triangleq P|_{z_i=1} \oplus P|_{z_i=2} \oplus ... \oplus P|_{z_i=p}
\end{aligned}
$$

where $P|_{z_i=v}$ is P obtained by instantiating any occurrence of variable z_i by value v. The manipulations of polynomials over the finite field modulo p, with p prime can be found in [2].

Synchronous data-flow languages (e.g. LUSTRE, SIGNAL) represent data as an infinite sequence of values called *stream*, and each data stream is combined with an associated *abstract clock* as a means of discrete time. Streams and stream relations, abstract clocks and clock relations are called functional constraints and temporal constraints, respectively. The structure of synchronous programs is usually described as a series of equational definitions, the whole system is represented as systems of equations. This original structure makes that it is natural to represent the program behaviors in terms of systems of equations. The compilers of these languages, such as that we consider here, are composed of a sequence of code transformations. The transformations and optimizations that rewrite or translate source code to eliminate inefficiencies of functional constraints and temporal constraints. Some of the transformations are non-optimizing translations from a synchronous language or its intermediate language to another, lower-level language (e.g. C, Java code). Abstract clocks and clock relations are used to represent all the control parts (e.g. activation events) and interaction between different components in system. The control flow resulting from the analysis of abstract clocks and clock relations is used to derive an optimized data-flow following the transformations of the compiler. Therefore, the correctness of clock analysis in synchronous language compilation strongly impacts the quality of the compiled program. And as we have mentioned above, we would like to cope with the semantics of abstract clocks and clock constraints. In other words, our aim is to build formal models which represent the behaviors of synchronous data-flow programs in terms of the presence, absence of values in a stream (abstract clock) and the clock relations. The principle is to encode the status of a value in a stream with two possible values: *absence* and *presence*. We will use the finite field modulo $p = 3, \mathbb{Z}/3\mathbb{Z}$, i.e. integers modulo $3 : \{-1, 0, 1\}$ to encode the states of values in a data stream. For the Boolean data stream x, three possible states of x at an instant time are encoded as: $present \wedge true \rightarrow 1; present \wedge false \rightarrow -1; absent \rightarrow 0$. For the non-boolean data streams, it only encodes the fact that the value is present or absent (the clock value of the data stream is *true* or *false*): $present \rightarrow \pm 1; absent \rightarrow 0$. And the clock of a data stream is the square $x^2 : 1\ if\ present,\ 0\ if\ absent$. Thus, two synchronous data streams (they have the same clock) x and y satisfy the constraint equation: $x^2 = y^2$. It is obvious that the abstract clocks and clock relations of a synchronous data-flow program can be modeled efficiently with PDSs with coefficients ranging over $\mathbb{Z}/3\mathbb{Z}$.

Definition 1. *A PDS is a system of equations which is organized into three subsystems of polynomial equations of the form:*

$$\begin{cases} Q(X,Y) = & 0 \\ X' & = P(X,Y) \\ Q_0(X) & = 0 \end{cases}$$

where:

- X *is a set of* n *variables, called state variables, represented by a vector in* $(\mathbb{Z}/3\mathbb{Z})^n$*;*
- Y *is a set of* m *variables, called event variables, represented by a vector in* $(\mathbb{Z}/3\mathbb{Z})^m$*;*
- $X' = P(X, Y)$ *is the evolution equation of the system. It can be considered as a vectorial function* $[P_1, ..., P_n]$ *from* $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ *to* $(\mathbb{Z}/3\mathbb{Z})^n$*;*
- $Q(X, Y) = 0$ *is the constraint equation of the system. It is a vectorial equation* $[Q_1, ..., Q_l]$*;*
- $Q_0(X) = 0$ *is the initialization equation of the system. It is a vectorial equation* $[Q_{0_1}, ..., Q_{0_n}]$.

Synchronous data-flow languages use some operators requiring memorization of past value of a data stream, that is done by introducing the state variables. The vector values $(x_1, ..., x_n)$, $(x'_1, ..., x'_n)$ store respectively the past values and the current values of the data streams that are involved in the memorizing operators (e.g. SIGNAL *delay* operator). Systems of polynomial equations characterize sets of solutions, which are *states* and *events* of programs. A system of equation based method consists in manipulating the equation systems instead of the solution sets, avoiding the enumeration of the state space [2]. There is no terminal state since a synchronous data-flow program takes the input data streams that are infinite flows of values, for every state of its PDS there exist always the events to produce the next state.

2.2 Overview of the SIGNAL Language Features

In SIGNAL language [8], a signal noted as x, is a *sequence of values with the same type* $x(t_i)_{i\in\mathbb{N}}$*, which are present at some instants.* The set of instants (or time tags) where a signal is present is the *clock* of the signal, noted $\hat{x}$. A particular type of signal called *event* is characterized only by its presence, and always has the value *true*. The constructs of the language use an equational style to specify the relations between signals in the form $\mathcal{R}(x_1, ..., x_k)$, where the values of signals and the abstract clocks of signals $x_1, ..., x_k$ are the functional constraint and temporal constraint, respectively. Systems of equations on signals are built using a composition construct which defines a *process*. A whole SIGNAL program is a process which runs infinitely taking parameters, input signals for computing the output signals to react to the environment. The language is based on seven different types of equations to construct primitive processes or equations specifying computations over signals. We will present each equation along with its semantic meaning and the implicit relationships between the clocks of the input and output signals.

- *Equation on Data:* The equation $y := f(x_1, ..., x_n)$ where f is an *n-ary* relation over numerical or boolean data types, defines a process whose output $y(t)$ for tag $t \in \hat{y}$ is $y(t) = f(x_1(t), ..., x_n(t))$. The clock constraint of the input and output signals is $\hat{y} = \hat{x}_1 = ... = \hat{x}_n$.

- *Delay:* The equation $y := x\$1$ init a defines a process whose output $y(t_i) = a$ if t_i is the initial time tag, and for every other tag, $y(t_i) = x(t_{i-1})$. The clock constraint of the input and output signals is $\hat{y} = \hat{x}$.
- *Merge:* The merge equation $y := x$ default z defines a process whose output at time tag t is $y(t) = x(t)$ when $t \in \hat{x}$ and $y(t) = z(t)$ if $t \notin \hat{x} \wedge t \in \hat{y}$. The clock constraint of the merge equation is $\hat{y} = \hat{x} \cup \hat{z}$.
- *Sampling:* The sampling equation $y := x$ when b defines a process whose output signal $y(t)$ has value $x(t)$ when the signal x is present and the boolean signal b is present with the value *true*. The clock constraint of input and output signals is $\hat{y} = \hat{x} \cap [b]$ where $[b] = \{t \in \hat{b} | b(t) = true\}$.
- *Composition:* $P \triangleq P_1 \mid P_2$ where P_1 and P_2 are processes. P consists of the composition of the systems of equations. The composition operator is commutative and associative.
- *Restriction:* $P \triangleq P_1$ where x, where P_1 and x are a process and a signal, respectively. It enables local declarations in the process P_1, and leads to the same constraints as P_1.
- *Equation on clocks:* The SIGNAL language allows clock constraints to be defined *explicitly* by equations. The signal's clock is represented in SIGNAL by a special signal of type *event* which carries only a single value *true*. It specifies the presence of the signal, denoted $\hat{}x$. Thus, equations on clocks over signals are equations over their corresponding event signals. They are: *(i)* the synchronization relation $x \hat{=} y \triangleq \hat{}x = \hat{}y$, *(ii)* clock union relationship $x \hat{+} y \triangleq \hat{}x$ default $\hat{}y$, *(iii)* clock intersection relationship $x \hat{*} y \triangleq \hat{}x$ when $\hat{}y$.

Furthermore, the unary form of the sampling operation *when* b returns an event signal representing the clock of $[b]$. The special event signal $\hat{}0$ denotes the null clock (the clock that is never present).

2.3 PDS Model of SIGNAL Programs

In order to model SIGNAL programs behaviors, their processes are translated into systems of polynomial equations over $\mathbb{Z}/3\mathbb{Z}$. Each individual SIGNAL equation is translated into a polynomial equation. The language uses some primitive equations to construct programs. Thus, we only need to define the translation of these primitive equations to polynomial equations over the finite field $(\mathbb{Z}/3\mathbb{Z})^n$. The composition equation type is simply translated as the combination of the polynomial equations in the same equation system. For the equations on clocks they are derived directly from the primitive equations. Table 1 shows the translation of the primitive equations of the SIGNAL language. The delay operator \$ requires memorizing the past value of the signal, that is done by introducing the *state variable* ξ, where ξ stores the previous value of the signal and ξ' stores the current value of the signal. For example the simple SIGNAL program shown in Table 2 that specifies the alternative presence between the input signals A and B is translated in the PDS model with variables a, b, x and zx corresponding to the events A, B and boolean signals X and ZX and a state variable ξ for the delay operator. In particular, SIGNAL allows one to explicitly manipulate clocks

through some derived con- structs that can be rewritten in terms of primitive ones. For instance, $y :=$ when b is equivalent to $y := b$ when b.

Table 1. Translation of the primitive equations

Boolean signals		Non-boolean signals	
$y :=$ not x	$y = -x$	$y := f(x_1, ..., x_n)$	$y^2 = x_1^2 = ... = x_n^2$
$z :=$ x and y	$z = xy(xy - x - y - 1)$ $x^2 = y^2$		
$z :=$ x or y	$z = xy(1 - x - y - xy)$ $x^2 = y^2$		
$z :=$ x default y $z :=$ x when y	$z = x + (1 - x^2)y$ $z = x(-y - y^2)y$	$z :=$ x default y $z :=$ x when y	$z^2 = x^2 + y^2 - x^2y^2$ $z^2 = x^2(-y - y^2)$
$y :=$ x\$1 init y_0	$\xi' = x + (1 - x^2)\xi$ $y = x^2\xi$ $\xi_0 = y_0$	$y :=$ x\$1 init y_0	$y^2 = x^2$

Table 2. Program *altern* and its PDS model

```
process altern =
 ( ? event A, B;
   ! )
 (| X := not ZX
  | ZX := X$ 1
  | A ^= when X
  | B ^= when ZX
  |)
 where
 boolean X, ZX init false;
 end;
```

initial equations:
$\xi = -1$
evolution equations:
$\xi' = x + (1 - x^2) * \xi$
constraint equations:
$x = -zx, zx = \xi * x^2,$
$a^2 = -x - x^2, b^2 = -zx - zx^2$

3 Formally Verified Compilation Approaches

3.1 Definition of Correct Translation: Refinement

Given a PDS model L over the finite field $\mathbb{Z}/3\mathbb{Z}$, it can be viewed as an *intensional Labeled Transition System* (iLTS) [10] as defined in Definition 2:

Definition 2. *An intensional Labeled Transition System is a structure* $L = (Q, Y, \mathcal{I}, \mathcal{T})$*, where* Q *is a set of states,* Y *is a set of* m *variables* $Y_1, ..., Y_m$*,* $\mathcal{I}$ *is a set of initial states, and* $\mathcal{T} \subseteq Q \times \mathbb{Z}/3\mathbb{Z}[Y] \times Q$ *is the transition relation. Each transition is labeled by a polynomial over the set* Y*.*

The iLTS representation of a PDS can be obtained directly from the set of state variables, event variables, systems of initial equations, evolution equations, and constraint equations as follows:

- $Q = \mathcal{D}_X$, where $\mathcal{D}_X = \prod_{i\in[1,n]} \mathcal{D}_{x_i} = (\mathbb{Z}/3\mathbb{Z})^n$ as the domain of a set of variables $X = (x_1, ..., x_n)$
- $Y = Y, \mathcal{D}_Y = \prod_{i\in[1,m]} \mathcal{D}_{y_i} = (\mathbb{Z}/3\mathbb{Z})^m$
- $\mathcal{I} = Sol(Q_0(X))$
- $(q, P_q(Y), q') \in \mathcal{T}$ where $P_q(Y) \equiv Q(q, Y) \oplus (P(q, Y) - q')$

We write $q \xrightarrow{P(Y)} q'$ (or for short $q \xrightarrow{P} q'$), instead of $(q, P(Y), q') \in \mathcal{T}$. Then iLTSs can be viewed as an "intensional" representation of classical LTSs, where the labels are tuples in $(\mathbb{Z}/3\mathbb{Z})^m$: each arrow of the iLTS labeled by $P(Y)$ intensionally represents as many arrows labeled by some $y \in Sol(P(Y))$. We will call $Ext(L)$ the corresponding "extensional" LTS.

Definition 3. *Let $L = (Q, Y, \mathcal{I}, \mathcal{T})$ an iLTS. The infinite sequence $\sigma = q_0, y_0, q_1, y_1, q_2, y_2, ...,$ where $q_i \in Q, y_i \in \mathcal{D}_Y$ for each $i \in \mathbb{N}$, is an execution of L if it satisfies the following requirements:*

- $q_0 \in \mathcal{I}$.
- *there exists a polynomial $P(Y)$ such that $(q_i, P(Y), q_{i+1}) \in \mathcal{T} \wedge y_i \in Sol(P(Y))$ for each $i \in \mathbb{N}$.*

We denote by $\sigma_{act} = y_0, y_1, y_2, ...$ is an action-based execution, $||L||, ||L||_{act}$ the sets of executions and action-based executions of the iLTS L, respectively.

Consider the two iLTSs $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ and $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$, to which we refer respectively as a source program and a compiled program produced by a synchronous data-flow compiler. We assume that they have the same set of event variables. In case the set of event variables of the compiled model is different from the set of event variables of the source model, we consider only the common event variable and the different event variables are considered as *hiding events* [14]. Our aim is to prove that the desired behaviors of the source program are preserved during the compilation. In our case, the set of action-based executions models the desired behaviors of the program. The behaviors reflect the states of data streams and the data stream clocks constraints of the program. The strongest notion of behavior preservation during compilation is that the source program A and its compiled program C have exactly the same desired behaviors:

$$\forall \sigma_{act}.\ (\sigma_{act} \in ||C||_{act} \Leftrightarrow \sigma_{act} \in ||A||_{act}) \tag{1}$$

Requirement (1) is too strong in general to be in practical for synchronous data-flow languages. The source language is usually non-deterministic, compilers are allowed to select one of the possible behaviors of the source program. In this case, the compiled program C will have fewer behaviors than the source program A. Additionally, compilers do transformations, optimizations for removing or eliminating some wrong behaviors of the source program (e.g. eliminating subexpressions, trivial clock constraints). To address these issues, we relax the requirement (1) as follows:

$$\forall \sigma_{act}.(\sigma_{act} \in ||C||_{act} \Rightarrow \sigma_{act} \in ||A||_{act}) \tag{2}$$

Requirement (2) says that all action-based executions of C are acceptable executions of A. And we say that C *refines* A w.r.t action-based executions. We write $C \sqsubseteq A$ to denote the fact that C refines A. In the next section we present a method to establish the refinement between the two given models C and A.

3.2 Proving Refinement by Simulation

We now discuss an approach to automatically reason that a compiler preserves semantics of the source program during its compilation, in the sense of refinement relation. Given two iLTSs A and C, we propose a *symbolic simulation* for the two iLTSs to establish that $C \sqsubseteq A$. The symbolic simulation satisfies the property that if there exists a symbolic simulation for (C, A) then $C \sqsubseteq A$.

Definition 4. *Let $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ be two iLTSs. A symbolic simulation for (C, A) is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$ which satisfies the following properties:*

- *(A) $\forall q_1 \in \mathcal{I}_1, \exists q_2 \in \mathcal{I}_2, (q_1, q_2) \in \mathcal{R}$.*
- *(B) for any $(q_1, q_2) \in \mathcal{R}$ it holds that: if $q_1 \xrightarrow{P} q_1'$ there exists a finite set of transitions $(q_2 \xrightarrow{P_i} q_2^i)_{i \in I}$ (where I is a set of indexes) with*
  - *$(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ and*
  - *$(q_1', q_2^i) \in \mathcal{R}, \forall i \in I$.*

$(P \Rightarrow \prod_{i \in I} P_i) \equiv 0$ denotes that the polynomial $(P \Rightarrow \prod_{i \in I} P_i)$ is equivalent to the zero polynomial, which means that $Sol((P \Rightarrow \prod_{i \in I} P_i)) = Sol(0) = (\mathbb{Z}/3\mathbb{Z})^m$ or $Sol(P) \subseteq Sol(\prod_{i \in I} P_i)$. Condition (A) asserts that every initial state of C is related to an initial state of A. According to condition (B), for every transition of the state q_1 which is labeled by the set of events (or actions) represented by $Sol(P(Y))$, there exist some transitions of the state q_2 which are labeled by the same set of events. And it states that every outgoing transition from q_1 must be matched by outgoing transitions from q_2. Thus, Definition 4 captures exactly classic action-based simulation definition of standard LTSs. Since symbolic simulation is closed under arbitrary unions, there is a greatest symbolic simulation. In the following parts, when we talking about symbolic simulation, we imply talk about the greatest symbolic simulation.

C is simulated by A (or, equivalently, A simulates C), denoted $C \preceq A$, if there exists a symbolic simulation for (C, A). Given two states $q_1 \in Q_1$ and $q_2 \in Q_2$, the state q_1 is simulated by q_2, denoted $q_1 \preceq q_2$, if there exists a symbolic simulation $\mathcal{R}$ for (C, A) with $(q_1, q_2) \in \mathcal{R}$. In that case, we say that the two states "q_1 and q_2 are similar".

Definition 5. *Let $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ and $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ be two iLTSs. We define a family of binary relations $\preceq_j \subseteq Q_1 \times Q_2$ by induction over $j \in \mathbb{N}$.*

- $\preceq_0 \triangleq Q_1 \times Q_2$.

– $q_1 \preceq_{(j+1)} q_2$ *iff for all* $(q_1, P, q_1') \in \mathcal{T}_1$*, there exists a finite set of transitions* $(q_2, P_i, q_2^i)_{i\in I}$ *with* $(P \Rightarrow \Pi_{i\in I} P_i) \equiv 0 \wedge q_1' \preceq_j q_2^i$ *for all* $i \in I$*, where* I *is a set of indexes.*

Based on the above definition, we can now have the following theorem which gives us a method to compute the greatest symbolic simulation for two iLTSs.

Theorem 1. *Let* $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ *and* $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ *be two iLTSs.*

1. *There exists a symbolic simulation for* (C, A) *if and only if there exists a simulation for* $(Ext(C), Ext(A))$*.*
2. *Then for all* $q_1 \in Q_1$ *and* $q_2 \in Q_2, q_1 \preceq q_2$ *iff* $q_1(\bigcap_{n\in\mathbb{N}} \preceq_n) q_2$*, where* $(\bigcap_{n\in\mathbb{N}} \preceq_n) = \preceq_0 \cap \preceq_1 \cap ... \cap \preceq_n$*.*

Proof. (1) The proof can be found in [10].
(2) Since the number of state variables, event variables and the value domain of a PDS are finite then its iLTS is finite. Symbolic simulation over a finite iLTS (therefore finitely branching) is the limit of nested projective equivalences. Thus we can use the same proof method as in [16] for strong simulation. We omit the proof here.

The use of a symbolic simulation as a proof method to establish the refinement between the two given models C and A is stated in the following theorem.

Theorem 2. *Let* $C = (Q_1, Y, \mathcal{I}_1, \mathcal{T}_1)$ *and* $A = (Q_2, Y, \mathcal{I}_2, \mathcal{T}_2)$ *be two iLTSs. If there exists a symbolic simulation for* (C, A)*, then* $C \sqsubseteq A$*.*

Proof. The proof of Theorem 2 is trivial with following Lemma 1.

Lemma 1. *Let* C *and* A *be iLTSs,* $\mathcal{R}$ *is a symbolic simulation for* (C, A)*, and* $(q_1, q_2) \in \mathcal{R}$*. Then for each infinite (or finite) execution* $\sigma_1 = q_{0,1}, y_{0,1}, q_{1,1}, y_{1,1}, q_{2,1}, y_{2,1}, ...$ *starting in* $q_{0,1} = q_1$ *there exists an execution* $\sigma_2 = q_{0,2}, y_{0,2}, q_{1,2}, y_{1,2}, q_{2,1}, y_{2,2}, ...$ *from state* $q_{0,2} = q_2$ *of the same length such that* $(q_{j,1}, q_{j,2}) \in \mathcal{R}$ *and* $y_{j,1} = y_{j,2}$ *for all* j*.*

Proof. Due to the lack of space, we omit the proof here.

With an unverified compiler of synchronous data-flow language, each compilation phase is followed by our refinement verification process to provide formal guarantees as strong as those provided by a formally verified compiler. Indeed, consider the following process:

$$\begin{array}{rl} Cp'(A) = & \text{if } Cp(A) \text{ is} \\ & \quad \text{Error} \rightarrow \text{Error} \\ \mid & \quad \text{OK}(C) \rightarrow \text{ if } C \sqsubseteq A \text{ then OK}(C) \text{ else Error} \end{array}$$

where $Cp(A)$ is the compilation of A to either compiled code (written as $Cp(A) = \text{OK}(C)$) or compilation errors (written as $Cp(A) = \text{Error}$).

3.3 Composition of Compilation Phases

Compilation is always decomposed into several phases of transformations, optimizations through intermediate representations. It is better to decompose the verification process too. Fortunately, our verification process can be decomposed well thanks to the transitive property of symbolic simulation. Let A, I and C are three iLTSs, if $I \preceq A$ and $C \preceq I$ then $C \preceq A$ (the proof is trivial based on the definition of symbolic simulation). We assume that there are two compilation stages Cp_1 and Cp_2 from source program A to I and I to C, respectively. Consider the composition compilation as follows:

$$\begin{array}{rl} Cp(A) = & \text{if } Cp_1(A) \text{ is} \\ & \text{Error} \rightarrow \text{Error} \\ \mid & \text{OK}(I) \rightarrow \text{ if } I \sqsubseteq A \text{ then } Cp_2(I) \text{ else Error} \end{array}$$

It is obvious to see that the compilation $Cp(A)$ is formally verified from A to C.

4 Proving the SIGNAL Compiler

4.1 Implementation of Symbolic Simulation with SIGALI

In this section, we discuss how to implement the proof method with symbolic simulation for the two iLTSs of a source program and its compiled form using the companion model-checker of the Polychrony toolset, SIGALI. Symbolic simulation can be implemented as an extended library of SIGALI, we represent a PDS as an iLTS in the more specific form $L = (X, X', Y, \mathcal{I}, \mathcal{T})$, where:

- X, X', Y are the sets of state and event variables as in the PDS,
- $\mathcal{I}(X) = Q_0(X)$ is the polynomial representing the set of initial states, $Sol(I)$,
- $\mathcal{T}(X, Y, X') \equiv Q(X, Y) \oplus (P(X, Y) - X')$ is the polynomial representing the set of transitions.

In SIGALI, polynomials are internally represented as *ternary decision diagrams* (TDD) [5] which are an extension of *binary decision diagrams* (BDD) [1]. They are convenient for an efficient manipulation the polynomial equation systems. Theorem 1 gives us an iterative algorithm to compute the greatest symbolic simulation for (C, A). It can be obtained by computing the convergence of the sequence $(\mathcal{R}_j)_{j \in \mathbb{N}}$ as in Algorithm 1 which can be efficiently implemented with the fixed-point computation of the SIGALI kernel (see Appendix B). The correctness of Algorithm 1 is proved by the following proposition.

Proposition 1. *For all* $j \in \mathbb{N}, \mathcal{R}_j(x_1, x_2) = 0$ *if and only if* $x_1 \preceq_j x_2$.

Proof. $\Rightarrow$) We use an induction proving method over j. It holds obviously with $j = 0$. Assume that we have $\mathcal{R}_{j+1}(x_1, x_2) = 0$ and let $x_1 \xrightarrow{P} x_1'$ be a transition in C. It is clear that $P(Y) \equiv \mathcal{T}_1(x_1, Y, x_1')$. We define the polynomial $Q(Y) \equiv \exists x_2' \mathcal{T}_2(x_2, Y, x_2') \oplus \mathcal{R}_j(x_1', x_2')$, $\mathcal{R}_j$ being computed in Algorithm 1 above. This

Algorithm 1. Compute symbolic simulation $\mathcal{R}(X_1, X_2)$

Require: $C = (X_1, X_1', Y, \mathcal{I}_1, \mathcal{T}_1), A = (X_2, X_2', Y, \mathcal{I}_2, \mathcal{T}_2)$
Ensure: $\mathcal{R}(X_1, X_2)$
1: $\mathcal{R}_0(X_1, X_2) \equiv 0$
2: **while** $\mathcal{R}_j(X_1, X_2)$ is not convergent **do**
3: $\mathcal{R}_{j+1}(X_1, X_2)$ is the canonical generator of the $\equiv$-class of:
4: $\mathcal{R}_j(X_1, X_2) \oplus$
5: $\forall X_1' \forall Y[(\mathcal{T}_1(X_1, Y, X_1') \Rightarrow \exists X_2'(\mathcal{T}_2(X_2, Y, X_2') \oplus \mathcal{R}_j(X_1', X_2'))]$
6: **end while**
7: **if** $\forall X_1[(\mathcal{I}_1(X_1) \Rightarrow \exists X_2(\mathcal{I}_2(X_2) \oplus \mathcal{R}(X_1, X_2))]$ **then**
8:
9: **return** $\mathcal{R}(X_1, X_2)$
10: **else**
11: **return** $\mathcal{R}(X_1, X_2) \equiv 1$
12: **end if**

polynomial captures the set $\{y | \exists x_2 \xrightarrow{P_i} x_2^i, P_i(y) = 0 \wedge x_1' \preceq_j x_2^i\}$. By the definition of $\mathcal{R}_{j+1}$, the y value is in $Sol(\mathcal{T}_1(x_1, Y, x_1'))$, thus $Sol(P(Y)) \subseteq \bigcup_i Sol(P_i)$, which means $x_1 \preceq_{(j+1)} x_2$.
$\Leftarrow$) We can apply again an induction method over j similar to the proof of the Theorem 1. Thus we omit it here.

Proposition 2. *Algorithm 1 terminates and at the end,* $\mathcal{R}(x_1, x_2) = 0$ *if and only if* $x_1 \preceq x_2$.

Proof. Termination is guaranteed by the fact that relations $\mathcal{R}_j$ are finite and nested. The second statement is a corollary of Proposition 1 and Theorem 1.

4.2 Proving the Compiler Transformations

The compiler of the SIGNAL language [3] that we consider is composed of a sequence of code transformations. Some transformations are optimizations that rewrite the code to eliminate subexpressions, inefficiencies. The compilation process may be seen as a sequence of morphisms rewriting SIGNAL programs to SIGNAL programs. And the final steps (C or Java code generation) are simple morphisms over the ultimately transformed SIGNAL program. For convenience, the transformations of the compiler are classed into three stages:

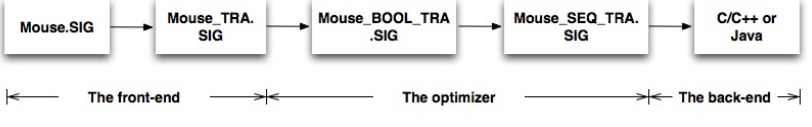


Fig. 1. Scheme of the SIGNAL compiler

- *The front-end*: non-optimizing translations from the source program in SIGNAL language to a program in SIGNAL language. The clock information of all signals in the source program is calculated, which is called *clock calculus*.
- *The optimizer*: the synchronization and precedence relations of all signals and clocks are represented in a directed labeled graph structure called the *Data Control Graph* (DCG); it is composed of a *Clock Hierarchy* (CH) and a *Conditioned Precedence Graph* (CPG). A node of this CPG is a primitive equation or, in a hierarchical organization, a composite SIGNAL process containing its own DCG. Then the optimizations are performed on the output of the front-end stage based on the DCG.
- *The back-end*: translations from the optimized final SIGNAL program to executable code (C/C++ or Java).

For instance, consider a source program called *Mouse.SIG* (example program available in the online examples of the Polychrony toolset), the transformations of the stages front-end, optimizer, back-end are *Mouse_TRA.SIG*, *Mouse_BOOL_TRA.SIG*, and *Mouse_SEQ_TRA.SIG*, respectively.

The optimized final program *Mouse_SEQ_TRA.SIG* is translated directly to executable code. We are interested in the first two stages of the compiler: the non-optimizing translations and the optimizations. The intermediate forms in the transformations of the compiler may be expressed in the SIGNAL language itself. Moreover the Polychrony toolset provides a function to translate a SIGNAL program into a PDS over the finite field $\mathbb{Z}/3\mathbb{Z}$. Then the correctness of the compiler is proved in each transformation of the two first stages. For instance, we consider the compilation of *Mouse.SIG* program, the verification asserts that $Mouse\_SEQ\_TRA.SIG \preceq Mouse\_BOOL\_TRA.SIG \preceq Mouse\_TRA.SIG \preceq Mouse.SIG$ along the transformations of the SIGNAL compiler.

Experimental Results. We here provide some experimental results verifying the transformations of the SIGNAL compiler with a simulation based proof method. The experimental results deal with the complexity of the symbolic simulation computation. All the examples here are available in the online examples of the Polychrony toolset. In the X, Y, 'Correct' columns, we write the numbers of state variables, event variables and the correctness of the compiler transformations, respectively (hence, the transition relation $\mathcal{T}(X, Y, X')$ will have $2X + Y$ variables). We measure description complexity of the symbolic simulation by the size of fix point computation in Algorithm 1 (in terms of the number of TDD nodes that we need to represent the manipulation of polynomial equation systems). The number of TDD nodes is showed in SIGALI model checker only when it is big enough, so for the tests whose numbers of TDD nodes are not showed we write "Small". We denote $\mathcal{R}_1(X_1, X_2), \mathcal{R}_2(X_1, X_2), \mathcal{R}_3(X_1, X_2)$ are symbolic simulations for $(A\_TRA.z3z, A.z3z)$,$(A\_BOOL\_TRA.z3z, A\_TRA.z3z)$, and $(A\_SEQ\_TRA.z3z, A\_BOOL\_TRA.z3z)$, respectively, for the compilation of the SIGNAL program, called A.

Table 3. Experimental results

Name	X	Y	$\mathcal{R}_1(X_1, X_2)$ TDD nodes	$\mathcal{R}_2(X_1, X_2)$ TDD nodes	$\mathcal{R}_3(X_1, X_2)$ TDD nodes	Correct
MOUSE.z3z	2	5	Small	Small	Small	Yes
MOUSE_TRA.z3z	2	5				
MOUSE_BOOL_TRA.z3z	2	6				
MOUSE_SEQ_TRA.z3z	2	6				
RAILROADCROSSING.z3z	2	40	Small	Small	Small	Yes
RRCROSSING_TRA.z3z	2	40				
RRCROSSING_BOOL_TRA.z3z	2	39				
RRCROSSING_SEQ_TRA.z3z	2	39				
CHRONOMETER.z3z	6	33	Small	Small	Small	Yes
CHRONOMETER_TRA.z3z	6	33				
CHRONOMETER_BOOL_TRA.z3z	6	37				
CHRONOMETER_SEQ_TRA.z3z	6	37				
ALARM.z3z	19	45	3775163	3810301	4721454	Yes
ALARM_TRA.z3z	19	45				
ALARM_BOOL_TRA.z3z	19	53				
ALARM_SEQ_TRA.z3z	19	53				

5 Related Work and Conclusions

The notion of translation validation was introduced in [15] by A. Pnueli et al. to verify the code generator of SIGNAL. In this work, the authors define a language of symbolic models to represent both the source and target programs called *Synchronous Transition Systems (STS)*. A STS is a set of logic formulas which describe the functional and temporal constraints of the whole SIGNAL program and its generated C code. Then they use BDD representations to implement the symbolic models STSs, and their proof method uses a SAT-solver to reason on the signals and clock constraints of STSs. It amounts to the mapping for selected states, consisting of the values of input-output-memory variables, for the source and the target code. The drawback of this approach is that in some cases, the code generator eliminates the use of a local register variable in the generated code and then, the mapping cannot be established. Additionally, for a large SIGNAL program, the logic formula is asked to SAT-solver to solve is very large that makes some inefficiency. Another related work is the approach of J. C. Peralta et al. [13] in a similar approach as the work of A. Pnueli et al. In particular, they translate both the SIGNAL (multi-clocked) specifications given in SIGNAL language and its generated code C/C++ or Java simulator into LTSs. Then, an appropriate pre-order test on both LTSs can be interpreted as a refinement between a generated code implementation and its source SIGNAL specification. The refinement they propose is a bisimulation relation and they use the existing tools to generate the greatest bisimulation relation for the source SIGNAL specification and the target generated code in C/C++. In case there is no bisimulation relation, counterexamples are generated automatically. However, this approach has not been fully automated.

This paper presents the correctness proof of the transformations, optimizations of the multi-clocked synchronous programming language compiler and applies this approach to the highly industrial synchronous data-flow language SIGNAL's compiler. We are interested in proving that abstract clocks and clock relations semantics of source programs are preserved during the compilation phases of the compiler. The desired behaviors of a given source program and its compiled program are represented as PDSs over the finite field of integers modulo $p = 3$. A refinement relation between the source program and its compiled form is used to express the preservation. A proof by simulation is presented to establish the refinement relation. Each compilation stage is followed by our refinement verification process to provide formal guarantees as strong as those provided by a formally verified compiler. If the compilation task from the source program to the compiled form applies without compilation errors, and the compiled form refines the source program, then the compiled form is produced as output else the compiler terminates with an error.

We have implemented and integrated our verification process within the Polychrony toolset by extending the functionality of the existing model checker SIGALI to prove the correctness of the front-end and optimizations phases of the optimizing SIGNAL compiler.

As future work, given a synchronous data-flow program and the corresponding generated C/C++ code, we would like to formally verify that the generated code correctly implements the source program. As we have shown, the verification process can be decomposed into several stages as the decomposition of the compilation task, thanks to the transitive property of symbolic simulation. Thus we only need to prove that there exists a symbolic simulation for the generated C/C++ code and the optimized final program given that the optimized final program refines the source program. In order to do that, we could first translate the asynchronous C/C++ code into the synchronous language SIGNAL. One of the methods is to represent C/C++ code in the Static Single Assignment (SSA) intermediate form and then translate the SSA intermediate form into SIGNAL [4]. The rest of work is the same as the verification process we have presented in this paper.

References

1. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
2. Le Borgne, M., Benveniste, A., Le Guernic, P.: Dynamical systems over Galois fields and control problems. In: Proceedings of 33th IEEE on Decision and Control, vol. 3, pp. 1505–1509 (1991)
3. Besnard, L., Gautier, T., Le Guernic, P., Talpin, J.-P.: Compilation of polychronous data flow equations. In: Synthesis of Embedded Software. Springer (2010)
4. Besnard, L., Gautier, T., Moy, M., Talpin, J.-P., Johnson, K., Maraninchi, F.: Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In: Proceedings of the 9th Workshop on Automated Verification of Critical Systems, AVOCS (2009)

5. Dutertre, B., Le Borgne, M., Marchand, H.: SIGALI: un système de calcul formel pour la vérification de programmes SIGNAL. Manuel d'utilisation. Note technique, non publiée (December 1998)
6. Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
7. Gamatie, A.: Designing embedded systems with the SIGNAL programming: Synchronous, Reactive Specification. Springer, New York (2009) ISBN 978-1-4419-0940-4
8. Le Guernic, P., Talpin, J.-P., Le Lann, J.-C.: Polychrony for system design. Journal for Circuits, Systems and Computers 12(3), 261–304 (2003)
9. Halbwachs, N.: A synchronous language at work: the story of LUSTRE. In: 3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2005) (July 2005)
10. Kouchnarenko, O., Pinchinat, S.: Intensional approaches for symbolic methods. Electronic Notes in Theoretical Computer Science (August 1998)
11. Marchand, H., Rutten, H., Le Borgne, E., Samaan, M.: Formal verification of SIGNAL programs: Application to a power transformer station controller. Science of Computer Programming 41(1), 85–104 (2001)
12. Polychrony Toolset, http://www.irisa.fr/espresso/Polychrony/
13. Peralta, J.C., Gautier, T., Besnard, L., Le Guernic, P.: LTSs for translation validation of (multi-clocked) SIGNAL specifications. In: 8th IEEE/ACM International Conference on Formal Method and Models for Codesign, MEMOCODE (2010)
14. Pinchinat, S., Marchand, H., Le Borgne, M.: Symbolic abstractions of automata and their application to the supervisory control problem. In: INRIA Technical Reports No 1279, pp. 1–29 (November 1999)
15. Pnueli, A., Shtrichman, O., Siegel, M.D.: Translation validation: From SIGNAL to C. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 231–255. Springer, Heidelberg (1999)
16. Milner, R.: Operational and algebraic semantics of concurrent processes. Research Report ECS-LFCS-88-46, Lab. for Foundations of Computer Science, Edinburgh (February 1988)
17. Van Glabbeek, R.J.: The Linear Time-Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves (Extended Abstract). In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
18. VeriSync Project, http://www.irit.fr/Verisync/

Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples

Herbert Rocha, Raimundo Barreto, Lucas Cordeiro, and Arilo Dias Neto

Federal University of Amazonas
Av. General Rodrigo Octávio Jordão Ramos, 3000
Campus, Coroado I - Manaus/Amazonas
{herberthb12,lucasccordeiro}@gmail.com,
{rbarreto,arilo}@dcc.ufam.edu.br
http://portal.ufam.edu.br

Abstract. One of the main challenges in software development is to ensure the correctness and reliability of software systems. In this sense, a system failure or malfunction can result in a catastrophe especially in critical embedded systems. In the context of software verification, bounded model checkers (BMCs) have already been applied to discover subtle errors in real projects. When a model checker finds an error, it produces a counter-example. On one hand, the value of counter-examples to debug software systems is widely recognized in the state-of-the-practice. On the other hand, model checkers often produce counter-examples that are either too large or difficult to be understood mainly because of the software size and the values chosen by the respective solver. This paper proposes a method with the purpose of automating the collection and manipulation of counter-examples in order to generate new instantiated code to reproduce the identified error. The proposed method may be seen as a complementary technique for the verification performed by state-of-the-art BMC tools. In particular, we used the ESBMC model checker to show the effectiveness of the proposed method over publicly available benchmarks and, additionally, a comparison with the tool Frama-C.

1 Introduction

Building complex software systems has been a great challenge to software engineers. This situation can become worse when such software system belongs to a critical embedded system (e.g., aeronautics, space, automotive, health applications) that has to be formally verified to identify errors that may result in failures during the software execution. Thus, verification techniques and software testing are indispensable items for high quality software development.

In the last few years, we can observe a trend towards the application of formal verification techniques to the implementation level. Bounded model checking (BMC) is going to this direction since it has been applied to reason about low-level ANSI-C programs, usually checking safety and/or liveness properties,

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 128–142, 2012.

considering single- and multi-threaded applications [5,6]. BMCs have gained popularity due to their ability to handle the full semantics of actual programming languages, and to support the verification of a rich set of properties such as shared variables and locks, arithmetic under- and overflow, pointer safety, array bounds, deadlocks, and fixed-point arithmetic [5].

This paper proposes a method called EZProofC that aims to apply a software bounded model checker, in this case ESBMC (*Efficient SMT-Based Context-Bounded Model Checker*), with the purpose of verifying critical parts of a software written in the C programming language and, additionally, collecting data to show the evidence that failures might happen. ESBMC is a state-of-the-art symbolic context bounded model checker, which performs comparable to other off-the-shelf software model checkers (e.g., CBMC, SATABS) [5]. The motivation of this work is that data collected by verification tools is usually not trivial to be understood, mainly due to the amount of variables and values involved in the counter-example as well as the lack of a standard output to represent the counter-example. The proposed method uses the data provided by counter-examples to generate new instantiated code to reproduce the identified error. In this paper, the instantiated code is a particular instance of the code with the variable values provided by the BMC, which are enough to reproduce the error. This work thus proposes a method where developers can confirm the results provided by the bounded model checker, and additionally, alleviates the process of analyzing large counter-examples, as well as counter-examples that do not characterize an error (i.e., a spurious counter-examples). We adopted the C programming language since it is the standard language to implement several kinds of software including performance-critical software [11]. However, our techniques can also be applied to other programming languages like C++ and Java.

We show the effectiveness of the proposed method over publicly available benchmarks and, additionally, a comparison with the tool Frama-C [4]. Our experimental results show that EZProofC is able to automatically reproduce all failures found in the benchmarks by the adopted BMC tools through the instantiation of the code. Additionally, EZProofC shows a great advantage in comparison to Frama-C since we do not need to write specifications (i.e., pre- and post-conditions) in the source code. We advocate that automating the data collection process we may disseminate the application of formal methods, help developers not very familiar with this subject, and consequently help them to verify more complex C programs.

2 Context-Bounded Model Checking with ESBMC

Model checking has been used successfully to verify actual software (as opposed to abstract system designs) [3,5,12], including multi-threaded applications written in low-level programming languages such as ANSI-C [5]. In context-bounded model checking, the state spaces of such applications are bounded by limiting the size of the program's data structures (e.g., arrays) as well as the number of loop iterations and context switches between the different threads that are explored

by the model checker. In symbolic model checking, a set of verification conditions (VCs) is derived from the (bounded) system, which are then solved using a Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) solver.

ESBMC is a symbolic context-bounded model checker based on SMT solvers, which allows the verification of single- and multi-threaded software with shared variables and locks [5]. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic. It can reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, the program to be analyzed is modeled as a state transition system $M = (S, R, S_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $S_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. ESBMC identifies each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a safety property ϕ, a context bound C and a bound k, ESBMC builds a *reachability tree* (RT) that represents the program unfolding for C, k and ϕ. ESBMC then derives a *verification condition* (VC) ψ_k^π for each given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counter-example of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

where function I characterizes the set of initial states of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which ESBMC can extract the values of the program variables to construct a counter-example.

A counter-example is a trace that shows that a given property does not hold in the model [1]. Counter-examples allow the user: (i) to analyze the failure; (ii) to understand the root of the error; and (iii) to correct either the specification or the model, in this case, from the property and the program that has been analyzed respectively. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use it to check all paths.

3 EZProofC Method

This section describes the main steps of the EZProofC method[1] which aims to explore the counter-examples generated by the ESBMC model checker, in such a way that it can generate a new instantiated code to reproduce the errors. It is important to emphasize here that we could adopt any BMC tool.

Figure 1 shows an overview of our proposed method. The EZProofC method consists of the following steps: (i) code preprocessing; (ii) model checking with ESBMC; (iii) generation of a new instantiated code; and (iv) code execution and confirmation of defects.

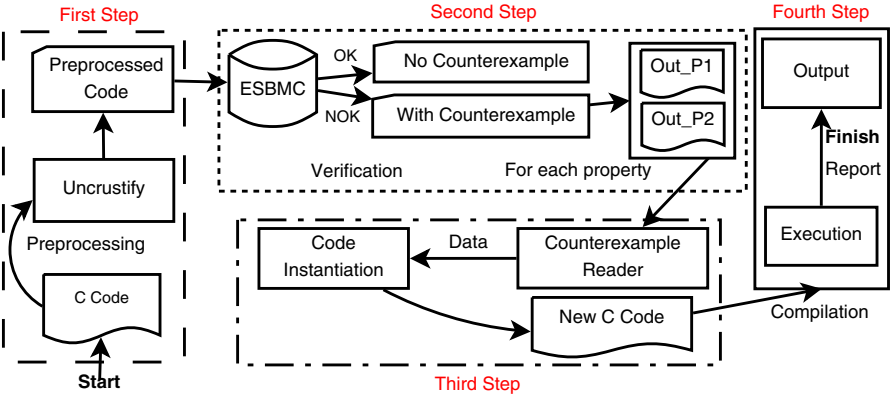


Fig. 1. Flow structure of the proposed method

To explain the main steps of our proposed method, we use the application Sendmail[2], in particular, the code `tTflag_arr_two_loops_bad.c` extracted from the Verisec[3] benchmark suite, which is the the standard Unix mail (SMTP) server. This code has 64 lines of code and aims to parse a string of digits into two signed integers.

3.1 First Step: Code Preprocessing

In the first step, the analyzed code is preprocessed using the tool *UNCRUSTIFY*[4] that will preprocess the code, as show in Figure 2, to define a standard formatting to improve the presentation of the formatting items such as: indentation, block delimiters, one command per line, delineation of structures, and other formatting aspects. This preprocessing step allows a better identification of structures contained in the code, facilitating its handling and making it easier to implement the next steps. It is important to note that Figure 2 presents just a fragment from the original code.

[1] Available at `https://sites.google.com/site/ezproofc/`
[2] Available at `http://www.sendmail.org`
[3] Available at `http://se.cs.toronto.edu/index.php/Verisec_Suite`
[4] Available at `http://uncrustify.sourceforge.net`

```
1  #define INSIZE 14
2  int main (void){
3  unsigned char in[INSIZE+1];
4  unsigned char c;
5  int i, j;
6  int idx_in = 0;
7  ...
8  /*accumulate last(int) from in (char[])*/
9  c = in[idx_in];
10 if (c == '-')
11 {
12  i=0;
13  idx_in++;
14  c = in[idx_in];
15  while (('0' <= c) && (c <= '9'))
16  {
17    j = c - '0';
18    i = i * 10 + j;
19    idx_in++;
20    c = in[idx_in];
21   }
22 }
23 }
```

Fig. 2. C code fragment already pre-processed

3.2 Second Step: Model Checking with ESBMC

In the second step, we use the ESBMC to verify the properties that are violated in the code. ESBMC divides the verification in two levels: In the first level, ESBMC determines which properties might be violated by means of preliminary static analysis (using abstract interpretation), for determining program locations that potentially contain an error (these properties are called claims). It is worth to note that claims are automatically generated by ESBMC. Due to the imprecision of the static analysis, there is the need to go the second level, that is, ESBMC has to confirm that these claims are indeed genuine errors by using a more complete and accurate verification technique (it is important to emphasize that during the verification, ESBMC adopts the program slicing technique [14]).

The verification result may be classified in two ways: the code was checked and there is no counter-example (i.e., the property was verified but no error has been found up to the given bound k) and the code was verified and there is a counter-example (i.e., a property violation has been found, as shown in Figure 3) which presents the violation of the property "`idx_in<15`" identified in the code fragment shown in Figure 2 (line 20). Additionally, for the verification process, ESBMC has an Eclipse plug-in, [5] which allows the user to locate the variable in the counter-example directly in the analyzed code. To explain clearly each proposed step, we decided to analyze one specific `claim` as shown in line 20 of Figure 2.

The property "`idx_in<15`" has been violated due to the fact that in the array index `in` the variable `idx_in` exceeds the upper bound of the array `in` as defined in line 3 (`in[INSIZE+1]`) of Figure 2, where `INSIZE` is defined with the value 14.

[5] Available at `http://www.eclipse.org`

```
Counterexample:

(.....)
State 55 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 9 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::c=45 (00101101)

State 58 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 13 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::idx_in=9 (00000000000000000000000000001001)

State 59 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 14 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::c=48 (00110000)

State 96 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 17 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::j=3 (00000000000000000000000000000011)

State 97 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 18 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::i=33 (00000000000000000000000000100001)

State 98 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 19 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::idx_in=15 (00000000000000000000000000001111)

State 93 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main thread 0
------------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::c=51 (00110011)

Violated property:
 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main
 array `in' upper bound
 idx_in < 15

VERIFICATION FAILED
```

Fig. 3. Counter-example

As the loop in line 15 does not control the value of the variable `idx_in`, in state 98 this variable receives a value greater than the upper bound of the array `in`, which thus causes the UPPER BOUND violation.

3.3 Third Step: Code Instantiation

The third step is divided into two phases: analysis of counter-examples produced in step 2 and generation of a new instantiated C code. Algorithm 1 details the method to run both phases. The runtime complexity of this algorithm is $O(n + m)$, where n is the size of the analyzed C code and m is the size of the counter-example. The inputs of this algorithm are the analyzed code (`Code`) and the counter-example (`CE_Out`). Initially, the counter-example (`CE_Out`) is analyzed to collect several pieces of information, such as: (1) the variables involved in the property violation; (2) the line number where values are assigned to variables; and (3) the specific value for each variable. This information is obtained by the function `GetValuesCEER` (line 1 of the Algorithm 1) through regular expressions applied to the counter-example file. This function returns a set that contains data about the variables found in the counter-example (e.g., `Var{vline = 9,var = c,vvalue = 45}`), the violated property (`P`) and the line number where the property has been violated (`line_p`).

In this way, the analyzed code is read (starting from line 8), as well as the counter-example. If the line number of the variable identified in the counter-example is equal to the line number of the analyzed code we can generate a new

```
   Input: Code, CE_Out
   Output: New instantiated code
   // first phase
 1 Var,P,line_p ← GetValuesCEER(CE_Out);
 2 SCE ← {Var{vline,var,vvalue},P,line_p};
 3 size ← GetTotalLineCE(SCE[Var[]]);
 4 Lines, tline ← GetValuesCode(Code);
 5 SCode ← {Lines{ }, tline};
 6 UPCASE ← {Set of specific cases for counter-example data collection};
 7 i, k ← 1;
   // second phase
 8 while i ≤ SCode[tline] do
 9     if i == SCE[Var[vline[k]]] AND k ≤ size then
10         if SCE[P] OR SCE[Var[vvalue[k]]] ∈ UPCASE then
11             New_Line ← StartTrigger(SCE[P], SCE[Var[vvalue[k]]]);
12             WriteLineCode(New_Line); k ← k + 1;
13         end
14         else
15             New_Line ← "SCE[Var[var[k]]] = SCE[Var[vvalue[k]]]";
16             WriteLineCode(New_Line); k ← k + 1;
17         end
18     end
19     else
20         WriteLineCode(SCode[Lines[i]]); i ← i + 1;
21     end
22 end
```

Algorithm 1: Counterexample2NewCode

line of code; where the identified variable receives the value abstracted from the counter-example (e.g., the following values of the variables gathered from the counter-example `line = 9, var =c` and `value = 45` result that the variable New Line (in line 15) receives the text c = 45, which thus generates a new line). Importantly, the instantiation of the variables in the new code is strictly executed according to the sequence in the counter-example. For instance, if the same variable in the counter-example is mentioned multiple times in the same line (for example, in loops), only the last value found in the counter-example will be assigned to the variable in the instantiated code.

Improving the counter-example data collection, the proposed method may require a separate approach for some specific cases, where it is applied to the verification step of the EZProofC method (see Section 3.2) or triggered by the analysis of the counter-example. Line 10 of the Algorithm 1 checks whether the property or a variable in the counter-example is in a set of specific cases already predefined (line 6 variable `UPCASE`). Thus, if there is some specific case in the counter-example that has been identified, the proper approach is applied by the adoption of the function `StartTrigger` in the line 11, as following:

(i) When the violation of a property is identified and there is not enough information about the counter-example, it is necessary to use in the verification step with ESBMC, particularly in smaller code, the option `--no-slice`

which does not remove unused equations of the program for generating the counter-example. Another way to diversify the values of variables, and hence the result in the counter-example, is to apply non-deterministic values to them, e.g. `a[0]=nondet_int();`

(ii) In some specific cases, the violation of the property UPPER BOUND can generate a counter-example without the data about the upper bound of the array. In this case, the method firstly identifies the array name and, through the analysis of the code, it can collect the upper bound. This procedure is accomplished by two elements: the first is the function `NUM_OF(arr)` to get the array size; and the second element is an `assert` that will contain the result of the function `NUM_OF(arr)` and the index value of the array that was identified in the counter-example, thereby the structure of the assert will be the following `assert((N)<=NUM_OF(arr)-1)`, where `N` is the index value, that will be adopted to validate the bound of the array;

(iii) Considering dynamic memory allocation violations, the proposed method has to analyze: (1) if the pointer is referencing to the correct object; (2) if the pointer points to an invalid object; (3) if the object considered is a dynamic object; and (4) the argument of a `free` function call if in the deallocation procedure is still a valid pointer value. The aim of this analysis is to obtain a right assertion about the property identified.

The second phase of this third step from the EZProofC method aims to generate a new instantiated code. The method only makes a copy of the original code (in line 20 of the Algorithm 1), and replaces variables assignments using the specific values identified in phase one (in line 12 or 16). In the case of properties such as `UPPER BOUND` or `LOWER BOUND`, the proposed method includes assertions in the instantiated code to reproduce the error, as mentioned before in line 11 about the triggers in the analysis of the counter-example. Such assertions contain data from the property identified in the counter-example. The final result of this step is an instantiated C code with the values of variables that are extracted from the counter-example, as shown in Figure 4. It is worth noting that in the counter-example (see Figure 3), the property violated was `UPPER BOUND`, and the data was "`idx_in<15`". In this case, in line 20 of Figure 4, the proposed method has included an assertion in order to reproduce the error.

In particular, in this example it is obvious that the assertion will fail. This is because the previous instruction assigns exactly a value that contradicts the assertion. However, it is worth observing that this assignment comes directly from the counter-example, implying that there is a situation where this assignment happens in one of the execution paths.

3.4 Fourth Step: Code Execution and Confirmation of Errors

In the third step of this method, we generate one instantiated program for each property violated. In this fourth step, each instantiated code is compiled and executed. The result of the execution demonstrates the error (`Line:20:main: Assertion & 'idx_in<15' failed. Aborted`) pointed out by the counterexample.

```
#define INSIZE 14
int main (void){
unsigned char in[INSIZE+1];
unsigned char c;
int i, j;
int idx_in = 0;
...
/*accumulate last(int) from in (char[])*/
c =45 ; //<- by EZProofC
if (c == '-')
{
 i =0 ;
 idx_in = 9 ; //<- by EZProofC
 c =48 ; //<- by EZProofC
 while (('0' <= c) && (c <= '9'))
 {
    j =3 ; //<- by EZProofC
    i =33 ; //<- by EZProofC
    idx_in = 15 ; //<- by EZProofC
    assert(idx_in <15); //<- by EZProofC
    c =51 ; //<- by EZProofC
 }
}
}
```

Fig. 4. C code already instantiated

4 Experimental Results

This section describes the planning, design, execution, and the analysis of the results of an empirical study conducted with the purpose of evaluating the proposed method when applied to the verification of standard ANSI-C benchmarks and, additionally, a comparison with the tool Frama-C[6] [4] version Boron-20100401. Frama-C is a suite of tools dedicated to the analysis of software written in C. Frama-C makes it possible to observe sets of possible values for the variables of the program at each point of the execution. Frama-C also allows verifying that the source code satisfies a provided formal specification. The specifications can be written in a dedicated language, in this case, ANSI/ISO C Specification Language (ACSL).

The experiments were conducted on an Intel Core 2 Duo CPU, 2Ghz, 3GB RAM with Linux OS. The proposed method was implemented in a tool called EZProofC using the ESBMC v1.16 model checker.

4.1 Planning and Design the Experiments

The goal of this empirical evaluation is to analyze the impact of the proposed method with the purpose of confirming the properties reported by the model checker as possible errors in the code. This confirmation is based on the number of properties (*claims*) reported by the model checker, which should be confirmed by the proposed method.

[6] http://frama-c.com/

In order to evaluate the proposed method, we considered 211 ANSI-C programs from six different benchmarks selected with the aim to evaluate the capacity and performance of methods and techniques in the identification of errors. Moreover, such ANSI-C programs from these standard benchmark suites represent real implementations. The adopted benchmarks were: (i) EUREKA[7] which contains programs that allow us to assess the scalability of the model checking tools on problems of increasing complexity. It is worth observing that some of the programs represent more than one execution, with different input data. For instance, the program `bubble_sort1_13.c` represents 13 instances (from 1 to 13) of the program `bubble_sort.c`. The program `prim4_8.c` represents 5 instances (from 4 to 8) of the program `prim.c`; (ii) SNU[8] which contains C programs used for worst-case execution time analysis, where such programs are mostly of numeric analysis and DSP (Digital Signal Processing) algorithms; (iii) WCET[9] which, in the same way as SNU, contains programs used for worst-case execution time analysis; (iv) NEC[10] which contains C programs that allow us to check error-detection easily since they provide ANSI-C programs with and without known errors; (v) Siemens (SIR[11]) which is a test suite for lexical analyzer, pattern matching and (vi) some ANSI-C programs taken from the CBMC (C bounded model checker) tutorial [12].

During this empirical evaluation, each program of the benchmark was executed using 3 methods: (1) Application of the EZProofC method (see Section 3), i.e., code preprocessing, identification of claims, verification, analysis of counterexamples, and code instantiation; (2) Application of the tool Frama-C with the option `-val`, which means that the *value analysis* plug-in is called in such a way that it computes automatically variation domains for the variables of the program. This plug-in is used to infer absence/presence of runtime errors; and (3) Application of the tool Frama-C with the plug-in `Jessie`, which is a plug-in that allows deductive verification of C programs annotated with ACSL [2]. The *verification conditions (VC)* are verified by the `Z3` theorem prover [13], which is the same standard theorem prover used by the ESBMC model checker. In this way the tool Frama-C was executed as: `frama-c -jessie -jessie-atp=z3 <file.c>`, where `<file.c>` is the C code that will be verified.

4.2 Experiment's Execution and Results Analysis

After executing the benchmarks, we obtained the results shown in Table 1, where each column of this table means: (1) the application identification (ID), (2) the C program name and, additionally, in some cases, the range of instances, e.g.,

[7] http://www.ai-lab.it/eureka/bmc.html
[8] http://www.cprover.org/goto-cc/examples/snu.html
[9] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[10] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
[11] http://sir.unl.edu/portal/index.html
[12] http://www.cprover.org/cbmc/doc/manual.pdf
[13] http://research.microsoft.com/en-us/um/redmond/projects/z3/

file1_13.c, meaning that there are 13 instances, from 1 to 13. In Table 1 the programs from 1 to 16 come from the EUREKA benchmark, from 17 to 19 come from the CBMC tutorial, program 20 comes from the NEC benchmark, from 21 to 22 come from the SNU benchmark, program 23 comes from WCET benchmark, and program 24 comes from SIR benchmark; (3) the lines of code - LOC (#L); (4) the amount of identified *warnings* (#W) and the execution time (TW) of the Frama-C with the *value analysis plug-in*; (5) the total number of properties (or `claims`) that may be violated (#P), the execution time of the properties identification spent by ESBMC and EZProofC (TC), the execution time of the verification of all properties spent by ESBMC (TV), total number of properties that have been violated and reproduced using the EZProofC method (#V), and the number of lines in the counter-examples (CE); and (6) the number of properties found in common (Same Claims & Warnings) between the EZProofC (claims) and the Frama-C (warnings).

It is important to note that for programs with several instances, the number of violated properties presented is that of the highest instance value. Additionally, in case of programs with more than one instance, the number of lines in the counter-examples (#CE) and properties found in common (Same Claims & Warnings) is respectively the largest counter-example found and the largest number of properties found in common. The results of the

Table 1. Details related to the execution time of the benchmarks

#	Module	#L	Frama-C		EZProofC/ESBMC					Same Claims
			#W	TW	#P	TC	TV	#V	CE	& Warnings
1	bf5_20.c	49	6	<1s	33	<1s	<60s	0	-	0
2	bubble_sort1_13.c	51	2	<1s	25	<1s	<15s	0	-	0
3	fibonacci1_13.c	25	1	<1s	1	<1s	<1s	0	-	0
4	init_sel_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
5	minmax1_13.c	19	6	<1s	9	<1s	<3s	0	-	0
6	minmax_unsafe1_13.c	19	6	<1s	9	<1s	<4s	1	16	0
7	n_k_gray_codes1_13.c	45	36	<1s	22	<1s	<120s	0	-	11
8	no_init_bubble_sort_safe1_13.c	25	2	<1s	14	<1s	<7s	1	32	1
9	no_init_sel_sort1_13.c	41	5	<1s	25	<1s	<15s	12	144	3
10	no_init_sel_sort_safe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
11	no_init_sel_sort_unsafe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
12	prim4_8.c	79	12	<1s	30	<1s	<60s	0	-	3
13	selection_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
14	strcmp1_13.c	15	4	<1s	6	<1s	≈14400s	3	80	0
15	sum1_13.c	21	1	<1s	1	<1s	<1s	1	48	0
16	sum_array1_13.c	11	1	<1s	7	<1s	<3s	1	8	0
17	assert_unsafy.c	15	4	<1s	1	<1s	<1s	1	24	0
18	bound_array.c	16	2	<1s	10	<1s	<10s	1	30	1
19	division_by_zero.c	32	3	<1s	1	<1s	<1s	1	24	1
20	ex26.c	29	4	<1s	8	<1s	≈420s	2	1236	1
21	crc_det.c	125	1	<1s	15	<1s	≈840s	0	-	1
22	select_det.c	122	3	<1s	39	<1s	≈14400s	3	40	1
23	cnt_nondet.c	139	0	<1s	16	<1s	<1s	0	-	0
24	Siemens_print_tokens2.c	508	90	<1s	51	<1s	≈18000s	1	3344	34

application of the proposed method, as well as the EZProofC tool are available at https://sites.google.com/site/ezproofc/.

As shown in Table 1, the EZProofC method is *scalable* to any code and counter-example size, since the complexity of the proposed method algorithm is $O(n + m)$. The execution time of EZProofC is thus linear, even when considering different code sizes, as we can see in the experiments' execution time.

One could argue that the selected benchmarks may not represent well all the possible scenarios for applying the proposed method, mainly when taking into account the programs size in terms of LOCs. However, as an example consider the experiment with the program 20 from Table 1, which has only 29 LOCs but it was the one that produced some of the largest counter-examples, in this case 1236 lines. Note further that this counter-example has a trace that shows all the variables, as well as the assignments included in a specific execution (i.e., including loops) that will result in the violation of the property that has been identified by ESBMC (i.e., unwind of a specific execution of the program). The drawback of the EZProofC tool is that it relies on the scalability of the adopted model checker, since it depends only on it to generate the counter-examples. Apart from that, the proposed method is able to scale to large sizes of counter-examples, in this case, from 8 up to 3344 lines. However, we believe that the limiting factor on the size of the counter-example is far beyond this.

Analyzing the Frama-C tool with the *value analysis plug-in*, it is important to emphasize that the results about *warnings* (in the column #W) are very effective, providing the user with a good support to explore the code that has been analyzed. However, such *warnings* were not only about safety properties, but involved analysis of the structures of the code (e.g., return of functions). This partly explains why the number of properties between the EZProofC (claims) and the Frama-C (warnings) (column Same Claims & Warnings of Table 1) are rather different.

The Frama-C tool also allows the use of other plug-ins, for instance, the plug-in `Jessie`, which aims to perform deductive verification of C programs not using, in this case, static analysis. The C program does not need to be complete nor annotated to be analyzed with the Jessie plug-in [10]. However, in the experiment conducted, Jessie plug-in did not find any property violation, i.e., no error was found, even though Frama-C pointed out several *warnings*. Jessie plug-in also allows to prove that C functions satisfy their specification as expressed in ANSI/ISO C Specification Language (ACSL). We understand that the verification of Frama-C could be improved by writing such specifications on the analyzed C code. However, the inclusion of such specifications may be hard and error-prone, especially for legacy code. Therefore, if we compare the use of Frama-C/Jessie and the EZProofC, we argue that a great advantage of EZProofC is not requiring such auxiliary specifications. EZProofC is a completely automatic method that does not need to write specifications, and neither preconditions and postconditions. Additionally, in the case of the Frama-C, the user has to act explicitly to reproduce the error using the computed values.

In these experiments some situations need to be pointed out about the application of the EZProofC method.

- In program 20 from Table 1, EZProofC identified properties of safety pointers and dynamic memory allocation (`POINTER_OFFSET` and `SAME-OBJECT`). The property identified was `UPPER BOUND` and the data was `!(2 * y + POINTER_OFFSET(x) >= 200) || !(SAME-OBJECT (x, &b[0]))`. However, after handling all information (see Section 3.3), this resulted only in the following assertive `(2*y + x >= 200) || (x != b)`.
- Program 24 is considered the golden version code (i.e., the supposed correct version). Taking into account that this code is very large, and requires a significant amount of memory, the verification was performed in a function-by-function basis. Particularly, we checked the function `get_token`. The error identified in this code is the `UPPER BOUND` violation of `array buffer`, which is declared with the upper bound of 80. However, based on the proof of the error, it is noticed that the index of this array, the variable `i`, exceeded the upper bound, causing the violation of property $i < 81$, in the same way as identified in the work of Cordeiro *et.al.* [6].

We have shown that the manipulation of the counter-example is not always a trivial task. During the experiments, we obtained relatively large counter-examples (e.g., 3344 lines). However, the application of our proposed method decreases substantially the complexity of this task, i.e., the EZProofC solves the problem in less than 1s (without the verification step with the model checker), to manipulate a large amount of data, variables and their values. It is important to emphasize the need for verifying each property (*claims*) identified in the analyzed code. This is because these properties do not necessarily correspond to errors, but these are only potential failures. This is the reason by which the number of properties identified in Table 1 is greater than or equal to the number of errors reproduced.

5 Related Work

In the technical literature, there are several tools and methods for analysis of counter-examples and debugging code for error-proof. Many studies have addressed this problem (e.g. [4], [7], [9], [13]), that aim to find the root cause of a failure in the model, and propose automated means of extracting more information about the model, facilitating the debugging process.

Ji *et al.* [9] present a software debugger used for finding errors in C programs. In the same way, EZProofC aims to demonstrate errors found by BMCs. The difference is that our technique tests the system exhaustively for verifying that a given property is part of the model. Additionally, BMCs run the code symbolically, that is, they do not test programs with fixed entry values, but create a mathematical model of the program [1]. Debuggers, however only evaluate execution paths that were defined according to the input variables. Thus, a debugger will not exhaustively test the state space of the analyzed code.

Taghdiri and Jackson [13] propose a counter-example guided refinement of an abstraction to check programs written in any programming language that

supports procedure declarations and can be translated to logical constraints. In the same way as our work, they propose a "validity check", where the validity of each behavior in the counter-example is checked in the original program. They use a SAT solver, and our work uses ESBMC that adopts an SMT solver. Nevertheless, if the counter-example is invalid they propose to adopt a "specification inference", that is, the specification is not provided by the user but automatically inferred from the code. In our opinion, the drawback of such method is the limited applicability since they target to solve only structural properties, i.e., properties that constrain the configuration of the heap after the execution of a procedure.

Astrée[14] [7] is a completely automatic analyzer that aims to prove the absence of run time errors (RTE) in C programs. The design of Astrée is based on abstract interpretation, which is a formal theory of discrete approximation. Astrée analyzes structured C programs, with complex memory usages, but without dynamic memory allocation and recursion. It also excludes union types, unbounded recursive functions calls, and the use of C libraries. In the same way as Astrée, the EZProofC aims to produce a correctness proof for complex software without any false alarm (or spurious counter-examples). However, EZProofC differs from Astrée in the sense that the proposed verification is made by a bounded model checker which provides support for structures not supported by Astrée.

6 Conclusions and Future Work

The main purpose of this paper is to help developers not familiar with formal verification techniques to use a model checker tool to find failures in the software and to verify that such errors may happen. We described a method called EZProofC that aims to contribute as a complementary technique to the verification performed by BMCs. Specifically, we have used the ESBMC tool, which is a state-of-the-art symbolic context bounded model checker. Basically, our method proposes to automate the gathering and manipulation of the counter-example generated by ESBMC in order to reproduce the identified error.

The experimental results have shown to be very effective over publicly available benchmarks. In this case, we could reproduce all failures encountered by the adopted BMC tool. On the one hand, we demonstrate that EZProofC has some advantages, when compared to Frama-C, mainly because EZProofC can automatically reproduce the identified property violation, through the generation of a instantiated code. On the other hand, the Frama-C requires the user to act explicitly to demonstrate the error using computed values.

We noticed that due to the state space explosion problem, the user may ask to the BMC to adopt simplifications in the model (e.g. function-by-function verification). In some situations, this can lead to spurious results, i.e., a counter-example may not truly characterize an error. In this way, we want to investigate the inclusion of additional data during the phase of new instantiated code generation in order to demonstrate the verification (with such simplifications). For

[14] http://www.astree.ens.fr/

instance, in the case of verifying a program function-by-function, we need to include the values of variables that are dependent on other functions other than the function being verified. Additionally, we intend to extend our experiments to evaluate the usability of the proposed method. We also plan to adapt the proposed method to use other model checkers (Blast [3] and Java PathFinder [8]) that rely on other abstraction techniques. We think that the adjustment will be in most part on regular expressions, which was the way we implemented data gathering and new code generation.

Acknowledgement. The authors acknowledge the support granted by FAPESP process 08/57870-9, CAPES process BEX-3586/10-3, and by CNPq processes 575696/2008-7, and 573963/2008-8.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. In: CEA LIST and INRIA (2009), http://frama-c.cea.fr/acsl.html
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transf. (STTT) 9, 505–525 (2007)
4. Canet, G., Cuoq, P., Monate, B.: A Value Analysis for C Programs. In: Intl. Conf. on Source Code Analysis and Manipulation (SCAM), pp. 123–124 (2009)
5. Cordeiro, L., Fischer, B.: Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In: Intl. Conf. on Software Engineering (ICSE), pp. 331–340 (2011)
6. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Transactions on Software Engineering (TSE) 99 (2011), http://eprints.ecs.soton.ac.uk/22291/
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
8. Havelund, K.: Java PathFinder, A Translator from Java to Promela. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, p. 152. Springer, Heidelberg (1999)
9. Ji, J.H., Woo, G., Park, H.B., Park, J.S.: Design and Implementation of Retargetable Software Debugger Based on GDB. In: Intl. Conf. on Convergence and Hybrid Information Technology (CHIT), vol. 1, pp. 737–740 (2008)
10. Marché, C., Moy, Y.: Jessie plugin tutorial. In: INRIA (2010), http://frama-c.com/download/jessie-tutorial-Carbon-20101201-beta1.pdf
11. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: Compiler Enforced Temporal Safety for C. SIGPLAN Notes 45, 31–40 (2010)
12. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. Int. J. Softw. Tools Technol. Transf. (STTT) 11, 187–202 (2009)
13. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 144–153 (2004)
14. Tip, F.: A survey of program slicing techniques. Journal Programming Languages 3(3) (1995)

MULE-Based Wireless Sensor Networks: Probabilistic Modeling and Quantitative Analysis

Fatemeh Kazemeyni[1,2], Einar Broch Johnsen[1],
Olaf Owe[1], and Ilangko Balasingham[2]

[1] Department of Informatics, University of Oslo, Norway
[2] The Intervention Center, Oslo University Hospital, Oslo, Norway

Abstract. Wireless sensor networks (WSNs) consist of resource-constrained nodes; especially with respect to power. In most cases, the replacement of a dead node is difficult and costly. It is therefore crucial to minimize the total energy consumption of the network. Since the major consumer of power in WSNs is the data transmission process, we consider nodes which cooperate for data transmission in terms of groups. A group has a leader which collects data from the members and communicates with the outside of the group. We propose and formalize a model for data collection in which mobile entities, called *data MULEs*, are used to move between group leaders and collect data messages using short-range and low-power data transmission. We combine declarative and operational modeling. The declarative model abstractly captures behavior without committing to specific transitions by means of probability distributions, whereas the operational model is given as a concrete transition system in rewriting logic. The probabilistic, declarative model is not used to select transition rules, but to stochastically capture the result of applying rules. Technically, we use probabilistic rewriting logic and embed our models into PMaude, which gives us a simulation engine for the combined models. We perform statistical quantitative analysis based on repeated discrete-event simulations in Maude.

1 Introduction

Formal methods traditionally consider qualitative properties of models such as various correctness properties. However, many communities (additionally) expect quantitative analysis results, which can be difficult to obtain for such models. In contrast, approaches based on probability distributions over possible transitions are able to provide numerical results; for example, the probability of reaching a certain state with a given probability for message loss. *Probabilistic rewrite theories* [17] form a semantic framework for system specification which is capable of specifying both nondeterministic and probabilistic behaviors of systems, extending rewriting logic [21]. Probabilistic rewrite theories can be used instead of traditional rewrite theories to model networks with different probabilistic and nondeterministic behaviors. The execution of models given as probabilistic rewrite theories can be simulated using the Maude rewriting tool [6], which allows tool-supported analysis.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 143–157, 2012.

In this paper, we apply a combination of *operational specifications* of behavior, given as a transition system, with *declarative specifications*, given by probability distributions, using probabilistic rewrite theories. Abstract declarative specifications are used to underspecify behavior when it is difficult to predict the exact behavior of the model in terms of specific transitions, whereas operational specifications are used otherwise. This way, the probability distributions are not associated with the choice of transitions rules, but rather with the outcome of applying transitions. Combining declarative and operational specifications as a means for underspecification can in some cases remove oversimplifying assumptions from the operational model; this makes the resulting specifications more realistic while they can still be analyzed using quantitative techniques. Using Maude to simulate the basic behavior of models given as probabilistic rewrite theories, we apply a statistical quantitative analysis method based on discrete-event simulation, in order to obtain numerical results about the combined model.

The proposed modeling approach is illustrated by a case study in the domain of underwater wireless sensor networks (UWSNs). WSNs consist of small nodes with sensing, computing, and communication devices, which collaboratively monitor and collect data from the environment. Resource limitations in WSNs raise the importance of efficient communication protocols among sensor nodes. Especially, limitations of energy resources need to be considered in order to improve the longevity of the nodes [26]. Data transmission is expensive with respect to power, therefore, the management of communication between nodes is an important factor for network power efficiency. In UWSNs [7], communication uses acoustic data transmission through water. Due to its acoustic nature, transmission costs more power than in usual WSNs, and message loss may occur. One approach to UWSNs is Mobile Ubiquitous LAN Extension (*MULE*) systems [29]. A (data) MULE is a mobile object, such as a vehicle with large and replaceable energy resources. A MULE system consists of a three-tier architecture: (i) *sensor nodes*, in the lower level, which gather data; (ii) mobile agents as MULEs, in the middle level, which move around in the network area and collect nodes' data using single-hop short range transmission; and (iii) access points or *sink* nodes, in the upper level, which receive the data from the MULEs. MULEs move independently from the sensors, and in most cases randomly or following predefined paths. The MULE architecture is an energy efficient solution for data gathering in WSNs that is also scalable and flexible with respect to the network size [3].

This paper develops a probabilistic model that is a combination of declarative and operational models for data collection in a MULE-based WSN, extending a grouping protocol introduced in [16]. In this protocol sensor nodes form groups, using coalitional game theory, in order to save energy in the network. A group has a selected node called *leader* which is responsible for receiving data from the group members and for communication with the outside of the group. To further improve energy efficiency, MULEs gather the data from group leaders. We model MULEs by using a probability distribution of the MULEs' locations in order to abstractly model their movement and the rate of message loss. We combine this declarative specification of MULE-based communication with an

operational model of the grouping protocol in rewriting logic [21], and use the Maude tool [6] to simulate the stochastic behavior of the resulting model. Combining a series of Maude simulations, we obtain numerical insight about the behavior of this protocol. The numerical results show that using the grouping protocol is beneficial to MULE-based WSNs with respect to energy conservation.

Related Work. Protocol validation is mostly done with simulation-based tools, using NS, OMNeT+. Formal analysis techniques are much less explored in the development and analysis of WSNs, but start to appear. Among automata-based techniques, the TinyOS operating system has been modeled as a hybrid automaton [9] and UPPAAL has been used for analyzing the LMAC protocol [11] and the temporal configuration parameters of radio communication [30]. A recent process algebra for active sensor processes includes primitives for, e.g., sensing [8]. Ölveczky and Thorvaldsen show how a rich specification language like Maude is well-suited to model WSNs, using Real-Time Maude to analyze the performance of the OGCD protocol [24].

In this paper, we use probabilistic rewrite theories [17] as the formal modeling language and the Maude tool to develop a grouping protocol for MULE-based WSNs that exhibit probabilistic behavior, building on a protocol proposed in [16], which applies coalitional game theory but does not consider message loss and probabilistic modeling. From the modeling point of view, PRISM [19] is another probabilistic modeling language that comes with probabilistic model-checking and quantitative analysis tools [18]. Some process algebraic approaches to modeling, verification, and analysis of probabilistic models are the PEPA [13] and EMPA [5] frameworks, the Probabilistic KLAIM coordination language [25], and the Stochastic π calculus [27]. PMaude, the probabilistic extension of Maude, is a rewrite-based modeling language. PMaude offers a natural way to describe the structures considered in Stochastic CLS, so from a modeling perspective, it is more suitable for our purpose. In contrast to PRISM, PMaude cannot verify quantitative properties. The VeStA [28] tool, which support both PMaude and PRISM, fails when running as big state spaces as we have in our model. As a solution, we take Maude extended with probabilistic rules, using sampling from given distributions, and add a tailor-made external layer producing quantitative results by repeated probabilistic simulations. Consequently, we do not perform stochastic model checking as VeStA offers, but our analysis can provide some quantitative information as well as diagrams of attribute values during one simulation and the average of the values of different simulations, which are important for understanding and comparing a protocol's efficiency. The Real-Time Maude approach [24] has also been combined with probabilistic model-checking to analyze the LMST protocol [15]. They use VeStA to perform statistical model checking, while in our approach, a probabilistic rewrite theory is used to build the combined declarative and operational model with a simple discrete time model. The PVeStA tool [31] is a client-server-based parallelization of VeStA. The CaVi tool combines simulation in Castalia with probabilistic model-checking [10]. There are some works that follow the same approach as ours, but in different fields. For instance, the authors of [4] use stochastic

abstraction and model checking for the communication system of the airplanes. We work on the higher layers of the network and use rewriting logic for our analysis, in contrast to the BIP toolset that is a component-based framework.

Different aspects of UWSNs have recently been studied. In [7] several research challenges in this area are discussed, while [26] provides an overview of networking protocols for UWSNs. Recent studies on the energy conservation in WSNs are surveyed in [3]. Cluster-based protocols have been studied in some research such as [22], which proposes a cluster-based routing protocol for UWSNs, regardless of the nodes' locations. A well-known work related to energy efficiency of WSNs is LEACH [12], a cluster-based protocol that uses randomized rotation of local cluster-based stations to distribute the energy load among the sensors. MULEs have not only been used in UWSNs, but also for other kinds of WSNs, see, e.g., [14]. We combine a MULE-based architecture and grouping of nodes, in order to increase the energy efficiency of WSNs.

Paper Overview. Section 2 summarizes probabilistic rewrite theories. Section 3 describes the grouping protocol in MULE-based sensor networks, and Section 4 introduces our declarative model of MULE-based communication. Section 5 describes the proposed formal model, while the methods for statistical quantitative analysis are introduced in Section 6. The paper ends with Section 7, containing the conclusions and suggested future work.

2 Probabilistic Rewrite Theories and PMaude

Rewriting logic (RL) extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. A *rewrite theory* is a tuple (Σ, E, L, R) where the signature Σ defines the function symbols, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts. Sorts are specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of suitable sorts defined in the equational logic. The global state configuration is defined as a multiset of these terms. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Formal models defined in rewriting logic [21] are executable in Maude [6]. Maude provides a tool framework that includes tools such as a reachability analyzer, an LTL model checker, and InVa (invariant model checker for infinite state-spaces).

Probabilistic rewrite theories form a general semantic framework for the specification of systems with both nondeterministic and probabilistic behavior [17]. In [17] it is shown that probabilistic rewrite theories represent a *unifying* semantic framework, i.e., that certain mappings exist between several different probabilistic modeling formalisms and probabilistic rewrite theories. This framework is an extension of *rewrite theories* [21], capturing the evolution of a system through a series of *conditional probabilistic rewrite rules* with the syntax

$$t(\vec{x}) \longrightarrow t'(\vec{x}, \vec{y}) \textbf{ if } cond(\vec{x}) \textbf{ with probability } \vec{y} := \pi(\vec{x}), \tag{1}$$

where $\vec{x}$, $\vec{y}$ are sets of variables and $t(\vec{x})$, $t'(\vec{x}, \vec{y})$ are terms in an algebra of fully simplified terms, with respect to a membership equational theory and a collection of structural axioms [21]. Also, $cond(\vec{x})$ is a condition that needs to be met for the rewrite (1) to take place and π is a probability distribution over a set of substitutions for $\vec{y}$, possibly depending on the variables $\vec{x}$ of the term $t(\vec{x})$. Such rules are nondeterministic, as the variables $\vec{y}$ in their right-hand side do not also appear in the left-hand side. The notation := in (1) can be understood as a standard `let` expression in functional languages, allowing us to specify the probability distribution which the variables $\vec{y}$ follow.

PMAUDE is introduced in [1] as a specification language for general probabilistic rewrite theories. In general, probabilistic rewrite rules such as (1) are nondeterministic, as the variables $\vec{y}$ in their right-hand side do not appear in the left-hand side, rendering them *nonexecutable* in Maude. However, Maude can be used to *simulate* a PMAUDE specification, provided that all variables $\vec{y}$ in rules like (1) are replaced with actual values *sampled* from the probability distribution $\pi(\vec{x})$. Thus, the executable Maude conditional rewrite rules have the form

$$t(\vec{x}) \longrightarrow t'(\vec{x}, \texttt{sampleFromPi}(\vec{x})) \textbf{ if } cond(\vec{x}),$$

where $\texttt{sampleFromPi}(\vec{x})$ is an operation that samples from the probability distribution π in (1). The same paper [1] introduces a technique, namely an Actor PMAUDE module, which can be used to create executable PMAUDE specifications that are free from any source of nondeterminism. This is achieved by considering the current state of the system as a multiset of *objects* and *messages*, in which, time is made explicit through a global floating point value. In an executable PMAUDE specification all rewrite rules are *scheduled* to execute at random moments of time, with the interval between two consecutive executions following an exponential probability distribution. Recall that the exponential distribution has cumulative distribution function $F(x) = 1 - \mathrm{e}^{-\lambda x}$, where $\lambda \in \mathbb{R}$ is called the *rate* parameter. As shown in [1], a stochastic time model can be implemented in the following manner: A `Configuration` is the sort of the state of a subsystem, to which the rewrite rules typically apply. In order to handle scheduling of the concurrent objects, *time* is added to the global configuration of the system, and the sorts *execution mark* and *scheduled execution mark* are added as subsorts of `Configuration`.

```
subsort Time ExecMark ScheduledExecMark < Configuration .
op time: Float → ExecMark .
op execute : Oid → ExecMark .
op [_,_] : Float ExecMark → ScheduledExecMark .
```

Here, `Oid` is the sort of object identifiers. The scheduled execution marks form the main ingredient of the stochastic time model introduced in [1], making it possible to quantify and resolve nondeterminism. A *tick* operation then makes the system evolve by unwrapping the scheduled execution marks into unscheduled ones and rendering exactly one object active. `Config` is the sort of the

global system, obtained from terms of sort `Configuration` by adding a pair of curly brackets:

```
op tick : Config → Config .
op {_} : Configuration → Config .
```

The motivation for having a *global configuration* sort is that, in order to specify the scheduling mechanism, the whole current configuration of the model must be considered. The semantics of the *tick* operation follows that of Actor PMaude [1], selecting the next object for execution in chronological order:

```
op tickAux : Float ExecMark Configuration → Config .
var CF : Configuration . vars T T' : Float . vars E E' : ExecMark .

eq tick({[T, E] CF}) = tickAux(T, E, CF) .
eq tick({CF}) = {CF} [owise] .
ceq tickAux(T, E, [T',E'] CF) = tickAux(T', E', [T,E] CF) if T' < T .
eq tickAux(T, E, CF time(T')) = {E CF time(T)} [owise] .
```

Here, **owise** equations are used only when no other equations apply and **ceq** indicates conditional equations. The global system configuration will contain exactly one time object `time(T)`. Execution marks of form `execute(O)` are added to the left-hand sides of all rewrite rules for an object `O`, as well as *scheduled* execution marks of form `[T+`δ`,execute(O)]` to their right-hand side, in order to make the new subconfiguration active at a later time, after a random interval of time δ has passed, following an exponential probability distribution with some fixed rate parameter, in our case 0.1. The random length of this interval is generated using a Maude operation denoted `sampleExpWithRate` (see Section 5). In the current implementation, the rates corresponding to the exponentially distributed waiting times of all scheduled execution marks are equal to 0.1. However, these rates can be given different values for each sensor, to simulate different sensor processor speeds. The tick rule `{ CF }` $\longrightarrow$ `tick({ CF })`, used when `CF` contains no execution mark, is built into our analysis through the script producing quantitative results. The tick rule advances time T and creates an execution mark.

3 Grouping Nodes in MULE-Based Sensor Networks

In WSNs, when a large number of sensor nodes are placed in the environment, neighbor nodes may end up being very close to one another. In this case, the transmission power level for communication with a neighbor can be kept low by using short-range multi-hop communication. Since nodes can *cooperate* to transmit data, multi-hop communication in sensor networks is expected to consume less energy than traditional single-hop communication [2]. Furthermore, multi-hop communication can effectively overcome some signal propagation effects experienced in long-distance wireless communication.

Grouping is a method of cooperation between nodes, to transfer data, in which nodes belong to distinct groups [20]. Each group has a *leader*; i.e., a node which is responsible for receiving data from the group members to route it to the sink, and also for communicating with other leaders. Outside the group, nodes always

use their maximum transmission power. Instead, by cooperating with the group members, nodes can use their minimum transmission power to reach the group leader, and consequently decrease the power consumed for communication inside the group. There are different approaches to group formation. The grouping can be done based on distance. For better grouping, other factors such as signal interference may also be considered. We use the grouping algorithm based on coalitional game theory proposed in [16], considering the grouping problem for WSNs as a coalitional game, in which the sensor nodes are the players and the game is concerned with whether a node should join a group or not, as well as which group is more beneficial to join. By using this algorithm, sensor nodes in our model can find a suitable group to join after each movement. In the model, nodes move to different locations according to a predefined set of movements.

4 A Declarative Model of MULE-Based Communication

In WSNs, nodes gather data from the environment and transmit them to sink nodes using data messages. We consider an extension of the grouping protocol in [16], in which nodes send messages to their group leaders and MULEs are responsible for moving around leaders to collect these messages and transmit them to sink nodes, in order to decrease the overall energy consumption of the network. Leaders always use their minimum power to communicate with MULEs. Also, nodes can send data messages at different rates. In general, it is better for the network to have a *fair message propagation*, in which nodes have equal message transmission rates, as it causes fair distribution of the power consumption in the network. Thus, in order to model the propagation of data messages, we assume that the next node to send a data message is selected uniformly from the set of all sensor nodes. According to this distribution, at each time tick, a node can send a data message with the same probability as the other nodes, namely $1/N$ where N is the number of nodes in the network. Single nodes communicate directly with the MULE using the maximum amount of power P^{max}. However, the nodes which belong to groups can send their data messages to the group leaders using minimum power P^{min}, and the leaders will send them to the MULE through short range communication.

Besides the modeling of data messages, the movements of MULEs are modeled using an abstract probabilistic approach to underspecify their concrete movements. The general assumption is that the MULE's movement is either random or mostly predefined [29]. Thus, we do not attempt to model a MULE's specific movements, but rather assume that the MULE always moves around the leader nodes, to increase the chance of successfully receiving messages. More precisely, Fig. 1 shows the type of probability density that we assume for locating the MULE at different coordinates. In this example, we considered three leaders at positions $(2, 3)$, $(10, 6)$, and $(4, 9)$. This probability is equal to the probability of successful message transmission between a data MULE and a leader. Outside the communication range of the leaders, the probability density breaks down to a small constant value (in our case study 0.02). Figure 2 shows a two-dimensional

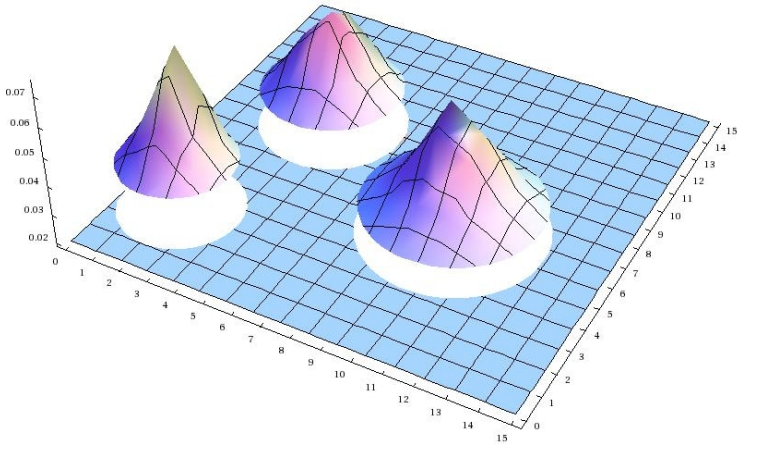


Fig. 1. Three-dimensional plot of the probability density function $f_{X,Y}(x, y)$ giving the probability of successful message transfer between a MULE found at position (x, y) and one of the leaders

density plot of the probability of successful message transfer between a data MULE found at polar coordinates (r, θ) with respect to a leader which is the *pole* of the polar coordinate system. The darker gray towards the center of the circle indicates higher values of the probability density function, while lighter gray indicates lower values. Notice from this diagram how the distance r between the leader and the MULE is calculated, as well as the angle θ between them. We may write

$$P = c \sum_{i=1}^{l} W_i, \tag{2}$$

where $c \in \mathbb{R}$ is a normalizing constant and $W_i \in \mathbb{R}$ is a weight corresponding to the chance of the MULE to be in the communication range of leader $i \in \{1, 2, \ldots, l\}$. We suggest to define this weight through the following formula

$$W_i = \int_0^{2\pi} \int_0^{Rmax_i} w_i(\theta, r)\ \mathrm{d}r\, \mathrm{d}\theta, \tag{3}$$

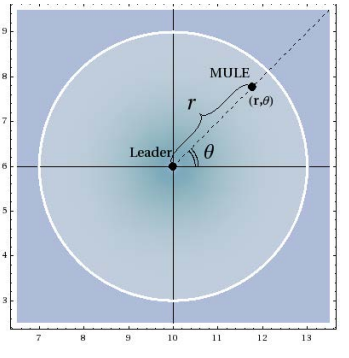


Fig. 2. Two-dimensional plot of the probability density function

with the intuition that the value $w_i(r, \theta) \in \mathbb{R}$ corresponds to the chance of successful communication between leader i and the MULE, where the polar coordinates of the MULE are given by $(r, \theta) \in [0, +\infty) \times [0, 2\pi)$ and considering that the leader is the *pole* of the polar coordinate system. Thus, the double integral in (3) calculates the "accumulated" weight associated with the leader i over the interior of the circle centered at i, with radius equal to $Rmax_i$, the communication range of i. The energy consumption of the leader

i, necessary to communicate with the MULE at a distance $r > 0$, is directly proportional to r^2 [23]. By making the natural assumption that the probability p_i of successful communication between i and the MULE is inversely proportional to the consumed energy, we obtain that p_i is inversely proportional to the squared distance r^2, which is the same order of magnitude as $(1+r)^2$. We prefer the latter expression since $1/(1+r)^2$ is well-defined for all $r \geq 0$, while $1/r^2$ is undefined for $r = 0$. Thus, we consider the weight function

$$w_i(r, \theta) = \frac{1}{2\pi} \cdot \frac{1}{(1+r)^2} \tag{4}$$

where the factor $1/(2\pi)$ corresponds to the assumption that there is an equal chance for the MULE to be located at any angle $\theta \in [0, 2\pi]$ around the leader i. In this case, we obtain a closed form expression for the weight W_i in (3):

$$W_i = \int_0^{2\pi} \int_0^{Rmax_i} \frac{\mathrm{d}r\, \mathrm{d}\theta}{2\pi(1+r)^2} = \frac{Rmax_i}{1+Rmax_i} \tag{5}$$

The signal range of each node is limited by its transmission power P_i. Following [23], the maximum distance $Rmax_i$ where the MULE can still receive messages from node i, using transmission power P_i, is given by $Rmax_i = \sqrt{P_i}$. When using the grouping protocol, we assume that P_i denotes the minimum receiving power of leader i, otherwise we assume that it corresponds to its maximum receiving power. Replacing the maximum distance $Rmax_i$ by $\sqrt{P_i}$ in (5), we obtain the following expression for the weight W_i:

$$W_i = \frac{\sqrt{P_i}}{1+\sqrt{P_i}} \tag{6}$$

The constant $c > 0$ is calculated such that (2) holds, i.e., $c = P \,/\, \left(\sum_{i=1}^{l} W_i\right)$, which allows us to define the probability $p_i = cW_i$, where $p_i \in [0, P]$, for the MULE to be in the range of leader i and to successfully communicate with it. We use these probabilities to model the behavior of the MULE when receiving or dropping messages. The main advantage of using probability distributions is that we obtain an abstract view of the MULE and ignore unnecessary details about the actual movements of the MULE vehicle and its physical communication with the sensors. In addition, our probabilistic approach for message propagation and MULE movement allows us to collect useful quantitative information for network analysis. Using discrete-event simulation, we obtain data related to the behavior of the network and to the amount of lost messages. Furthermore our model is flexible, i.e., it is easy to reuse it for different network configurations and MULE scenarios by just replacing the probability distribution in our model with another suitable distribution. In this sense, our formalism can be used as a *framework* for testing different MULE scenarios and algorithms.

In this paper, we used probabilistic rewrite theories [17] to model our grouping protocol, the propagation of data messages and also to model MULE behaviors. The next section describes how we can use this formalism to model the grouping protocol, while also incorporating probabilistic information.

5 Combining Declarative and Operational Models

In this section, we define a formal model of our proposed protocol in probabilistic rewriting logic. Our assumptions are: messages do not expire, and the number of nodes in the network is fixed, although they may move. The network is defined as a *system configuration*, a multiset of objects and messages, allowing the specification of local rules, for example to send data messages, as well as global rules, such as those used in the object scheduling mechanism. Following rewriting logic conventions, whitespace denotes the associative and commutative concatenation operator for configurations. The term $\langle$ `O: Node | leader:` L, `rpow:` E, `pow:` P, `buf:` B $\rangle$ denotes a `Node` object, where `O` is the object identifier, L its leader, E the remaining power, P the power capability, and B the message buffer.

As in [16], *unicast* messages have the form `(M from O to O')` where `M` is the message's body (possibly with parameters), `O` the source and `O'` the destination. A message will not reach its destination unless it is within the node's transmission range. *Multicasting* is modeled by allowing a set of destinations and equations which expand the destination set. *Wireless broadcasting* uses messages `(M from O to all)` where `all` is a constructor indicating that the message is sent to all nodes within range. We abstract from the actual data content of messages, and use a constant value for the message content.

In sensor networks, data is sensed from the environment continuously, and it should be transferred to the sink node. This process starts as soon as the network starts running and continues until all nodes run out of energy. Message passing is modeled by rewrite rules that can be applied at any time while the system is running, either during the grouping process or afterwards. These rules nondeterministically apply to enabled nodes in the network, so the nodes have an equal chance to pass messages to other sensor nodes.

If the selected node is a member of a group, then this node sends the data message to its group leader, using minimum power. Otherwise, it will send the message directly to the MULE, using maximum power. Fig. 3 describes the main MULE-based message passing rules in our model. The other rules in our model, such as those related to the underlying grouping protocol, are described in [16].

The *MsgFromNode* rule shows the message generated by a node. In this rule, `time(T)` is the current time, while `Pmin` and `Pmax` are defined by two equations that calculate minimum and maximum transmission power of nodes based on a value `P`, specific to each node, which we call the *power capability*.

The leaders transfer the data messages which they have received from their group members to the MULE. Rule *MsgFromLeader* represents the nondeterministic selection of one of the leaders that will pass the data message to the MULE. When a leader receives a data message from a node, it saves the message in its buffer `buf`. As soon as the buffer becomes full, the leader sends all messages to the MULE. This sending is modeled by means of a function `sendAll`, defined by two equations, which gives immediate sending of all messages in the buffer, since equations represent timeless actions in rewriting logic. In the rule, `Prec(P)` is defined by an equation calculating the power that a specific node consumes to *receive* a message, based on its power capability `P`.

```
rl [MsgFromNode]: ⟨ O : Node | leader: L, rpow: E, pow: P ⟩ execute(O) time(T)
⟶ if (L ≠ nil)
then ⟨ O: Node | leader: L, rpow: E-Pmin(P), pow: P ⟩ (msg from O to L)
else ⟨ O: Node | leader: L, rpow: E-Pmax(P), pow: P ⟩ (msg from O to "MULE") fi
[T+sampleExpWithRate(0.1), execute(O)] time(T).

rl [MsgFromLeader]: (M from O' to O) execute(O) time(T)
 ⟨ O : Leader | rpow: E, pow: P, buf: B ⟩
⟶ if #B+1≥Buffersize
then sendAll(⟨ O: Leader | rpow: E-Prec(P), pow: P, buf: push(B,M) ⟩ )
else ⟨ O : Leader | rpow: E-Prec(P), pow: P, buf: push(B,M) ⟩ fi
[T+sampleExpWithRate(0.1), execute(O)] time(T).

rl [MuleReceiveMsg]: (M from O' to "MULE") time(T) execute("MULE")
 ⟨ "MULE" : MULE | RecMsg: B, NumOfLostMsg: Y ⟩
⟶ if sampleBerWithP(probability)
then ⟨ "MULE": MULE | RecMsg: B, NumOfLostMsg: (Y+1) ⟩
else ⟨ "MULE": MULE | RecMsg: push(B,M), NumOfLostMsg: Y ⟩ fi
[T+sampleExpWithRate(0.1), execute("MULE")] time(T).

eq sendAll(⟨ O: Leader | buf: empty ⟩ ) = ⟨ O: Leader | buf: empty ⟩ .
eq sendAll(⟨ O: Leader | rpow: E, pow: P, buf: push(B,M) ⟩ ) =
(M from O to "MULE") sendAll(⟨ O: Leader | rpow: E-Pmin(P), pow: P, buf: B ⟩ ) .
```

Fig. 3. Rules for MULE-based communication. Each rule considers an object ready for execution, and reschedules the object using sampling. Irrelevant node attributes are omitted. Buffer operations include the constructor `push` and # for length. As in Maude, we assume multiset matching. Variables are capitalized. `msg` is here a constant.

The MULEs move and gather data messages which are sent by leaders or single nodes. The movement of the MULE causes some message loss, captured by the probability distribution of successful message passing (cf. Section 4). By using this probability distribution, we abstract from the movement of the MULE. In our model, every message sent to the MULE is received with a probability calculated by Equation 6 in Section 4, otherwise the message is lost (i.e., removed from the system configuration). Rule *MuleReceiveMsg* represents this process, with the `probability` variable giving the actual probability of successfully receiving a data message; i.e., $p_i = cW_i$, as defined in Section 4. The `sampleBerWithP` operation samples from the Bernoulli distribution; i.e., it returns `true` with a given probability p and `false` with probability $1 - p$.

We assume that a MULE transmits all the received messages to the sink. So in our model, there is no need for additional rules capturing the communication between the sink and the MULE. Further details about modeling the grouping and the routing protocols in rewriting logic can be found in [16]. In the present work, we extended all of the rules in the cited work to probabilistic rewrite rules, as well as added new equations. The validation of the group membership protocol can be achieved by using Maude's model checking tools. In [16], Maude's `search` tool has been applied to verify the correctness of the grouping protocol.

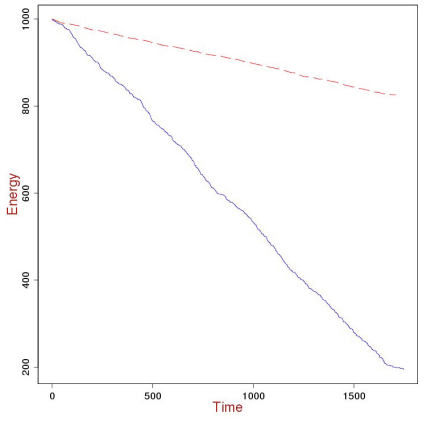


Fig. 4. The remaining energy of a node

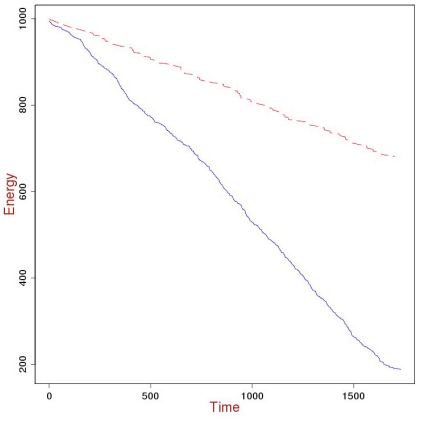


Fig. 5. The remaining energy of a leader

6 Quantitative Analysis of the Combined Model

This section proposes an approach to obtain quantitative results by guiding and monitoring Maude simulations. The basic idea is to control the run of the Maude model and monitor the system configuration at each tick. The desired data is extracted from the configuration, including numerical data stored inside each node. After the simulations, all the data extracted from the model is gathered and analyzed. To automate this process, we have implemented a Python script which extracts quantitative information from a system configuration of our model by parsing the configuration after the application of each tick rule, and extracting numerical data as queried by the user. This way, the script gathers data resulting from the application of a specified number of ticks. Finally, the script analyzes the data and provides a plot diagram showing the graph of a given system parameter against time. In addition, several simulations of the Maude model can be combined, producing a graph which averages the data obtained from each simulation. More precisely, we use a modified linear interpolation procedure that is able to precisely combine data from a set of graphs.

Thus, we used Maude to simulate our model of MULE-based WSNs, driven by the grouping protocol proposed in [16]. Our topology contains a MULE and two groups of six nodes each. Each node starts with 1000 units of energy. In the beginning of the model execution, the nodes start sending data messages. During the execution, they can move and join a new group. We capture the remaining energy of each node at every tick of the simulation, as well as the number of sent and received messages. We ran simulations for two distinct scenarios; namely, when the WSN uses the grouping protocol vs. when it does not. Our purpose is to compare the energy consumption of the nodes and the leaders, in each scenario. In addition, in order to obtain a better understanding of the network's efficiency, we define an *efficiency factor* F with the following expression $F = \frac{1}{LM} \sum_{n=1}^{N} E_n$, where N is the total number of sensor nodes, E_n is the remaining energy of node n and LM is the total number of messages that the MULE has lost. The efficiency

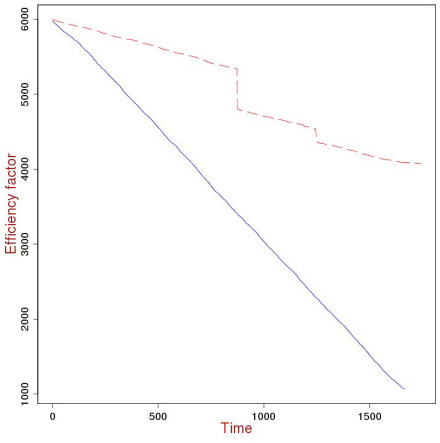


Fig. 6. The graphs of the F factor (for 6 nodes), when the MULE-based WSN is using (dashed line) and when it is not using the grouping protocol (solid line)

factor F represents a ratio between the energy consumption and the message loss in the network. More efficient networks, in terms of energy consumption and performance in message delivery, have higher values of F.

Figures 3 and 4 show the saved energy of a sensor node and of a leader, respectively, in a MULE-based WSN with (dashed line) and without (solid line) using the grouping protocol. We have also calculated the value of the F factor for a run, as the average of 5 simulations, and displayed the results in Fig. 6, in the case when the WSN uses the grouping protocol, as well as when it does not. By comparing the two graphs in Fig. 6, we observe a considerable improvement in the efficiency of the network when the grouping protocol is running. To generate each of the graphs, we ran 5 simulations (each simulation lasting for 1000 ticks).

7 Conclusion

This paper applies a combination of declarative and operational specification, using a probabilistic approach for underspecification in the operational model. Technically, this is achieved using the framework of probabilistic rewriting logic and PMaude. We demonstrate the approach on a grouping protocol for MULE-based WSNs and show how the declarative specification provides an abstract and flexible solution to model both fair message passing and underspecified MULE behavior in WSNs. Furthermore we use a statistical method for quantitative analysis of Maude models, which provides useful data sets and graphs for network analysis and performance evaluation of protocols. The obtained numerical results allow the energy efficiency of the network to be compared, with and without using the considered protocol. We have shown that using the grouping protocol improves the energy efficiency of the network. The particular choice of

parameter values used in the probabilistic modeling is based on our preliminary experience, and can easily be readjusted to fit better with reality. The approach taken provides a framework for further experimentation.

In future work, we intend to build on our current Maude model as well as to extend it, to capture *real-time* aspects of WSNs. Furthermore, we plan to subject our model to statistical model checking, to be able to statistically prove the correctness of large models. It is known that, due to their huge state space, it is practically impossible to verify such models using traditional model checking techniques. We also plan to make an integration of our current Maude implementation with the VeStA/PVeStA tool, which allows for *probabilistic reasoning* via statistical model checking and statistical quantitative analysis. Using VeStA, we would be able to verify the *statistical correctness* of the protocol proposed in this paper, as well as to make more precise quantitative analysis.

Acknowledgment. We would like to thank Lucian Bentea for his contribution to this paper, and in particular for his help with the implementation.

References

1. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. ENTCS 153(2), 213–239 (2006)
2. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002)
3. Anastasi, G., Conti, M., Di Francesco, M., Passarella, A.: Energy conservation in wireless sensor networks: A survey. Ad Hoc Networks 7(3), 537–568 (2009)
4. Basu, A., Bensalem, S., Bozga, M., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. Software Tools for Technology Transfer 14(1), 53–72 (2012)
5. Bernardo, M., Gorrieri, R.: A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Theoretical Computer Science 202(1-2), 1–54 (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. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Dario, I.A., Akyildiz, I.F., Pompili, D., Melodia, T.: Underwater acoustic sensor networks: Research challenges. Ad Hoc Networks 3(3), 257–279 (2005)
8. Dong, J.S., Sun, J., Sun, J., Taguchi, K., Zhang, X.: Specifying and Verifying Sensor Networks: An Experiment of Formal Methods. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 318–337. Springer, Heidelberg (2008)
9. Ergen, S.C., Ergen, M., Koo, T.J.: Lifetime analysis of a sensor network with hybrid automata modelling. In: Proc. 1st ACM Int. Workshop on Wireless Sensor Networks and Applications, pp. 98–104 (2002)
10. Fehnker, A., Fruth, M., McIver, A.K.: Graphical Modelling for Simulation and Formal Analysis of Wireless Network Protocols. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Fault Tolerance. LNCS, vol. 5454, pp. 1–24. Springer, Heidelberg (2009)
11. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)

12. Heinzelman, W.R., Chandrakasan, A., Balakrishnan, H.: Energy-efficient communication protocol for wireless microsensor networks. In: Proc. 33rd Hawaii Int. Conf. on System Sciences, vol. 8, p. 8020 (2000)
13. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
14. Jain, S., Shah, R.C., Brunette, W., Borriello, G., Roy, S.: Exploiting mobility for energy efficient data collection in wireless sensor networks. Mobile Networks and Applications 11(3), 327–339 (2006)
15. 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)
16. Kazemeyni, F., Johnsen, E.B., Owe, O., Balasingham, I.: Grouping Nodes in Wireless Sensor Networks Using Coalitional Game Theory. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE 2010. LNCS, vol. 6117, pp. 95–109. Springer, Heidelberg (2010)
17. Kumar, N., Sen, K., Meseguer, J., Agha, G.: Probabilistic Rewrite Theories: Unifying Models, Logics and Tools. Technical report UIUCDCS-R-2003-2347, Dept. of C. S., Univ. of Illinois at Urbana-Champaign (2003)
18. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative analysis with the probabilistic model checker PRISM. ENTCS 153(2), 5–31 (2006)
19. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. ACM SIGMETRICS Performance Evaluation Review 36(4), 40–45 (2009)
20. Lloret, J., Palau, C.E., Boronat, F., Tomás, J.: Improving networks using group-based topologies. Computer Communications 31(14), 3438–3450 (2008)
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
22. Muhammad, A., Azween, A.: Dynamic cluster based routing for underwater wireless sensor networks. In: ITSim 2010, 3 (June 2010)
23. Noori, M., Ardakani, M.: A probabilistic lifetime analysis for clustered wireless sensor networks. In: Proc. WCNC 2008, pp. 2373–2378 (2008)
24. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410(2-3), 254–280 (2009)
25. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic Linda-Based Coordination Languages. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 120–140. Springer, Heidelberg (2005)
26. Pompili, D., Akyildiz, I.F.: Overview of networking protocols for underwater wireless communications. IEEE Communications Magazine 47(1), 97–102 (2009)
27. Priami, C.: Stochastic π-calculus. Computer Journal 38(7), 578–589 (1995)
28. Sen, K., Viswanathan, M., Agha, G.: VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In: Proc. 2nd Int. Conf. on the Quantitative Evaluation of Systems (QEST 2005), USA, p. 251 (2005)
29. Shah, R.C., Roy, S., Jain, S., Brunette, W.: Data mules: Modeling a three-tier architecture for sparse sensor networks. In: IEEE SNPA, pp. 30–41 (2003)
30. Tschirner, S., Xuedong, L., Yi, W.: Model-based validation of QoS properties of biomedical sensor networks. In: Int. Conf. on Embedded Software, pp. 69–78 (2008)
31. AlTurki, M., Meseguer, J.: PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)

Mechanized Extraction of Topology Anti-patterns in Wireless Networks

Matthias Woehrle[1], Rena Bakhshi[2], and Mohammad Reza Mousavi[1,3]

[1] Embedded Software Group, Delft University of Technology, The Netherlands
m.woehrle@tudelft.nl
[2] Vrije Universiteit Amsterdam, Department of Computer Science, The Netherlands
rbakhshi@few.vu.nl
[3] Eindhoven University of Technology, Eindhoven, The Netherlands
m.r.mousavi@tue.nl

Abstract. Exhaustive and mechanized formal verification of wireless networks is hampered by the huge number of possible topologies and the large size of the actual networks. However, the generic communication structure in such networks allows for reducing the root causes of faults to faulty (sub-)topologies, called anti-patterns, of small size. We propose techniques to find such anti-patterns using a combination of model-checking and automated debugging. We apply the proposed technique on two well-known protocols for wireless sensor networks and show that the techniques indeed find the root causes in terms of canonical topologies featuring the fault.

1 Introduction

Wireless (sensor) networks are increasingly used in critical areas such as geo-science [4] and medicine [22], where correct and seamless operation is imperative. Automated formal methods, in general, and model-checking, in particular, have been used to ensure the correctness of computer systems and communication protocols, but their application to wireless (sensor) networks is barred by the well-known state-space explosion problem. This problem is severely intensified in this domain due to the huge number of possible topologies (initial states) for model-checking as well as the huge number of possible actions (next-steps) to be taken by the numerous sensor nodes present in the protocol.

In the field of model checking, some reduction techniques such as symmetry and partial-order reduction have been proposed in order to combat the state space explosion problem. Nevertheless, our experiments show that even a combination of the traditional techniques falls short of providing a solution for the state spaces resulting from sizable wireless networks.

However, the generic structure of communication primitives in wireless network protocols may come to the rescue: nodes of such networks work independently, run similar or identical protocols, which are designed regardless of the size of the network, and the protocols comprise generic and simple communication primitives. Due this generic structure, a potential problem in a large network

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 158–173, 2012.

should be traceable to a generic root cause, which also shows itself in an "anti-pattern" of small size, i.e., minimal faulty sub-topologies that demonstrate the causes of possible failures. Subsequently, the problem of model-checking large networks (with a huge number of possible topologies) is reduced to checking a small set of anti-patterns in large networks. For this approach to be successful, the anti-patterns should canonically capture the essence of possible flaws in the design. Moreover, these anti-patterns can then also serve as guidelines for the network designers to make sure that the network topology never forms such anti-patterns in its future evolutions. In this paper, we propose an approach based on a combination of model-checking and automated debugging to find anti-patterns. Searching for anti-patterns is a formal test-based approach; it is sound but not complete, because the topology space is too large to be explored exhaustively. We examine the above-mentioned approach on a number of simple, yet typical, protocols for wireless sensor networks: a probabilistic code dissemination protocol, called Trickle [21] and a medium access control protocol, called LMAC [28].

The contributions of this paper can be summarized as follows:

- We provide two complementary algorithms for determining and understanding anti-patterns.
- We show, by means of case studies, that the detected anti-patterns can canonically describe faults / implicit assumptions in protocol descriptions.

The structure of the paper is as follows. In Section 2 we provide an overview of the related work. Section 3 describes our approach and its two main algorithms. In Section 4 we describe the case studies and use them to demonstrate our approaches. We conclude with Section 5.

2 Related Work

This paper focuses on understanding the root causes that let a protocol fail on particular topologies. As such, our work is closely related to isolating faults in software executions, of which the goal is to facilitate debugging by finding a minimal failing execution. Most prominently, delta debugging [29] uses a set of passing and failing conditions in order to efficiently uncover a small failing execution. Complementary to our approach is the approach to explain counterexamples of model-checking runs [6,12]. Note that we cannot generally rely on having a complete set of counterexamples; some model-checking tools, such as PRISM [13], do not provide any counter-example. For some expressive modal and temporal logics, it is arguably questionable what the counter-example should be [8]. Additionally, as illustrated by our case studies, some topology-dependent bugs cannot be easily understood and generalized using counter-examples, yet are easily comprehensible and generalizable by identifying the topology anti-pattern. Similarly, there has been previous work on leveraging logs of sensor networks executions in order to find root-causes of errors [18,23]. Different from these works, we work on high-level specifications of protocols and are particularly interested in faults

due to specific topology patterns. Hence, in contrast to finding the exact location in one node's software, we analyze topological anti-patterns to understand the root cause of a protocol failure.

Another related approach for debugging is diagnosis in general, and spectrum-based diagnosis [1] in particular. In spectrum-based diagnosis passed or failed executions are scrutinized and annotated with information about the execution of each line (block or module) of code. Note that for diagnosis we do not differentiate whether in a particular run a block caused the failure or not; we just consider whether it is part of the whole execution. Obviously we need a different formulation (than line of code) for debugging topologies. In particular, we investigated using subgraph inclusions as "basic blocks" and count the specific subgraphs included in a topology and count their occurrence in faulty and working examples. Thereby we can use the same ideas as in spectrum-based diagnosis of software using different similarity coefficients such as the Jaccard coefficient. Diagnosis has the advantage of only requiring a fixed set of executions, while our methods necessitate additional model checking runs and are thus more time intensive. However, as demonstrated by our experimental results, the results of our algorithms are much clearer compared to the results obtained by using spectrum-based diagnosis.

We use model-checking as a vehicle for our bug-hunting approach. Model checking is an exhaustive and fully automatic state space exploration technique that has been successfully applied to many academic and industrial systems [7]. However, a naive attempt for model checking wireless sensor networks is bound to fail, due to the well-known state-space explosion problem. To overcome this problem several techniques have been proposed to reduce the state space of such networks, in particular: symmetry [9,14] and partial order reduction [11,25,27], abstraction [2,17,19], and approximation [5,20], as well as domain-specific reduction techniques [16,24]. Our earlier experiments showed that, even after applying reduction techniques, model-checking networks of actual size still remains practically infeasible. Also reducing the number of topologies using measures of symmetry did not lead to a workable subset for networks of considerable size. Hence, we decided to change our strategy and first find anti-patterns of small size, which characterize possible causes for failure, and then efficiently search for these anti-patterns in networks of larger size. Our experimental results show that this does lead to an effective and efficient debugging procedure. In view of the probabilistic features of our case studies, in this paper, we focus on model checking of probabilistic models and to this end, we use PRISM [13] as our probabilistic model-checking tool.

An alternative approach that has been used in proving correctness of wireless (sensor) network protocols is computer-assisted theorem proving [15]. The advantage of this approach is that it can provide a general proof of correctness under given assumptions. The disadvantage is that the assumptions under which the protocol works correctly is not usually precisely specified and sometimes even not known to the designers. Moreover, theorem proving requires some

affinity with the proof tools and the underlying mathematical theories. The two approaches can, however, be combined by finding anti-patterns using our approach, generalizing them and using them as assumptions (i.e., absence of generalized anti-patterns) as proof obligations.

We use two case studies of topology-related faults in wireless sensor networks that have been previously discussed. The Trickle protocol [21] has been shown to be flawed in the presentation of Anquiro [24] with respect to its threshold value for overhearing broadcast transmission. We had to make our own model of Trickle in PRISM for our experiment, since the tool presented in [24] is not available; however, our technique is applicable to any model-checking and automated verification, including that of [24]. The LMAC protocol [28] has been first modeled and verified by Fehnker et al. [10] based on timed automata models using UPPAAL [3]. The authors considered a range of different topologies, of size up to five nodes, and manually determined the causes of failures. As we demonstrate in our case studies, our approach automatically arrives at the root cause without necessitating manual generation of test cases nor analysis of 61 topologies.[1]

3 Identifying Topology Anti-patterns

The goal of this work is to detect *anti-patterns*: small faulty topologies that characterize faults or implicit assumptions inherent to a particular protocol. Once these anti-patterns are exhaustively enumerated, the problem of checking correctness for larger designs is reduced to finding anti-patterns in them, which is much more efficient than model checking the state space. Our approach is inspired by the seminal work of Zeller et al. [29] on delta debugging. Similar to delta debugging, we investigate two complementary approaches for identifying topology anti-patterns: (*i*) *minimization* of topologies to find a set of minimal topologies that fail and (*ii*) *isolation* of a single edge that changes a passing topology to a failing topology.

The premise of our approaches is that we check, for a given topology, whether the protocol model P violates the required properties ϕ given a certain topology g, i. e., $(\mathcal{P}||g) \not\models \phi$; in the present paper, we achieve this by means of model checking. Our starting point is always a failing topology (or a set thereof) and we search for the root cause of failures in these topologies. As we only decrease topology sizes (and therefore state space) in our algorithms, the runs of the model checker should always return a pass or fail answer. In case the run does not terminate with a definite answer, we assume a passing run, since we cannot prove the presence of a fault. Note that we assume that wireless network protocols are designed for any type of network, i. e., protocol properties should hold invariant of the topology. Hence, it follows that if: $(\mathcal{P}||g) \not\models \phi \implies \forall g' \subset g : (\mathcal{P}||g') \not\models \phi$.

[1] Note that Fehnker et al. included duplicate topologies in their work. The actual number of unique topologies of size 5 is 58.

3.1 Minimization

For minimization, we start with a set of topologies G, where a topology $g \in G$ is a graph, i. e., $g = (V, E)$. We focus on the set of failed runs F, i. e., $F = \{g \in G \mid (\mathcal{P}||g) \not\models \phi\}$. Given F, we try to find a set of smallest topologies S in order to determine anti-patterns.

Minimization Algorithm. Algorithm 1 summarizes our approach. Based on the set of failing topologies F and using the procedure REDUCE, we minimize each failing topology w. r. t. its number of (i) nodes and (ii) edges by calling the procedure MINIMIZE. Note that this order is implied by the fact that reduction in the number of nodes (removing all of its connected edges) is more granular than removing a single edge.

Procedure MINIMIZE reduces the number of nodes or edges respectively. The procedure MINIMIZE searches for subgraphs of smaller size and adapts the bound on the topology size until decreasing the bound results in no failing topologies. We use the same procedure both for nodes and edges (parameterized by [NODES/EDGES]), as they work identically, except for the generation of subgraphs using the function SUB. SUB removes nodes or edges, respectively, depending on its parameter. SUB(g,n) returns all (connected) complete edge-induced subgraphs of graph g of order n (NODES), or size n respectively (EDGES). A complete edge-induced subgraph of graph $g = (V, E)$ is a graph $g' = (V', E')$, $V' \subset V$ and $E' = \{(v_1, v_2) | (v_1, v_2) \in E \wedge v_1, v_2 \in V'\}$. Note that we can trivially speed up the algorithm by memoizing calls to MINIMIZE with previously checked topologies.

The output of REDUCE is the set of smallest topologies S.[2] As we see in the case studies, this results in a small set of minimal topologies that may represent the essence of the fault.

3.2 Isolation

Minimization results in a small set of graphs that (may) explain the underlying fault of the protocol. Additional to minimization, we also perform fault isolation, i.e., to identify the discriminating edge that lets a protocol fail. We start on the one hand with a failing topology and on the other hand with a passing topology and close in on the fault. Algorithm 2 presents the details of the approach: The user provides one failing topology f_{in}. We use as an initial passing topology a graph with a single node that trivially satisfies requirements.[3] The algorithm relies on building the relative complement δ of the failing and the passing topology, i. e., $\delta = E_{f_-} \setminus E_{f_+}$, where f_-, f_+ are the currently smallest failing or largest passing topology respectively. We sample from δ to shrink the failing and extend the passing topologies, respectively, such that the new topology f_{new} is also a connected graph. Please note that in this way the passing topology is always a

[2] When building set S, we check that each element $s \in S$ is unique modulo graph isomorphism.

[3] Depending on the protocol requirements, a larger passing topology with more nodes may be used.

Algorithm 1. Network minimization based on binary search

procedure REDUCE($\mathcal{P}, \phi, F$)
  /* **input**
    $\mathcal{P}$, Protocol model
    ϕ, Protocol properties
    $F = \{f_1, \ldots, f_m\}$, Set of faulty topologies with $f_i = (V_{f_i}, E_{f_i})$
  output
    $S = \{s_1, \ldots, s_n\}$, Set of smallest topologies */
  $H = S = \emptyset$
  // Minimize faulty topologies
  for $f \in F$ **do**
    $H = H \cup$ MINIMIZE[NODES]$(\mathcal{P}, \phi, F, 1, |V_f|)$
  end for
  for $h \in H$ **do**
    $S = S \cup$ MINIMIZE[EDGES]$(\mathcal{P}, \phi, H, 1, |E_h| - |V_h| + 1)$
  end for
  return S
end procedure
procedure MINIMIZE[NODES/EDGES]$(\mathcal{P}, \phi, T, low, high)$
  /* **input**
    $\mathcal{P}$, ϕ as before
    T, Set of faulty topologies
    $low, high \in \mathbb{N}$, Upper and lower bound
  Topology minimization using binary search */
  if $high > low$ **then**
    $middle = \lfloor (low + high)/2 \rfloor$
    $U = \emptyset$
    for all $t \in T$ **do**
      for all $r \in$SUB[NODES/EDGES]$(t, middle)$ **do**
        if $(\mathcal{P}||r) \not\models \phi$ **then**
          $U = U \cup \{r\}$
        end if
      end for
    end for
    if $U \neq \emptyset$ **then**
      return MINIMIZE[NODES/EDGES]$(\mathcal{P}, \phi, U, low, middle - 1)$
    else
      return MINIMIZE[NODES/EDGES]$(\mathcal{P}, \phi, T, middle + 1, high)$
    end if
  else
    return T
  end if
end procedure

Algorithm 2. Fault isolation using a delta debugging strategy

```
1: procedure ISOLATE(P, φ, f_in)
2:     /* input
3:         P, Protocol model
4:         φ, Protocol properties
5:         f_in, Faulty topology
6:     output
7:         f_−, Smallest failing topology
8:         f_+, Largest passing topology */
9:     f_− = f_in, f_+ = ({0}, ∅) // Note that f_+ ⊆ f_−
10:    // Loop until one-edge difference between f_−, f_+
11:    while SIZE(f_− − f_+) > 1 do
12:        δ = E_{f_−} \ E_{f_+}
13:        f_new = f_− − δ', δ' ⊆ δ, s.t. f_new is connected
14:        if (P||f_new) ⊭ φ then
15:            f_− = f_new
16:        else
17:            f_+ = f_new
18:        end if
19:    end while
20:    return f_−, f_+
21: end procedure
```

subgraph of the failing topology. If the newly created topology passes we assign it to the currently largest passing topology f_+, else it is the currently smallest failing topology f_-. Thereby, we iteratively increase/decrease the topologies until they differ by a single edge. This single edge is instructive on why the protocol fails.

3.3 Discussion

The minimization and isolation algorithm are different than the original delta debugging formulation as graphs as relational data have a different structure than execution traces: The difference for the minimization algorithm is that instead of partitioning as described in delta debugging, we check all subgraphs of a given size (w. r. t. nodes and edges). Further research is needed to investigate different partitioning/bisection strategies, in particular how to handle the cut set of the partitioning, and compare them with the subgraph-based approach proposed in this work. Similarly, since we need to build a complement graph for the isolation algorithm, it does not matter whether we grow from the passing graph or decrease the failing graph. As such in our formulation we only remove from the failing graph yet still approach the isolating edge from both sides.

Please note that minimized topologies also include isolation information. In a minimal topology removing *any* edge will remove the fault. Since the minimization algorithm necessitates more model checking runs evaluations than isolation, there is a tradeoff between execution time and quality of results. Finally, we need

to consider that both algorithms are heuristics. That means if we have multiple faults in a protocol our algorithms potentially misses some of them. Since faults are typically gradually fixed, this is not an issue. Additionally, we show in the case studies in Sec. 4.5 that the algorithm can find the causes of multiple faults.

4 Case Studies

To demonstrate our methodology, we considered two protocols for wireless sensor networks, namely, Trickle [21] and LMAC [28]. In this section, we briefly describe each of the case studies and present the anti-patterns detected using our approach.

4.1 Experimental Setup

We base our experiments on a set of randomly generated undirected graphs with a dedicated sink node. We generate these graphs using the algorithm described in Rodionov et al. [26]. Our protocols features a notion of sink (a node from which the updates originate, see below). We run PRISM 4.0.1. All algorithms and graph operations are performed using Python 2.7 and NetworkX 1.6[4]. We automatically generate PRISM models for a fixed topology of the network. Our script takes a topology description as input, and generates the concrete PRISM model.

4.2 PRISM

We modeled both protocols using the probabilistic model checker PRISM [13]. The model checker automatically computes precise quantitative results based on an exhaustive analysis of a formal model. We specified the protocols in PRISM's state-based input language as discrete-time Markov chains (DTMCs), since they exhibit probabilistic behavior.

In PRISM, a system consists of a set of communicating modules, each with its local variables of the integer type. The evolution of each module is described by a set of guarded commands of the form: `[a] c` $\rightarrow$ `` `p`$_1$`:`e$_1$ `+ ... +` `p`$_n$`:`e$_n$`;``. Such a transition consists of the predicate `c` on the state variable, also called a *guard*, the action label `a`, and a probabilistic update relation `` `p`$_i$`:`e$_i$``. If `c` evaluates to true, then update `` e$_i$ `` is applied with the probability `` p$_i$ ``. Modules can synchronize either on global shared variables or on common actions labels. Note that PRISM implements CSP-style synchronization over an action label `a`: it requires the participation of all modules with the common action label `a` simultaneously.

Once the system is specified in terms of its modules, PRISM constructs a stochastic transition system for the composition of specified modules. Analysis is performed through model checking such systems against properties specified in the probabilistic temporal logic PCTL (for the DTMC model).

[4] http://networkx.lanl.gov/

4.3 Verifying Trickle

Description: Trickle is a probabilistic code dissemination (maintenance) protocol. The goal of the protocol is to update all nodes with new versions of a deployed software. The software update is first published at a sink (also called base or root) node and is propagated among the involved nodes using a "polite gossiping" approach.

In a nutshell, the protocol works as follows:

- Each node that hears about a new update, pulls the update from the source and schedules an announcement to inform the new update to its neighbors.
- If prior to the announcement, the node hears at least w neighbors announcing the update, it cancels its own announcement. (We call w the *broadcast parameter*.)
- If a node hears a neighbor announcing an older update (than its local version), it schedules an announcement of its own (newer) update as above.

If the network is connected, all nodes executing the Trickle protocol should eventually receive the published update.

Implementation: We are interested in Trickle's control flow and thus modeled a spread of 'the most recent update' throughout the network, executing Trickle. Thus, it is sufficient to use a single bit to indicate whether a node received the update (as 1), or an older version (as 0).

Each node i is modeled as a PRISM module, maintaining local variables $\texttt{rcv}_{ij}$ for all its neighbors j. These variables indicate whether the recent update has been received by node i from its neighbor j. Only the sink, i. e., node 0 has a constant `rcv0` with the value 1, thereby initially publishing the update. The nodes communicate via message channels, represented by the action labels $\texttt{msg}_i$, with node i 'broadcasting over this channel' to all its neighbors. The broadcast medium is implemented as an additional module `broadcaster`, simulating nodes that initiate a broadcast. The module chooses the broadcasting node uniformly at random. The node modules only wait for announcements and receive the update. Our model is parameterized on the broadcast parameter w, assumed to have value $w := 3$ in the following. This parameter defines an upper threshold: if a node has heard broadcasts from w neighbors, it stops broadcasting itself. The broadcaster has two types of transitions, a labelled command and a non-labelled one; the node modules have only labelled commands. Each synchronization (labelled) command in the model is guarded by the constant w. As soon as any node exceeds w, only non-synchronizing (local) transitions are enabled for this node at the broadcaster module.

In our experiments, we verify whether all nodes eventually receive the recent datum with probability 1. This can be formulated for PRISM as:

$$\mathit{filter}(\mathit{forall}, \mathcal{P} \geq 1[\mathcal{F}(\text{"}\mathit{all}\text{"})]) \tag{1}$$

where the state `all` is specified as the conjunction for n nodes: $\bigwedge_{i=0}^{n-1} \bigvee_j (\texttt{rcv}_{ij}! = 0)$. Simply put, `all` is the state where every node i has received the recent update $\texttt{rcv}_{ij}! = 0$ from a neighbor j (at least once).

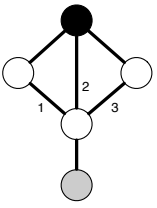


Fig. 1. Result of minimizing Trickle

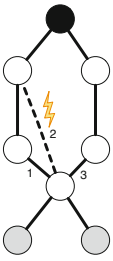


Fig. 2. Result of isolation for Trickle

Minimization Results: For Trickle, we generate a set of 50 random topologies of order 8; in this set of topologies there are four faulty topologies. We minimize these four topologies to a single anti-pattern that is shown in Fig. 1. In this figure, and all other figures to come, we denote the sink node with a black circle and the failing node with a gray circle. We can clearly see how the broadcast parameter (set to 3), prevents the parent of the gray node to send an update. This means that while the gray node is failing, the cause is actually due to the fact that it has a single parent that is blocked by the broadcast parameter.

Isolation Results: We select one faulty topology and perform isolation. Fig. 2 presents the results where the dashed edge indicates the edge difference between the passing and the failing topology. In this case the isolation algorithm merely removes a single edge from the initial failing topology, yet grows the passing topology to seven nodes. Although the resulting graphs feature two additional nodes compared to the minimization result in Fig. 1, the underlying fault is clearly visible in the differentiating edge 2. That is, adding a third predecessor node to the (single) parent of the gray node results in a failure.

4.4 Verifying LMAC

Description: LMAC [28] is a medium access control protocol for wireless sensor networks. LMAC is from the class of time-division multiple-access protocols: time is segmented into (time) frames and frames are split into fixed-length time slots. In each time slot a single node (in a given range) should have exclusive channel access for transmission in order to avoid collisions.

The goal of the protocol is to assign time slots to nodes in a distributed fashion. A node can then transmit its messages in the time slot it owns. For the nodes that are within range of each other, only one node owns a given time slot, so that only one node can transmit at a time. To limit the overall number of time slots, LMAC allows for reuse of time slots by the nodes at a non-interfering distance.

Nodes maintain a table of the time slot occupancy for its neighborhood. This table is synchronized with other nodes by transmitting a short control message in their time slots. In its time slot, a node broadcasts a bit array of slots chosen by

its (one-hop) neighbors and itself. When a node receives such a control message from a neighbor, it stores the respective time slots of the two-hop neighborhood in its table.

LMAC is initiated at a gateway node, which is the first to select a time slot to control, and, thereafter, initiates the protocol by sending its slot occupancy table. In the bootstrap state, i. e., after a node receives its first message, it listens for an entire time frame to any messages from its neighbors. Based on the control messages, nodes will determine the time slots that are currently occupied. Since a node cannot control a time slot occupied by its one-hop and two-hop neighbors, it randomly chooses one of the remaining time slots.

A node, that already owns a time slot, executes the protocol in three steps:

- It listens for messages during the time slots other than its own one. If the node detects a collision, it stores the corresponding time slot in order to notify its neighbors. This is necessary since a node cannot detect collisions it may have caused by itself, since the radio of a sensor node cannot transmit and listen at the same time.
- During its own time slot, the node transmits a control message, which includes the time slots it knows are occupied, and the time slots where it detected collisions.
- If the node is notified about a collision that occurred in its time slot, it chooses a number of time frames to wait, and proceeds to choose another available time slot as described above.

Eventually all nodes should be able to transmit messages in their time slots, without interfering with each others transmissions.

Implementation: We verify the time slot distribution procedure of LMAC. Each node i is modeled as a PRISM module, which maintains several local variables: the selected time slot $\texttt{own\_ts}_i$, $\texttt{state}_i$ indicating the state node i is in, and a time slot with detected collision $\texttt{col\_ts}_i$. In addition, every node i maintains an array slot $\texttt{ts}_{ij}$ to record the occupancy of all time slots j.

According to the LMAC protocol and the model in [28], nodes are assumed to be globally time synchronized. Thus, we model a global clock as a separate module `timer` with the current time slot number `timeslot` and the current time frame `timeframe` as variables. Three types of transitions of `timer` enable time progress in the model: *(i)* the non-synchronizing transition is enabled if a current time slot `timeslot` is not controlled by any node in the network, *(ii)* nodes transmit and receive control messages using the labelled transition `ts`, *(iii)* synchronization on the labelled transition `decide` allows nodes to select a time slot to control. A node decides uniformly at random on a new time slot to control (if more than one are unoccupied), and on the number of time frames to back-off.

Our PRISM model is parameterized by `t`, the number of time slots in each time frame. Note that there need to be sufficient time slots in a time frame for a slot allocation to be feasible, e. g., at least n time slots for cliques of size n.

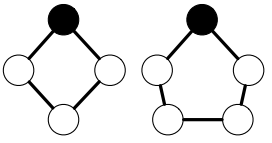


Fig. 3. Result of minimizing LMAC

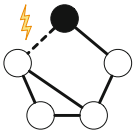


Fig. 4. Result of isolation for LMAC

Similar to Fehnker et al. [28], we introduced two rules in our model that were underspecified in the informal description of LMAC[5]:

- a node may not select a new time slot, if it did not received a control message from at least one of its neighbors;
- upon sending a control message to the neighbors, a node resets all array entries $\mathtt{ts}_{ij}$, except for its own controlled time slot. Thereby it propagates only time slot information received in the recent time frame.

In our experiments with LMAC, we verify whether any two nodes i and j that are one or two-hop neighbors will eventually proceed to choose a new time slot if they experienced a collision. This property is expressed as follows:

$$\begin{aligned} &\mathit{filter}(\mathit{forall},((\mathtt{own\_ts}_i = \mathtt{own\_ts}_j)\&(\mathtt{state}_i = 2)\&(\mathtt{state}_j = 2)) \\ &\quad =>\mathcal{P} \geq 1[\mathcal{F}\,((\mathtt{state}_i = 0)|(\mathtt{state}_j = 0))]) \end{aligned} \tag{2}$$

where $\mathtt{state}_i = 2$ denotes that the node i is broadcasting in the current time slot. $\mathtt{state}_i = 0$ means that the node i is recording occupancy of the time slots from control messages of its neighbors in order to select a new time slot.

Minimization Results: We run minimization on a (sub-)set of six node networks; in particular we choose three topologies that can be scheduled using merely four slots ($\mathtt{t}:= 4$). The minimization algorithm returns two topologies – a four node ring and a five node ring as shown in Fig. 3. This is the optimal (minimal) result: Rings of more than four nodes cause the LMAC property to fail. Note that for LMAC this is not because of some parameterization of the protocol or a specific issue with the neighborhood but an emergent detrimental property of the protocol. If two neighboring nodes that have no common neighbors end up in a collision, this collision cannot be detected by other nodes and hence cannot be resolved.

Isolation Results: We perform isolation on one of the topologies that we used for the minimization algorithm.[6] The result is depicted in Fig. 4. We can see here that the closing of this ring of four causes the fault. Note that we see here one additional detail – the ring of three in the lower left is not a problem; yet a ring of four in the upper right is.

[5] Our model corresponds to model 11 in [28].
[6] The faulty topology corresponds to number 29 in [28].

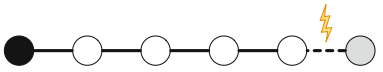


Fig. 5. Isolation of multiple faults

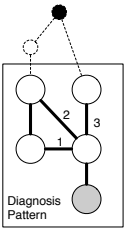


Fig. 6. Anti-patterns detected using diagnosis

4.5 Handling Multiple Faults

As a final test in our case study we look at the Trickle protocol and additionally inject an error for topologies that have a path from the sink in the graph of at least 5 hops in order to represent a second fault that is depth or forwarding related. Our minimization algorithm returns two topologies. One is related to the trickle fault; it is exactly the same as the fault described in Sec. 4.3. The other topology is a graph consisting of a chain of 6 nodes. As we can see the results clearly indicate the two different types of faults that we inject. In contrast, when we select a single topology and perform isolation we only investigate one underlying cause. In this case, we consciously select a topology that is due to both faults. As we can see in Fig. 5 isolation returns one of the two faults. In this case we find (the simpler) depth-related inject that gets triggered by the edge that increases the distance of the gray node to 5 hops.

4.6 Comparison to Diagnosis

In order to compare our results with diagnosis, we ran spectrum-based diagnosis as described in Sec. 2 for the Trickle testcase using subgraphs of order 4 and 5. We assume here a certain size of error which is reasonable, as we would start debugging from small-size topologies. The highest ranked subgraph, which is shown in the box in Fig. 6, is of order 5. Note that in this figure, we arrange the graph to make the pattern contained in the box more visible. In particular, we add a potential embedding into a larger graph outside the box just for clarification; however, this embedding is not part of the diagnosis result. In the pattern, the nodes on the top are not connected to a sink node. As we can see, such a pattern does not necessarily demonstrate the core cause. Hence, these results are not as helpful for debugging purposes as the patterns generated using the anti-pattern approach, which clearly identifies the source of the problem.

5 Conclusions

In this paper, we presented an approach, inspired by delta debugging, to find the root causes of failure in wireless network protocols. The causes are represented

in terms of minimal topologies, called anti-patterns. We also developed another approach inspired by fault diagnosis and showed that the approach based on delta debugging is more effective in demonstrating the root causes of failure.

Although anti-patterns explain the faults or implicit assumptions of a protocol, their presence in a larger network does not necessarily lead to a failure. We plan to extend our notion of anti-pattern to capture the boundary conditions on the nodes, which also capture when these faults do lead to a failure in larger topologies. We think that the isolation technique does provide additional information that can be used in characterizing the boundary conditions under which a fault will necessarily be triggered.

Acknowledgements. Matthias Woehrle is supported by the Dutch Technology Foundation STW and the Technology Programme of the Ministry of Economic Affairs, Agriculture and Innovation. The authors thank Koen Langendoen, Tim Willemse, Wan Fokkink and the anonymous reviewers for their valuable feedback.

References

1. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.: A practical evaluation of spectrum-based fault localization. Journal of Systems and Software 82(11), 1780–1792 (2009)
2. Bakhshi, R., Endrullis, J., Endrullis, S., Fokkink, W., Haverkort, B.: Automating the mean-field method for large dynamic gossip networks. In: Proc. Conf. on Quantitative Evaluation of SysTems (QEST), pp. 241–250. IEEE Computer Society (2010)
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. Beutel, J., Gruber, S., Hasler, A., Lim, R., Meier, A., Plessl, C., Talzi, I., Thiele, L., Tschudin, C., Woehrle, M., Yuecel, M.: PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In: Proc. 8th ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks (IPSN 2009), pp. 265–276. ACM/IEEE, San Francisco, CA, USA (2009)
5. Cadilhac, M., Hérault, T., Lassaigne, R., Peyronnet, S., Tixeuil, S.: Evaluating complex MAC protocols for sensor networks with APMC. In: Proc. Workshop on Automated Verification of Critical Syst. (AVoCS 2006). ENTCS, vol. 185, pp. 33–46. Elsevier (2007)
6. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT 2004/FSE-12, pp. 73–82. ACM, New York (2004)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
8. Clarke, E.M., Veith, H.: Counterexamples Revisited: Principles, Algorithms, Applications. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2004)
9. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design 9(1/2), 105–131 (1996)

10. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
11. Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991)
12. Groce, A., Visser, W.: What Went Wrong: Explaining Counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
13. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
14. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
15. Kamali, M., Laibinis, L., Petre, L., Sere, K.: Self-recovering sensor-actor networks. In: Mousavi, M.R., Salaün, G. (eds.) FOCLASA. EPTCS, vol. 30, pp. 47–61 (2010)
16. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
17. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-Valued Abstraction for Continuous-Time Markov Chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007)
18. Khan, M.M.H., Le, H.K., Ahmadi, H., Abdelzaher, T.F., Han, J.: Dustminer: troubleshooting interactive complexity bugs in sensor networks. In: SenSys 2008: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, pp. 99–112. ACM, New York (2008)
19. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for markov decision processes. In: Proc. Conf. on Quantitative Evaluation of SysTems (QEST), pp. 157–166. IEEE Computer Society (2006)
20. Laplante, S., Lassaigne, R., Magniez, F., Peyronnet, S., de Rougemont, M.: Probabilistic abstraction for model checking: An approach based on property testing. In: Proc. IEEE Symp. on Logic in Comput. Sci. (LICS 2002), pp. 30–39. IEEE Computer Society (2002)
21. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation, vol. 1, p. 2. USENIX Association (2004)
22. Lorincz, K., Chen, B.-R., Challen, G.W., Chowdhury, A.R., Patel, S., Bonato, P., Welsh, M.: Mercury: a wearable sensor network platform for high-fidelity motion analysis. In: Proc. ACM Conf. on Embedded Networked Sensor Systems, SenSys 2009, pp. 183–196. ACM, New York (2009)
23. Khan, M.M.H., Abdelzaher, T., Gupta, K.K.: Towards Diagnostic Simulation in Sensor Networks. In: Nikoletseas, S.E., Chlebus, B.S., Johnson, D.B., Krishnamachari, B. (eds.) DCOSS 2008. LNCS, vol. 5067, pp. 252–265. Springer, Heidelberg (2008)
24. Mottola, L., Voigt, T., Österlind, F., Eriksson, J., Baresi, L., Ghezzi, C.: Anquiro: enabling efficient static verification of sensor network software. In: Proc. ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA), pp. 32–37. ACM, New York (2010)

25. Peled, D.: All From One, One For All: on Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
26. Rodionov, A.S., Choo, H.: On Generating Random Network Structures: Trees. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003, Part II. LNCS, vol. 2658, pp. 879–887. Springer, Heidelberg (2003)
27. Valmari, A.: A Stubborn Attack on State Explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
28. van Hoesel, L., Havinga, P.: A lightweight medium access protocol (LMAC) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In: Proc. Workshop on Networked Sensing Systems (INSS), pp. 205–208. Society of Instrument and Control Engineers, SICE (2004)
29. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28, 183–200 (2002)

A Proof Framework for Concurrent Programs

Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen

ICIS, Radboud University Nijmegen, The Netherlands
{l.lensink,s.smetsers,m.vaneekelen}@cs.ru.nl

Abstract. This paper presents a proof framework for verifying concurrent programs that communicate using global variables. The approach is geared towards verification of models that have an unbounded state size and are as close to the original code as possible. The bakery algorithm is used as a demonstration of the framework basics, while the (full) framework with thread synchronization was used to verify and correct the reentrant readers writers algorithm as used in the Qt library.

1 Introduction

Parallelism has been employed for many years, mainly in high-performance computing. The physical constraints preventing an unlimited growth in processor speed have led to a revival of interest in concurrent computing. Parallel computing has become a dominant paradigm in computer architecture, particularly for multi-core systems [13].

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs. In practice, it can be incredibly difficult to track down these software bugs, because of their unpredictable nature: they are typically caused by infrequent 'race conditions' that are hard to reproduce. In such cases, it is necessary to thoroughly investigate 'suspicious' parts of the system in order to improve these components in such a way that correctness is guaranteed. The most thorough technique is to formally verify properties of the system under investigation.

In an earlier paper [14] a case study is presented that combines two formal verification methods, namely model checking and theorem proving. The idea is to use the model checker to track down and remove (concurrency) bugs, and to use the theorem prover to formally prove their absence. Model checkers and theorem provers have their own input languages. Therefore, the use of these formal tools requires that the original program is first converted to (modeled in) the language of the model checker, and subsequently translated into the language of the theorem prover. Obviously, both translations introduce potential sources of errors, particularly if these translations are performed manually.

The experience with this case study led us to develop a *general proof framework* with specific support to construct proofs of mutual exclusion, deadlock and starvation properties for concurrent algorithms that communicate using shared variables. The proof framework consists of a *model* that can be instantiated and used for different programs, a *set of theorems* that can be used to prove

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 174–190, 2012.

relevant properties of the system and a *general approach* towards solving the proofs and proof obligations generated by the framework. Using SPIN [5] as model checker, we investigate how (concurrent) Promela (the input language of SPIN) programs can be modeled in PVS [10]. We define an automatic translation within the framework that serves as a basis not only to facilitate the conversion of Promela into PVS, but also to support the investigation of general properties of parallel computer programs.

In this paper, this framework is introduced. It integrates model checking with theorem proving. An approach like this is used earlier in VeriTech [6], a translation framework between different formal notations. Novel in this approach is that it provides a translation extended with theorems and proof strategies, and unlike TAME [1], which is geared towards automaton models, the intended use is to prove properties of computer programs that make use of communication primitives. The use of the framework is explained applying it to a common mutual exclusion algorithm known as the *bakery* algorithm [7]. We demonstrate the power of the framework by applying it to a larger example, showing correctness of a solution to the reentrant readers-writers problem [15] that improves upon the widely used Qt C++ library by Trolltech. In that paper [15] it was described how a model was constructed and checked using the SPIN model checker. This revealed an error in the implementation, and a correction was suggested. The improved algorithm was subsequently shown to be correct in PVS. However, the PVS model was constructed manually. Here, we show how the model can be generated automatically, and how the proof can be structured using the support of the framework. For this paper, some knowledge of PVS and Promela or similar formal specification languages is presumed.

Section 2 introduces the framework basics. In Section 3 these framework basics are applied to a classic example, the bakery algorithm. Section 4 adds thread synchronisation to the framework and applies it to a large example, the reentrant readers-writers problem. Section 5 draws conclusions and suggests future work.

2 Framework Basics

The general approach is to take a piece of (parallel) code, and model it in a model checker to detect bugs. Subsequently, after improving the model it will be subject to verification in a theorem prover. To do this systematically, an embedding of the semantics of the model checker in the theorem prover is required. In our case, we use PVS as theorem prover and Promela as the modeling language. The embedding presented in this paper is a mixed shallow/deep one: the framework is based on a shallow embedding while the translation of the model into PVS exploits the native features of PVS as much as possible, and hence can be seen as a deep embedding.

Transition System

In essence, a SPIN model is a state transition system with temporal logic. Our framework reflects this directly by representing concurrently executing threads

by means of a state transition system. Each process runs in a *thread*. The semantics of executing threads are captured in a theory that specifies that each thread is either *Running*, *Waiting*, or *Blocked*. All threads have a threadid `tid` of type `TID`, a program counter `pc`, a return address `rtn` and a local store `local`. The types of these entities are provided as theory type parameters, and will be supplied by the concrete (translated) Promela program. The theory parameter `NT` denotes the number of concurrently executing threads.

```
Threads[NT:posnat, PC, LS, GS: TYPE] : THEORY BEGIN
  TID     : TYPE = below(NT)[1]
  TStatus : TYPE = { Running, Waiting, Blocked }
  TState  : TYPE = [# tid: TID, status: TStatus, local:LS, pc, rtn: PC #][2]
  Threads : TYPE = [ TID → TState]
  System  : TYPE = [# threads: Threads, current: TID, global: GS #]
  currThread(s: System): TState = s`threads(s`current)[3]
END Threads
```

The entire system state consists of all the threads combined with the global variable store `global` (again a theory type parameter), and a variable `current` signifying which thread is currently executing. The utility function `currThread` yields the state of the currently executing thread.

The (global) state transition relation of the system is determined by the (local) state transition of the concurrently executing threads. The behavior of each thread is specified by means of a `step` relation. This relation is defined in a separate theory `Model`, also containing definitions of the entities required by `Threads`. This `Model` theory, resulting from the translation of the Promela program (say P), has the following skeleton. Sections 3 and 4 show how this skeleton is instantiated for different Promela models.

```
Model[NT:posnat] : THEORY BEGIN
  PC:TYPE= below( ... number of instructions generated from P ... )[4]
  GV:TYPE= [# global variables appearing in P #]
  LV:TYPE= [# local variables of each thread in P #]
  IMPORTING Threads[NT, PC, LV, GV]
  step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool = generated from P
```

The effect of `step` is local, i.e. it only affects the local state of the currently executing thread, and possibly the global state of the system. The local states of other threads are untouched, which also follows from `step`'s type. To enforce this kind of locality for the entire system, we introduce the parameterized predicate `PredSys` on `System` that when applied to a system s, identifies all valid predecessors of s.

```
PredSys(s: System): pred[System] =
  { p: System | ∀(ot:TID): ot≠s`current ⇒ p`threads(ot) = s`threads(ot) }
```

[1] Denotes the set of natural numbers between 0 and `NT`, exclusive of `NT`.
[2] Recordtypes in PVS are surrounded by `[#` and `#]`.
[3] `r`x` denotes the selection of the `x`-component of record `r`.
[4] `below(n)` denotes the set $\{0..n-1\}$.

As usual, we will model parallel execution by non-deterministic interleaving. To anticipate on the proving process we already include the notion of invariancy, by means of an predicate on the `System` type, called `invSystem`. This leads to the following `interleave` relation, performing one execution step of a randomly selected running thread.

```
interleave(ps:(invSystem)[5], ss: System) : bool =
  PredSys(ss)(ps) ∧ currThread(ps)‘status = Running ∧
  LET cs=ps WITH [‘current = ss‘current][6] IN
    step(currThread(cs),cs‘global)(currThread(ss),ss‘global)
```

Theorems and Proofs

PVS has no innate notion of deadlock or starvation, so these have to be defined explicitly. *Deadlock states* are usually defined as states for which there are no outgoing edges. Although *final states* may have no outgoing edges, they are not considered as deadlock states. Assume that `zeroState` denotes a predicate identifying these final states, we can formulate *deadlock-freeness* as:

```
∀(s:(invSystem)): ¬zeroState(s) ⇒ ∃(t:System): interleave(s,t)
```

This interpretation of deadlock is often not precise enough. Consider for example a situation where a process executes a non terminating loop (because it is waiting for something that will never occur). Then, it might be that all other threads are waiting for this one to terminate before they can proceed. According to the definition there would be no deadlock. To handle this situation a refined notion of deadlock-freeness is needed. This refinement is based on the observation that if there exists a (well-founded) order $<$ on states such that from every non-final state s of the system a transition can be made to a state t with $t < s$, then the system will be free of deadlock. More formally:

```
NoDeadlock(s:(invSystem)) : bool =
    ¬zeroState(s) ⇒ ∃(t:System): interleave(s, t) ∧ t < s
```

Proving deadlock-freeness of a system boils down to giving an appropriate state ordering and showing that the generated `step` relation indeed has this `NoDeadlock` property.

The previous theorem can also guarantee freedom from starvation, if fairness of scheduling is imposed on the system. However, most (efficient) thread implementations do not provide this way of scheduling. Therefore, we introduce a more sophisticated notion of starvation freedom that makes no specific assumptions on the scheduling regimen. We base this notion on the following intuition: if a process *intends* to perform a certain action it will *eventually* be able to. The intention is signaled by a process entering a certain execution path. For instance, executing the instruction that puts it on the path to enter a critical section. Execution of the intended action is signaled by reaching the goal instruction, e.g. when the process finally gets the permission to enter the critical section. This

[5] PVS allows predicates to be used as types.

[6] `r WITH [‘x := e]` denotes a new record that is equal to `r` except for the `x`-component which has value `e`.

leads to the following definitions, in which both intention and goal are specified as PC values.

```
NoStarvation(s:(invSystem), t:TID, enter, goal:PC) : bool =
 s`threads(t)`PC = enter ⇒ eventually(s, t, goal)

eventually(s1:(invSystem), t:TID, goal:PC): RECURSIVE bool = ∀(s2:System):
  interleave(s1,s2) ⇒ s2`threads(t)`pc = goal ∨ eventually(s2,t,goal)
MEASURE noStarvationMeasure(s1,t)
```

In PVS all functions must be total. For recursive functions, such as `eventually` above, a so called *measure* must be provided. This measure, based on some well-founded order, ensures that at least one of the function arguments strictly decreases (according to the order) at each recursive call, thus ensuring termination. In the case above, termination also guarantees freedom of starvation, because only a finite number of interleaving steps are possible before the thread reaches its goal. Proving the absence of starvation boils down to giving a proper definition of `noStarvationMeasure`. In combination with deadlock-freeness this results in eventually reaching the goal. In the sequel, we will also specify the state ordering for deadlock-freeness as a measure with the obvious name `noDeadlockMeasure`.

In general, these measures will involve parts of the global system state as well as properties of individual threads. In order to define and facilitate reasoning about these measures the following small PVS theory proves to be very useful. It contains a folding operation `fsum` that accumulates the results of a function `fun`, provided as a parameter. The lemma relates the results of `fsum` applied to functions f and g for which there exists at most one argument for which f and g produce different results.

```
fsum(m:upto(N),fun:[below(N)→nat]):RECURSIVE nat =
  IF m=0 THEN 0 ELSE fun(m-1)+fsum(m-1,fun) ENDIF MEASURE m

fsum_diff: LEMMA
  ∀(n:upto(N),k:below(n),f,g:[below(N)→nat]):
    (∀(m:below(n)): m≠k ⇒ f(m)=g(m)) ⇒ fsum(n,f)+g(k) = fsum(n,g)+f(k)
```

Translating Spin Models to the Framework

In this section we show how Promela programs are translated into our PVS framework. Since we focus on concurrent systems in which processes communicate via shared variables, it is not necessary to cover Promela completely. In particular, the inter process communication via channels is left out. The core of the translation is the way Promela statements are treated. To facilitate a formal presentation, we introduce an abstract syntax for Promela statements that serve as input to the translator. As a result, we do not generate PVS directly, but make use of an intermediate target language *IL* which can be converted almost directly into an appropriate PVS theory. This is done to keep the core translation simple: some peephole optimizations, in particular small transformations that reduce the state space, can now be performed in a separate phase. The

conversion from *IL* to PVS is not fully elaborated but informally exemplified. The syntax of Abstract Promela Statements is given in the left column of the table below. $\overrightarrow{s}$ Denotes 0 or more and $<s>$ denotes 0 or 1 occurrences of s.

$\mathbb{L}$	: $x, y, \ldots$	local variables
$\mathbb{G}$	: $X, Y, \ldots$	global variables
V	::= $\mathbb{L} \mid \mathbb{G}$	all variables
$\mathbb{P}$	: $p, q, \ldots$	procedure names
E_{int}	: $1, x + y, \ldots$	integer expressions
E_{bool}	: $\mathsf{true}, x > 3, \ldots$	boolean expressions
E	::= $E_{\mathsf{int}} \mid E_{\mathsf{bool}}$	all expressions
SM	: LOCK, UNLOCK, WAIT, TRANS, NOTIFY	synchronization operations

PS	::= $V<[E_{\mathsf{int}}]> := E$	IL	::= ASS $V<[E_{\mathsf{int}}]> E$
	| if $\overrightarrow{G}$ <else TE> fi		| GOTO PC
	| do $\overrightarrow{G}$ <else TE> od		| SWITCH $\overrightarrow{(E_{\mathsf{bool}}, PC)}$ $<PC>$
	| $\mathbb{P}$		| CALL PC
	| atomic $\overrightarrow{PS}$		| RTN
	| $G.SM$		| ATO
G	::= $E_{\mathsf{bool}} \rightarrow TE$		| OTA
TE	::= $\langle \overrightarrow{PS}, bool \rangle$		| SM LI
		PC	::= $\mathbb{N}$
		LI	::= $\mathbb{N}$

Fig. 1. Syntax of Promela and the intermediate language IL

The abstract syntax (PS) reflects the essential statements of Promela: assignments, choices, and repetitions. The left-hand side of an assignment can be either a simple variable or the element of an array, explaining the optional selection. The boolean in the then[7] or else statement (TE) indicates whether or not the corresponding sequence of statements ends with a break. Functions in Promela are inlined. However, to reduce the size of generated code, we refrain from inlining and use simple procedure calls (no parameters, no result) instead. The synchronization operations (indicated by SM in the grammar), are not part of standard Promela. They are explained in Section 4. Note that (boolean and integer) expressions are not specified further; we can almost directly interpret these as PVS code.

The intermediate language given in the right column is largely self-explanatory. It has been designed in such a way that, on the one hand, it completely covers the intended source language, and, on the other hand, it can be interpreted directly by means of an appropriate PVS theory. *IL* resembles traditional low-level assembly languages, with the exception of the SWITCH instruction used in the translation of both choices and repetitions. This instruction takes a sequence of (boolean) expression-address pairs and randomly chooses one of the addresses

[7] There is no `then` keyword in Promela. G denotes a guard under which condition statements TE may be executed.

$$
\begin{aligned}
PS[\![v := e]\!]_\rho\ pc &= (pc+1, [\mathsf{ASS}\ v\ e]) \\
PS[\![\mathsf{if}\ gs\ eo\ \mathsf{fi}]\!]_\rho\ pc &= (pc_e, [\mathsf{SWITCH}\ gl\ el] \mathbin{+\!\!+} il_g \mathbin{+\!\!+} il_e) \\
&\quad where \quad (pc_g, gl, il_g) = \overrightarrow{G}[\![g]\!]_\rho\ pc+1\ pc_e\ pc_e \\
&\qquad\quad (pc_e, el, il_e) = <TE>[\![e]\!]_\rho\ pc_g\ pc_e\ pc_e \\
PS[\![\mathsf{do}\ gs\ eo\ \mathsf{od}]\!]_\rho\ pc &= (pc_e, [\mathsf{SWITCH}\ gl\ el] \mathbin{+\!\!+} il_g \mathbin{+\!\!+} il_e) \\
&\quad where \quad (pc_g, gl, il_g) = \overrightarrow{G}[\![g]\!]_\rho\ pc+1\ pc\ pc_e \\
&\qquad\quad (pc_e, el, il_e) = <TE>[\![e]\!]_\rho\ pc_g\ pc\ pc_e \\
PS[\![p]\!]_\rho\ pc &= (pc+1, [\mathsf{CALL}\ \rho(p)\ pc+1]) \\
PS[\![\mathsf{atomic}\ s]\!]_\rho\ pc &= (pc'+1, [\mathsf{ATO}] \mathbin{+\!\!+} il \mathbin{+\!\!+} [\mathsf{OTA}]) \\
&\quad where \quad (pc', il) = \overrightarrow{PS}[\![s]\!]_\rho\ pc+1
\end{aligned}
$$

$$
\begin{aligned}
&\overrightarrow{PS}[\![\ ]\!]_\rho\ pc = (pc, [\ ]) \qquad \overrightarrow{G}[\![\ ]\!]_\rho\ pc\ c\ e = (pc, [\ ], [\ ]) \\
&\overrightarrow{PS}[\![s : ss]\!]_\rho\ pc = (pc'', il \mathbin{+\!\!+} il') \quad \overrightarrow{G}[\![b \to s : gs]\!]_\rho\ pc\ c\ e = (pc'', (b, l) : gl', il \mathbin{+\!\!+} il') \\
&\quad where \quad (pc', il) = PS[\![s]\!]_\rho\ pc \qquad where \quad (pc', l, il) = TE[\![s]\!]_\rho\ pc\ c\ e \\
&\qquad\quad (pc'', il') = \overrightarrow{PS}[\![ss]\!]_\rho\ pc' \qquad\quad (pc'', gl, il') = \overrightarrow{G}[\![gs]\!]_\rho\ pc'\ c\ e
\end{aligned}
$$

$$
\begin{aligned}
&<TE>[\![\Diamond]\!]_\rho\ pc\ c\ e = (pc, \Diamond, [\ ]) \qquad <TE>[\![<e>]\!]_\rho\ pc\ c\ e = (pc', <el>, il) \\
&TE[\![\langle ss, b\rangle]\!]_\rho\ pc\ c\ e = (pc'+1, pc, il \mathbin{+\!\!+} [\mathsf{GOTO}\ l]) \quad where \quad (pc', el, il) = TE[\![e]\!]_\rho\ pc\ c\ e \\
&where \quad (pc', il) = \overrightarrow{PS}[\![ss]\!]_\rho\ pc \qquad TE[\![\langle [\ ], b\rangle]\!]_\rho\ pc\ c\ e = (pc, l, [\ ]) \\
&\qquad\quad l = if\ b\ then\ e\ else\ c \qquad where \quad l = if\ b\ then\ e\ else\ c
\end{aligned}
$$

Fig. 2. Translation of Promela into the intermediate language IL

corresponding to expressions evaluating to true. If none of the mentioned expressions is true, then either the else address is chosen (if available), or the instruction will block. The chosen address will become the new program counter value of the currently executing process. We describe the treatment of statements only; the translation of a complete model including procedure definitions, and local and global variable declarations is straightforward. The translation of an (abstract) Promela statement into the intermediate language *IL* is defined by the following set of mutual recursive functions $s[\![\cdot]\!]_\rho$. Here ρ is an environment mapping function names to PC values.

3 An Example: Bakery Algorithm

As an example we apply our method to Lamports bakery algorithm: a classic solution to the problem of N-mutual exclusion. The algorithm itself is analogue to the way bakeries give their customers turns by drawing a number from a machine, where the baker serves the lowest number when he is available. The algorithm listed below as a sequence of *PS* statements is essentially the same as Lamport's original. The translation of the program to *IL* is given below on the right-hand side.

```
Enter[tid] := true;                0  ASS Enter[tid] true
h := 0;                            1  ASS h 0
i := 0;                            2  ASS i 0
do i<NT                            3  SWITCH (i<NT,4) 9
 -> if h>Num[i]                    4  SWITCH (h>Num[i],5) 7
```

```
      -> h := Num[i];                5  ASS h Num[i]
       else ; fi;                    6  GOTO 7
    i := i + 1;                      7  ASS i i + 1
   else break;
od;                                  8  GOTO 3
Num[tid] := h + 1;                   9  ASS Num[tid] h + 1
Enter[tid] := false;                 10 ASS Enter[tid] false
i := 0;                              11 ASS i 0
do i<NT && !Enter[i]                 12 SWITCH(i<NT&&!Enter[i],13)(i>=NT,16)
 -> if Num[i]=0->;                   13 SWITCH(Num[i]=0,14)
       Num[i]>Num[tid]->;                 (Num[i]>Num[tid],14)
       Num[i]=Num[tid]&&i>=tid->;         (Num[i]=Num[tid]&&i>=tid,14)
    fi;
    i := i + 1;                      14 ASS i i + 1
 i >= NT -> break;
od;                                  15 GOTO 12
Num[tid] := 0;                       16 ASS Num[tid] 0
```

The complete model will execute the above code infinitely many times. In Spin, it is impossible to model check this example, because the drawn numbers are unbounded leading to an infinite state space. There exist several versions of the algorithm that use finite numbers when drawing, see Section 5.

Next we translate this *IL* program into the PVS framework. To reduce the number of different states of our model some of the statements are combined. Particularly, multiple assignments are implemented by a single instruction if they contain at most one (read/write) access to a global variable. For instance, the first three assignments of our example are combined into a single transition. For the implementation of the SWITCH statements, we will use COND expressions of PVS.

This yields the following instantiation of the `Model` skeleton. This theory also contains the proper instances of the parameters of `Threads` from Section 2. The step relation is not fully specified. For brevity only characteristic cases of this relation are given.

```
Model[NT:posnat] : THEORY BEGIN
  PC:TYPE= below(11)
  GV:TYPE= [# enter:ARRAY[below(NT) → boolean],num:ARRAY[below(NT) → nat]#]
  LV:TYPE= [# h: nat, i: nat #]
  IMPORTING Threads[NT, PC, LV, GV]

  step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool = LET pc=lv1`pc IN
  COND
    pc=0 → lv2=lv1 WITH [`local`h := 0,`local`i := 0,`pc := 1] ∧
           gv2=gv1 WITH [`enter(lv1`tid) := TRUE],
    pc=1 → COND lv1`local`i<NT → lv2=lv1 WITH [`pc := 2 ],
           ELSE → lv2=lv1 WITH [`pc := 5 ] ENDCOND ∧ gv2=gv1,
    ...
    pc=5 → lv2=lv1 WITH [`pc := 6 ] ∧
           gv2=gv1 WITH [`num(lv1`tid) := lv1`local`h + 1],
    ...
    pc=10 → lv2=lv1 WITH [`pc := 0 ] ∧ gv2=gv1 WITH [`num(lv1`tid) := 0]
```

```
ENDCOND
END Model
```

Theorems and Proofs

Proving the different properties requires (1) the instantiation of the (`noDeadlockMeasure` and `noStarvationMeasure`) measures needed for the theorems defined in Section 2, and (2) the definition of an invariant strong enough to prove that these measures indeed strictly decrease.

As to (1), we can observe the following:

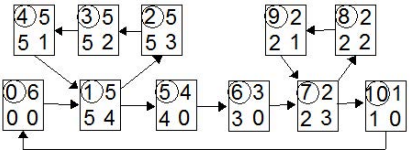
- The states themselves can be given a numerical ordering where each state in the control flow has a smaller number, with the starting state the smallest.
- If there is no transition possible to a smaller state according to the above numerical ordering, there is an increase of the local variable `i` until the maximum value of `NT` is reached.

This gives the following measure, defined using the `fsum` function.

```
noDeadlockMeasure(s:System): [nat,nat,nat] = (fsum(NT,mapBL(s)),
    fsum(NT,λ(t:TID):NT-s‘threads(t)‘local‘i), fsum(NT,mapBR(s)))
```

The ordering that is used is the lexicographical ordering on 3-tuples. The two auxiliary functions simply map the value of the program counter of a thread to a natural number. The `mapBR` values for each state are shown in the bottom right-hand corner and the `mapBL` in the bottom left-hand corner of the corresponding box (see the diagram below). The encircled numbers in the upper left-hand corner correspond to the value of the program counter. The fourth value (in the upper right-hand corner), given by `mapUR`, is used further on.

Absence of starvation means that if a process *intends* to enter the critical section, it will eventually do so. This is formulated using the program counter. Once a process has obtained a number, it arrives at the state with program counter is 6, from where it will proceed to the critical section at location 10.




```
Enter: PC = 6     Goal: PC = 10
BakeryStarvationFree: THEOREM
   ∀ (s:(InvSystem), t:TID): NoStarvation(s, t, Enter, Goal)
```

In order to prove this, we define the starvation measure based on the following system properties:

- Let $peers_t$ denote the set of processes that were choosing just after thread t has received its number.
- A thread t that draws a number will always get a larger one, except for the members of $peers_t$.
- For each thread t the size of $peers_t$ will only get smaller.
- The set of drawn numbers that are in front of the process that wants to enter the critical section, will only get smaller.

– It is possible to map the program counter to a natural number in such a way that these numbers will get smaller or the local variable `i` will decrease.

To keep track of peers we extend the global state with a *ghost/model* variable named `peers`. This value of `peers` is set to the value of the global variable `enter` at the moment the thread has drawn its number (the location with program counter 5), and the thread is removed from peer groups of other threads after leaving the critical section (indicated by the program counter value 10). This leads to some small modifications of the `step` relation, where `peers` is set at location 5 and 10:

```
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV): bool =
  LET pc=lv1‘pc IN
  COND ...
    pc=5 → lv2=lv1 WITH [‘pc := 6 ]
         ∧ gv2=gv1 WITH [‘num(lv1‘tid) := lv1‘local‘h + 1
                        , peers(lv1‘tid) := gv1‘enter ],
    ...
    pc=10 → lv2=lv1 WITH [‘pc := 0 ]
          ∧ gv2=gv1 WITH [‘num(lv1‘tid) := 0, peers := remove(peers,lv1‘tid)]
```

The `noStarvationMeasure` introduced in Section 2 can now be defined as follows. The corresponding ordering is the lexicographical ordering on 5-tuples.

```
b2N(b:bool): nat = IF b THEN 1 ELSE 0 ENDIF

noStarvationMeasure(s:System,t:TID):[nat,nat,nat,nat,nat]
= ( fsum(NT, b2N o s‘threads(t)‘peers)
  , fsum(NT, λ(t2:TID): LET nr = s‘global‘num IN
       b2N (nr(t2) ≠ 0∧ (nr(t2),t2)<(nr(t),t)∧ ¬s‘threads(t)‘peers(t2)))
  , fsum(NT, mapUR(s))
  , fsum(NT, λ(t2:TID): NT-s‘threads(t2)‘local‘i)
  , fsum(NT, mapBR(s)) )
```

An interesting safety property of our system is, of course, *mutual exclusion*: it should be impossible for two processes to be in the critical section at the same time. More concretely, when a process has 10 as its program counter, it will be the only one.

```
inCS(s:System,t:TID): bool = s‘threads(t)‘pc = 10

MutualExclusion(s:System) : bool =
  ∀(t1:TID): inCS(s,t1) ⇒ ∀(t2:TID): inCS(s,t2) ⇒ t1=t2
```

Before proving this property, we first explain some of our program invariant definitions. At the beginning of the proof process it may not be entirely clear what invariants will be needed. Therefore, these invariants are progressively constructed.

The transition relationship defined in `step` generates type correctness conditions. For instance, when the `num` array is indexed, the index may not exceed the total number of processes. This leads to the following property of the first invariant of the system:

```
numAccessed(pc:PC): bool = pc = 2 ∨ pc = 3 ∨ pc = 7 ∨ pc = 8

Prop1(s:System): bool =
  ∀(t:TID): numAccessed(s`threads(t)`pc) ⇒ s`threads(t)`local`i < NT
```

The most important invariant stipulates that whenever a process is in the loop where it compares the numbers drawn by each thread (indicated by the predicate `comparing`), then for all threads it has already examined, the current thread is greater according to the lexicographical ordering on `(num(t),t)`. In the same part of the program execution it also holds that if a thread is in the peer group, it cannot have the `enter` flag set. This is expressed by the property of the second invariant.

```
comparing(pc:PC): bool = pc = 7 ∨ pc = 8 ∨ pc = 9

Prop2(s:System): bool = ∀(t:TID):
  LET ts=s`threads(t) IN comparing(ts`pc) ⇒
    ∀ (k:TID): k ≠ t ∧ (k < ts`local`i ∨ k = ts`local`i ∧ ts`pc = 9) ⇒
      (s`global`num(t),t) < (s`global`num(k),k) ∧
      ts`peers(k) ⇒ ¬ s`global`enter(k))
```

All that has to be established further is that if a process enters the comparison loop, it will do so only if it has a number greater than all the numbers already given out. The only exception is made for processes that are in the peer group. In order to prove this, we need some extra invariants that are pretty straightforward. Their PVS code is left out for brevity.

- Processes can be entering only in states 0,1,2,3,4, and 5.
- After setting the peer group, each process is always part of its own peer group.
- At the beginning (states 0,1,2,3, and 4) the peer group is empty. At these states also the `num` value is 0; otherwise greater than 0.
- Finally, in state 10, `i` is always equal to `NT`.

The invariants guarantee that when a process proceeds to the critical section (location 10), all the other processes have larger numbers. This enables the proof of the measures. The safety property also follows directly from the invariant combined with the fact that the lexicographical ordering is well founded and has only one smallest element. The proofs of the theorems proceed by a case distinction on the value of the program counter, creating a symbolic execution of the algorithm. For all the possible cases only instances of the `fsum_diff` lemma (Section 2) and the invariant are needed to discharge all the proof goals. The simple structure of the proofs makes it feasible to prove larger algorithms, like the reentrant readers writers algorithm given in the next section, although their proofs end up being quite large. The proof file for the latter program is more than 20,000 lines. Despite its size, the proof itself took a PhD student a couple of weeks to complete.

4 Framework with Thread Synchronisation

Many concurrent algorithms are based on locking primitives that modern operating systems usually support. These primitives are not available in standard Promela but are added to the framework. In principle we could have modeled these locking primitives in Promela (like the bakery algorithm) and translated this model to PVS using the procedure as described in the previous sections. However, it appears to be more convenient to extend Promela with special synchronization constructs[8], and use a shallow embedding by also incorporating basic locks into our PVS framework.

Incorporating Locking Primitives

The idea of the basic locks is similar to, for example, the synchronization mechanism of Java. Shared resources are protected by locks. If a process wants exclusive access to these resources it performs a *lock* operation on the corresponding lock. Releasing a resource is done by calling *unlock*. Besides, processes should be able to relinquish their turn temporarily by means of a *wait* command and also be able to wake other processes up using *notify*. Another primitive is *transfer*, which allows the process to explicitly hand over the execution privilege to the first waiting process. This operation plays an essential role in our algorithm in order to guarantee absence of starvation. Furthermore, we have built in basic support for implementing atomic statements. In Promela one can enforce a sequence of statements to be non-interruptible by placing these statements in an atomic context. Although these atomic statements can be simulated in PVS by locks, we prefer to represent them more efficiently by a separate system extension. It suffices to use a single global boolean to indicate whether the currently executing process is interruptible. This leads to the following adapted `Threads` theory.

```
Threads[NT, NL:posnat, PC, LS, GS:TYPE] : THEORY BEGIN
  TID : TYPE = below(NT)
  LID : TYPE = below(NL)
  TState, Threads:  TYPE /* as before */
  LState: TYPE = [# lockedBy: lift[TID],  blocked, waiting: list[TID] #]
  Locks : TYPE = [ LID → LState]
  System: TYPE = [# threads:Threads,locks:Locks,
                    atomic:bool,current:TID,global:CV #]
END Threads
```

The new theory parameter `NL` denotes the number of locks appearing in the program, also used to identify each lock by a `LID`. This also explains why the lock variables of our intermediate language *IL* were represented by natural numbers; see Section 2. The system state now contains a variable `locks` holding the `LState` of each lock. This state indicates whether the lock is occupied (in which case

[8] In fact, we've already anticipated on this extension in the definition of the abstract Promela syntax; see Section 2.

`lockedBy` refers to the corresponding thread) and maintains two queues for holding the blocked and waiting processes. The boolean variable `atomic` indicates that no context switch is allowed. The lock operations are defined as a separate PVS theory. As an example the implementation of the `transfer` operation is given.

```
LOCK[NT, NL:posnat,PC:TYPE,LV:TYPE,GV:TYPE]: THEORY BEGIN
IMPORTING Threads[NT, NL, PC, LV, GV]
LSystem(lid:LID): TYPE = { s: System | s`locks(lid)`lockedBy = up(s`current) }

lock  (lid:LID)(s:System):       System
unlock(lid:LID)(s:LSystem(lid)): System

transfer(lid:LID)(s:{ s1: LSystem(lid) | cons?(s1`locks(lid)`waiting) } ):
  LSystem(car(s`locks(lid)`waiting)) = LET ls = s`locks(lid) IN
    s WITH [`threads(car(ls`waiting))`status := Running,
            `locks(lid)`lockedBy  := up(car(ls`waiting)),
            `locks(lid)`waiting   := cdr(ls`waiting)]

wait  (lid:LID)(s:LSystem(lid)): System
notify(lid:LID)(s:LSystem(lid)): System
END LOCK
```

As usual, a process can only perform an unlock, wait, transfer or notify if it is the owner of the lock. This requirement is expressed in the dependent type `LSystem`. Moreover, transfer has the additional requirement that it is only allowed if the waiting queue of the corresponding lock is not empty. Again, this is enforced by defining the type of the system parameter dependently.

In our framework, a thread can only access its own state and the global variables of the system; see the `step` relation. However, a thread executing a synchronization operation may indirectly affect other system components. It may even change the status of other threads. Instead of passing the complete system state to the (local) step relation, we have implemented these 'system calls' by extending the result of `step` with a function of type `[System → System]`. This yields the adjusted type of `step`, and the implementation of `interleave`, using an auxiliary function `sysStep`:

```
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV,sc:[System → System]): bool

sysStep(s1: (invSystem), s3:System):bool= ∃(s2:System, sc:[System → System]):
  step(currThread(s1),s1`global)(currThread(s2),s2`global,sc) ∧ s3 = sc(s2)

interleave(s1:(invSystem),s2:System):bool= PredSys(s2)(s1) ∧
 LET ct=s2`current IN
  IF s1`atomic THEN ct=s1`current ∧ sysStep(s1,s2)
  ELSE s1`threads(ct)`status=Running ∧ sysStep(s1 WITH [`current:=ct],s2)
  ENDIF
```

Example: Reentrant Read-Write

A more complex synchronization mechanism involves processes that acquire access to resources for both reading and writing: the classic readers-writers problem. Several kinds of solutions exist. Here, we will consider a reentrant *read-write* locking mechanism that employs writers preference. A thread can acquire the lock multiple times, even when the thread has not fully released the lock: locking can be *reentrant*. Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and occurring less frequently. The main disadvantage of this choice is that it can result in *reader starvation*: when there is always a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed.

Specifying the entire algorithm would take too much space. The part that shows the Promela version of *readLock* used for acquiring the lock for reading is given below. As one can see, the locks appearing in this program are represented by variable names. In our translation these names will be mapped to natural numbers. This is not included in the translation function, but can be added straightforwardly (e.g. by parameterizing the translation with an additional environment that performs this mapping). The result of the translation is on the right-hand side of the listing.

```
Mutex.LOCK;                        0 LOCK 0
if Count[tid]=0 ->                 1 SWITCH (Count[tid]=0,2) 9
 do CurrWr!=NT||WaitWr> 0 ->       2 SWITCH (CurrWr!=NT||WaitWr>0,3) 7
      WaitRe := WaitRe + 1;        3 ASS WaitRe (WaitRe + 1)
      Mutex.WAIT;                  4 WAIT 0
      WaitRe := WaitRe - 1;        5 ASS WaitRe (WaitRe - 1)
    else break;
 od;                               6 GOTO 2
 ThrCount := ThrCount + 1;         7 ASS ThrCount (ThrCount + 1)
else ; fi;                         8 GOTO 9
Count[tid] := Count[tid] + 1       9 ASS Count[tid] (Count[tid] + 1)
Mutex.UNLOCK;                      10 UNLOCK 0
                                   11 RTN
```

The part of the `step` relation that corresponds to this program fragment is shown below. In the complete model, the values of the program counter depend on the exact location of this function in the original program, which may be different from the given values.

```
Model[NT:posnat] : THEORY BEGIN
PC : TYPE = below(8)
GV: TYPE = [# count:ARRAY[below(NT)→nat],CurrWr,WaitWr,WaitRe,ThrCount:nat #]
LV: TYPE = [# rNest, wNest, maxLocks: nat #]
step(lv1:TState,gv1:GV)(lv2:TState,gv2:GV,sc:SysCall): bool =
  LET pc = lv1`pc IN
  COND
    pc=0 → lv2 = lv1 WITH [`pc := 1] ∧ gv1 = gv2 ∧ sc = lock(0),
    pc=1 → COND gv1`count(lv1`tid)=0 → lv2 = lv1 WITH [`pc := 2],
```

```
            ELSE → lv2 = lv1 WITH [‘pc := 6] ENDCOND∧ gv1 = gv2 ∧ sc=id,
    ...
    pc=7 → lv2 = lv1 WITH [‘pc := lv1‘rtn] ∧ gv2 = gv1 ∧ sc=id
  ENDCOND
END Model
```

The complete Promela model also contains a few ghost variables (`rNest`, `wNest` and `maxLocks`) that limit the number of nested locks a process is allowed to use. If no such limit was imposed, it would be impossible so show absence of starvation.

Comparing the ad hoc proof given in our earlier paper [15] with the proof that is possible using the framework, we find only small differences. There are several advantages of using the framework and the iterating invariant technique. Firstly, the model does not have to be translated by hand, decreasing the likelihood of a translation error. Secondly, using the iterating invariant technique, the partial proofs need less readjustments in order to accommodate changes to the invariants. And thirdly, the framework provides free lemmas that had to be proven manually in the earlier ad hoc proof.

The invariant needed to prove the theorems is large, but revolves around the relationships of the possible values of the variables used in the program at certain points in their execution past, similar to what was done in Section 3. The PVS model that was used in the concrete proof was adjusted in order to reduce the number of possible state transitions. This manually performed optimization was based on the observation that if a model uses a single lock and all accesses to global variables are synchronized (which is the case in our example) one can use the atomic instead of the (first) lock of the system. This means that a process will never have status `Blocked`. The code for the `wait`, `notify` and `transfer` needs to be adjusted in order to obtain the correct behavior[9].

5 Related Work

Providing support for domain specific theorem proving environments within general theorem provers in the area of state transition systems is present in TAME [1]. However, this tool set offers tactics and templates to construct proofs using PVS and is geared towards proving properties of SCR, timed and I/O automata.

A translator between different formal specification languages is VeriTech [6]. It uses an intermediate notation to translate from and into different languages, among others, Promela. PVS is not supported.

Basten and Hooman [3] provide an indirect approach to proof support for models that originate from model checkers. They first define the semantics of process algebra in PVS and then investigate the difference in proving behavior depending of the kind of embedding that is used.

For the purpose of developing consistent requirement specifications, De Groot [4] introduces a framework that is used for the transformation of transition

[9] The full PVS files of both examples can be found at `http://www.cs.ru.nl/S.Smetsers/frameworkexamples`

systems (given as specifications in the model checker Uppaal [8]) to specifications in PVS.

An embedding of Promela lite (a Promela like language) is given by Ripon and Miller [12]. However, they use this embedding to prove lemmas concerning symmetry detection and not to prove properties of specific models.

For finite state models, translating from the model checker to the theorem prover can be circumvented by using the PVS built-in model checker [9].

Pantelic et al. [11] combine model checking and theorem proving to analyze the classic readers-writers problem. However, the authors start from a tabular specification of the solution rather than from a real algorithm. This tabular specification is translated straightforwardly into SPIN and PVS. Some properties (like safety and clean completion) can be derived semi-automatically.

The bakery algorithm is a classical solution to the mutual exclusion problem. In Lamport's original version the numbers drawn by the customers can grow infinitely, leading to an unlimited state space which makes it unsuited for being model checked directly. However, several modifications have been proposed to restrict the drawn numbers also leading to a finite state space [2]. The advantage of using a theorem prover is, of course, that there are no limitations on the values being used. This made it possible to work directly with the original unbounded algorithm.

6 Conclusions and Future Work

In this paper we have presented a framework for constructing formal correctness proofs of Promela models. The framework is restricted to concurrent processes that communicate via global variables. It enables reasoning about basic synchronization protocols, such as the bakery algorithm, as well as more complex synchronization mechanisms, such as the reentrant read/write locks provided by the Qt library. The framework provides basic theories and proof support for constructing proofs of fundamental concurrency properties, such as (the absence of) deadlock and starvation. Formulating these properties is structured by introducing suitable abstract functions and predicates that are instantiated based on the original model. Proving actually boils down to constructing an appropriate invariant and to showing that this invariant indeed holds for the constructed state transition relation.

Our future plans are to extend the framework in such a way that it covers the complete Promela language, e.g. by adding constructs for modeling message passing. Furthermore, the proof process can be partially automated by defining appropriate PVS-tactics to avoid repeating certain sequences of proof steps. Also, many auxiliary mappings of program counters to natural numbers that were needed to define proper measures, can be generated automatically. Another venue to explore is to establish the soundness of the translation in order to make sure that the generated PVS conforms to the semantics of the Promela code.

References

1. Archer, M.: TAME: Using PVS strategies for special-purpose theorem proving. Annals of Mathematics and Artificial Intelligence (2000)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
3. Basten, T., Hooman, J.: Process Algebra in PVS. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 270–284. Springer, Heidelberg (1999)
4. de Groot, A.: Practical Automaton Proofs in PVS. PhD thesis, Radboud University Nijmegen (2008)
5. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
6. Katz, S.: Faithful Translations among Models and Specifications. In: Oliveira, J., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 419–434. Springer, Heidelberg (2001)
7. Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. Commun. ACM 17(8), 453–455 (1974)
8. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
9. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: 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)
10. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 747–752. Springer, Heidelberg (1992)
11. Pantelic, V., Jin, X.-H., Lawford, M., Parnas, D.L.: Inspection of concurrent systems: Combining tables, theorem proving and model checking. In: Software Engineering Research and Practice, pp. 629–635 (2006)
12. Ripon, S., Miller, A.: Verification of symmetry detection using pvs. ECEASST 35 (2010)
13. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (March 2005)
14. van Gastel, B., Lensink, L., Smetsers, S., van Eekelen, M.: Reentrant Readers-Writers: A Case Study Combining Model Checking with Theorem Proving. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 85–102. Springer, Heidelberg (2009)
15. van Gastel, B., Lensink, L., Smetsers, S., van Eekelen, M.: Deadlock and Starvation Free Reentrant Readers-Writers. Sci. Comput. Program. 76(2), 82–99 (2011)

A UTP Semantics of pGCL as a Homogeneous Relation

Riccardo Bresciani and Andrew Butterfield*

Foundations and Methods Group,
Trinity College Dublin,
Dublin, Ireland
{bresciar,butrfeld}@scss.tcd.ie

Abstract. We present an encoding of the semantics of the probabilistic guarded command language (pGCL) in the Unifying Theories of Programming (UTP) framework. Our contribution is a UTP encoding that captures pGCL programs as predicate-transformers, on predicates over probability distributions on before- and after-states: these predicates capture the same information as the models traditionally used to give semantics to pGCL; in addition our formulation allows us to define a generic choice construct, that covers conditional, probabilistic and non-deterministic choice. As an example we study the Monty Hall game in this framework.

1 Introduction

The Unifying Theories of Programming (UTP) research activity seeks to bring models of a wide range of programming and specification languages under a single semantic framework in order to be able to reason formally about their integration [12,5,2,22]. A success in this area has been the development of the *Circus* language [21], which is a fusion of Z and CSP, with a UTP semantics, providing specifications using a "state-rich" process algebra along with a refinement calculus; recent extensions to *Circus* have included timed [23] and synchronous [7] variants. Recent interest in aspects of the POSIX filestore case study in the Verification Grand Challenge [6] has led us to consider integrating probability into UTP, with a view to eventually having a probabilistic variant of *Circus*.

UTP is based on (state-)predicate transformers, whereas probabilistic models typically involve distributions over states, and so the best way to integrate probability into the UTP framework is not obvious. This paper presents first steps in constructing a theory of probabilistic programs that is expressed using predicate-transformers[1]. The focus here is on a UTP theory that captures the semantics of the probabilistic guarded command language (pGCL) [15], by

* The present work has emanated from research supported by Science Foundation Ireland grant 08/RFP/CMS1277 and, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre.

[1] So probabilistic programs are predicates too (with apologies to C.A.R. Hoare [11]).

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 191–205, 2012.

means of predicates involving a homogeneous relation among distributions over states.

This paper is structured as follows: we describe the background to both UTP and pGCL (§2); discuss the motivation for and technical details of our observable variables (§3); give the semantics of pGCL in our framework (§4); and conclude (§5).

2 Background

2.1 UTP

UTP follows the key principle that "programs are predicates" [11]: theories in UTP are expressed as second-order predicates over a pre-defined collection of free observation variables, referred to as the *alphabet* of the theory. The predicates are generally used to describe a relation between a before-state and an after-state, the latter typically characterised by dashed versions of the observation variables. For example, a program using two variables `x` and `y` might be characterised by having the set $\{x, x', y, y'\}$ as an alphabet, and the meaning of the assignment `x := y+3` would be described by the predicate

$$x' = y + 3 \wedge y' = y.$$

In effect UTP uses predicate calculus in a disciplined way to build up a relational calculus for reasoning about programs.

In addition to observations of the values of program variables, often we need to introduce observations of other aspects of program execution via so-called auxiliary variables. So, for example, in order to reason about total correctness, we need to introduce boolean observations that record the starting (ok) and termination (ok') of a program, resulting in the above assignment having the following semantics:

$$ok \Rightarrow ok' \wedge x' = y + 3 \wedge y' = y$$

(if started, it will terminate, and the final value of `x` will equal the initial value of `y` plus three, with `y` unchanged).

A problem with allowing arbitrary predicate calculus statements to give semantics is that it is possible to write unhelpful predicates such as $\neg ok \Rightarrow ok'$, which describes a "program" that must terminate when not started. In order to avoid assertions that are either nonsense or infeasible, UTP adopts the notion of "healthiness conditions" which are monotonic idempotent predicate transformers whose fixpoints characterise sensible (healthy) predicates. Collections of healthy predicates typically form a sub-lattice of the original predicate lattice under the reverse implication ordering [12, Chp. 3]. Key in UTP is a general notion of program refinement as the universal closure of reverse implication[2]:

$$S \sqsubseteq P \;\hat{=}\; [P \Rightarrow S]$$

[2] Square brackets denote universal closure, *i.e.* $[P]$ asserts that P is true for all values of its free variables.

$$
\begin{aligned}
\mathtt{wp.abort}.PostE &\mathrel{\hat{=}} 0 \\
\mathtt{wp.skip}.PostE &\mathrel{\hat{=}} PostE \\
\mathtt{wp}.(\underline{x} := \underline{e}).PostE &\mathrel{\hat{=}} PostE[\underline{e}/\underline{x}] \\
\mathtt{wp}.(prog_1; prog_2).PostE &\mathrel{\hat{=}} \mathtt{wp}.prog_1.(\mathtt{wp}.prog_2.PostE) \\
\mathtt{wp}.(prog_1 \lhd c \rhd prog_2).PostE &\mathrel{\hat{=}} (\mathtt{wp}.prog_1.PostE)|_c + (\mathtt{wp}.prog_2.PostE)|_{\neg c} \\
\mathtt{wp}.(prog_1 \sqcap prog_2).PostE &\mathrel{\hat{=}} \min\{\mathtt{wp}.prog_1.PostE, \mathtt{wp}.prog_2.PostE\} \\
\mathtt{wp}.(prog_1 \ {}_p\!\oplus prog_2).PostE &\mathrel{\hat{=}} p \cdot \mathtt{wp}.prog_1.PostE + (1-p) \cdot \mathtt{wp}.prog_2.PostE
\end{aligned}
$$

Fig. 1. wp-semantics of pGCL, adapted from [15, p. 26]
Notation: $[\underline{e}/\underline{x}]$ denotes free occurrences of $\underline{x}$ replaced by $\underline{e}$; $|_c$ denotes expectation limited to states satisfying c.

Program P refines S if for all observations (free variables) S holds whenever P does.

The UTP framework also uses Galois connections to link different languages and theories with different alphabets [12, Chp. 4], and often these manifest themselves as further modes of refinement.

2.2 pGCL

pGCL extends GCL with an additional language construct, namely that of probabilistic choice $prog_1 \ {}_p\!\oplus prog_2$, denoting a statement that executes $prog_1$ with probability p, and $prog_2$ with probability $(1-p)$ [17,15,16,19].

In [15] pGCL is given a semantics that generalises Dijkstra's weakest precondition semantics to what they term a *weakest pre-expectation semantics*.

An expectation is a function that assigns a weight (a non-negative real number) to program states: it is therefore a random variable. An expectation corresponding to a predicate can be defined as a random variable that maps a state to 1 if it satisfies the predicate and to 0 otherwise. Arithmetic operators and relations are extended pointwise to expectations, as is multiplication by a scalar.

If $PostE$ is a (post-)expectation after running program $prog$, then $\mathtt{wp}.prog.PostE$ is the corresponding weakest[3] (pre-)expectation before the program runs: for each state it returns the minimum expected final weight.

The weakest pre-expectation semantics for pGCL is shown in Figure 1. The key features to note in this semantics are that probabilistic choice is the obvious weighting of its alternatives' expectations, whereas demonic choice returns the pointwise minimum.

Non-determinism is crucial in order to define a sensible refinement relation[4]:

$$spec \sqsubseteq prog \mathrel{\hat{=}} \forall PostE \bullet \mathtt{wp}.spec.PostE \leq \mathtt{wp}.prog.PostE$$

[3] One expectation is weaker than another if for all states it returns at most the same weight — it is the $\leq$ relation lifted pointwise.

[4] We have definition of refinement that matches that of pGCL, which we do not discuss in this paper.

A program *prog* refines a specification *spec* if the minimum expected weight for each state after *prog* has run is at least as much as we would get after *spec* has run.

An alternative model for pGCL is one that sees a program as a function from initial states to sets of probability distributions over the state space [10,15]

$$S \to \mathbb{P}(S \to [0,1])$$

Programs with semantics of this form can be sequentially composed using Kleisli composition (See Appendix A), which can be interpreted as lifting the semantic domain to relations between before- and after-distributions $((S \to [0,1]) \leftrightarrow (S \to [0,1]))$ and then using relational composition [15, Chp. 5]. It is this form that has formed the basis for most of the prior work encoding pGCL semantics in UTP (see Section 2.3).

2.3 Probabilistic UTP

There has already been a certain amount of work looking at encoding probability in a UTP setting. He and Sanders have presented an approach to unification of probabilistic choice with standard constructs [9], and this work provides an example of how the laws of pGCL could be captured in UTP as predicates about program equivalence and refinement. However only an axiomatic semantics was presented, and the laws were justified via a Galois connection to an expectation-based semantic model.

Sanders and Chen then explored an approach that decomposed demonic choice into a combination of pure probabilistic choice and a unary operator that accounted for demonic behaviour [3]. There they commented on the lack of a satisfactory UTP theory, where probabilistic and demonic choice coexist.

A probabilistic BPEL-like language has recently been described by He [8] that gives a UTP-style semantics for a web-based business semantics language. This language is GCL with extra constructs to handle probabilistic choice and compensations and coordination operators, including exception handling. The UTP model that is developed does not relate before- and after-variables of the same type, but instead uses predicates to encode a relationship between an initial state and a final probability distribution over states.

What all the treatments above have in common is that the UTP predicates relate an initial program variable state (σ) to a final probability distribution (δ') over states, so the relation is not homogenous. This complicates the definition of sequential composition (which has to involve some form of Kleisli composition) and also makes building links to homogeneous UTP theories more difficult. The collection of theories surrounding *Circus* are all based on homogeneous relations (before- and after-observations of the same type). This means that all of these theories have uniform definitions of many common language features, such as sequential composition. This is the main motivation for the development of a homogeneous UTP theory of pGCL.

In this paper, we present a UTP encoding of pGCL semantics as a homogenous relation between probability distributions over the set of possible states, relating a before-distribution (δ) to an after-distribution (δ').

3 Observing Distributions

In UTP we usually talk about variables and the values they map to, so a naïve (and quite straightforward) generalization to handle probability would simply consist of mapping variables to distributions over their values, and that would lead our semantic model to be a mapping from variables to value-distributions:

$$Var \to (Val \to [0..1])$$

Although such an easy generalization may look appealing, it fails to give the appropriate semantics. The reason for this is that many properties of interest depend on an "entanglement" among the variables and this is not captured by the above model.

In order to retain all of the necessary information, we have to consider distributions relating entire program states to a corresponding weight, and we have the form:

$$\delta, \delta' : (Var \to Val) \to [0..1]$$

Later on we will see how these can be related to the expectations being transformed by the semantic model of pGCL already described.

This need to bundle all the information regarding program variables into a single observation is not a major constraint. In fact in many presentations of *Circus*-like languages it is often the convention to model program variable values with a single state observation $\sigma : Var \to Val$, and to treat it as a finite map, which simplifies the treatment of alphabets to a considerable degree: our approach here towards pGCL is analogous. For the purposes of this paper, to keep things simple and to allow us to focus on the key concepts, we shall assume that the set of program variables is finite and fixed, and all states are total functions on this variable set.

We now look at some mathematical preliminaries regarding distributions.

Generally speaking we can define a distribution as a function χ mapping states to real numbers[5], and define its *weight* as:

$$\|\chi\| \triangleq \sum_{\sigma \in \mathrm{dom}\, \chi} \chi(\sigma)$$

We will be working with the following two sub-classes:

- a *weighting distribution* π has the property that for every state σ we have $\pi(\sigma) \leq 1$ — we define two particular weighting distributions, ϵ and ι, as the ones mapping every state to 0 and 1 respectively. There is no limit for the distribution weight;

[5] In other words, it is a real-valued random variable — pGCL expectations are therefore distributions with the additional constraint of having only non-negative values.

– a *probability distribution* δ is a weighting distribution with the additional property that $\|\delta\| \leq 1$.

We will use the term *sub-distribution* to refer to a probability distribution where $\|\delta\| < 1$ and the term *full distribution* to refer to a probability distribution where $\|\delta\| = 1$.

Generally speaking, it is possible to operate on distributions by lifting pointwise operators such as addition, multiplication and multiplication by a scalar; analogously we can lift pointwise all traditional relations and functions on real numbers.

In the case of pointwise multiplication, it is interesting to see it as a way of "re-weighting" a distribution: we have a particular interest in the case when one of the operands is a weighting distribution π, as we will use this operation to give semantics to choice constructs. We opt for a postfix notation to write this operation, as this is an effective way of marking when pointwise multiplication happens in the operational flow: for example if we multiply the probability distribution δ by the weighting distribution π, we will write this as $\delta\langle\pi\rangle$.

Given a condition (predicate on state) c, we can define the weighting distribution that maps every state where c evaluates to `true` to 1, and every other state to 0. The value of each state can be seen as the boolean value of c in that state multiplied by 1, so we overload the above notation and note this distribution as $\iota\langle c\rangle$[6]. In general whenever we have the multiplication of a distribution by $\iota\langle c\rangle$, we can use the postfix operator $\langle c\rangle$ for short, instead of using $\langle\iota\langle c\rangle\rangle$.

It is worth pointing out that if we multiply a probability distribution δ by $\iota\langle c\rangle$, we obtain a distribution whose weight $\|\delta\langle c\rangle\|$ is exactly the probability of being in a state satisfying c.

3.1 Assignment

The challenge we now face is describing how assignment, which is very much oriented towards individual variables, is given a semantics in terms of a distribution that involves complete entanglement of those variables. In effect an assignment statement `x:=e` involves a partial entanglement of variable x with the variables mentioned in e. In general as we build up larger programs using single assignment as the basic component we observe an increasing degree of entanglement, which can often be captured as an appropriate simultaneous assignment, so we shall work at this level here.

Given a simultaneous assignment $\underline{\mathtt{v}}\mathtt{:=}\underline{\mathtt{e}}$, where underlining indicates that we have lists of variables and expressions of the same length, we denote its effect on an initial probability distribution δ by $\delta\{\!|\underline{e}/\underline{v}|\!\}$. The postfix operator $\{\!|\underline{e}/\underline{v}|\!\}$ reflects the modifications introduced by the assignment — the intuition behind this, roughly speaking, is that all states σ where the expression $\underline{e}$ evaluates to the same value $\underline{val} = \mathrm{eval}_\sigma(\underline{e})$ are replaced by a single state $\sigma' = (\underline{v} \mapsto \underline{val})$ that maps to a probability that is the sum of the probabilities of the states it replaces.

$$(\delta\{\!|\underline{e}/\underline{v}|\!\})(\sigma') \;\widehat{=}\; \Sigma_{\{\sigma \mid \sigma' = \sigma \dagger \{\underline{v} \mapsto \mathrm{eval}_\sigma(\underline{e})\}\}}\, \delta(\sigma)$$

[6] If we see c as a predicate, then $\iota\langle c\rangle$ is the corresponding expectation.

$$
\begin{aligned}
\texttt{abort} &\mathrel{\hat{=}} \texttt{true} \\
\texttt{skip} &\mathrel{\hat{=}} \delta' = \delta \\
\underline{x} := \underline{e} &\mathrel{\hat{=}} \delta' = \delta\{\!|\underline{e}/\underline{x}|\!\} \\
A;B &\mathrel{\hat{=}} \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta') \\
A \lhd c \rhd B &\mathrel{\hat{=}} \exists \delta_A, \delta_B \bullet A(\delta\langle c\rangle, \delta_A) \wedge B(\delta\langle \neg c\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
A \;{}_p\!\oplus\; B &\mathrel{\hat{=}} \exists \delta_A, \delta_B \bullet A\big(p \cdot \delta, \delta_A\big) \wedge B\big((1-p) \cdot \delta, \delta_B\big) \wedge \delta' = \delta_A + \delta_B
\end{aligned}
$$

Fig. 2. UTP Semantics for the deterministic constructs of pGCL

Here we treat the state as a map, where † denotes map override; this operator essentially implements the concept of "push-forward" used in measure theory, and is therefore a linear operator.

Assignment preserves the overall weight of a probability distribution if $\underline{e}$ can be evaluated in every state, and if not the assignment returns a sub-distribution, where the "missing" weight accounts for the assignment failing on some states (this failure prevents a program from proceeding and causes non-termination).

These are the most significant elements and constructs that characterise our framework: this has been a presentation from a fairly high level, and it should have provided the reader with a working knowledge of the framework; a formal and rigorous definition of the elements presented so far is beyond the scope of this paper and can be found in [1], along with some soundness proofs.

4 UTP Semantics of pGCL

We are going to express the semantics of pGCL in UTP using predicates based on a homogeneous relation among probability distributions: we will see programs as *distribution-transformers*, as they change a before-distribution δ into an after-distribution δ'.

This semantics can be related to the relational semantics and the `wp`-semantics of pGCL. [1]

4.1 Deterministic Constructs

The semantic definitions for all deterministic constructs of pGCL are listed in Figure 2 and we will now proceed to discuss each one.

The failing program `abort` is represented by the predicate `true`, which captures the fact that it is maximally unpredictable. Program `skip` makes no changes and immediately terminates.

Assignment $\underline{x} := \underline{e}$ remaps the distribution as has already been discussed in the previous section 3.1.

Sequential composition $A; B$ is characterised by the existence of a "mid-point" distribution that is the outcome of the first program, and is then fed into the second.

We characterise conditional choice $A \lhd c \rhd B$ by using the condition (and its negation) to filter the left- and right-hand programs appropriately, and we

simply sum the (now effectively disjoint) distributions. Probabilistic choice $A \,{}_p\!\oplus\, B$ simply uses the probability and its complement to scale the distributions for merge — this definition preserves all usual properties. In effect the predicate is only satisfied by any combination of left and right distributions that is pointwise larger than the minimum of both.

It is possible to build an isomorphism to relate the semantics of deterministic constructs described so far to the semantics proposed by Kozen [13,14] for probabilistic programs.

4.2 Non-deterministic Choice

We are now going to address non-determinism. According to the relational semantics of pGCL from [10,15], which sees programs as relations from a state σ to a probability distribution, we have that[7]

$$(A \sqcap B).\sigma = \cup_{p \in [0..1]} (A \,{}_p\!\oplus\, B).\sigma$$

If a demonic choice is performed on a state, the set of resulting distributions is that containing all possible distributions resulting from a probabilistic choice with probability p varying in the range $[0..1]$.

Seeing this, one could (reasonably?) expect the following definition for non-deterministic choice in our framework:

$$A \sqcap B \stackrel{?}{=} \exists p \bullet A \,{}_p\!\oplus\, B$$

However this definition does not work. In particular, with the above definition, we can prove the following (which is most definitely not a law of pGCL) :

$$(A \sqcap B); (C \,{}_p\!\oplus\, D) = (C \,{}_p\!\oplus\, D); (A \sqcap B) \qquad (!?)$$

It describes a demonic choice that is both history-aware, and *prescient*, and this latter ability to look into the future is undesirable, and infeasible.

The key point to note is that the first statement is talking about the possible resulting distributions starting from one single state, whereas this last definition considers all possible starting states. As a result the set of after-distributions that satisfy this definition of demonic choice (for a given before-distribution) is strictly smaller then the set of after-distributions satisfying the first statement. We can easily see this by considering that if we take the Kleisli lifting of $(A \sqcap B).\sigma$ for σ ranging over the whole state space. We obtain some after-distributions which are the result of composing programs where p is not constrained to be constant over all states, and these cases are ruled out in the proposed definition by the single quantification of p valid for all states.

The solution is therefore to take a weighting distribution π, use it with its complementary distribution $\bar{\pi} = \iota - \pi$) to weight the distributions resulting from the left- and right-hand side respectively, and existentially quantify it:

$$A \sqcap B \;\;\hat{=}\;\; \exists \pi, \delta_A, \delta_B \bullet A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B$$

[7] Here we are using the point notation for function application, as in [15].

In this way π can range over the set of weighting distributions, and the set of after-distributions satisfying this second definition coincides with the set obtainable via the Kleisli lifting mentioned above.

A few more comments: usually we talk about demonic non-determinism when we are expecting the worst-case behaviour, to model something that behaves "as bad as it can" for any desired outcome, nevertheless our definition of non-deterministic choice *per se* mandates no such behaviour: depending on the context where it is used (*e.g.* in a framework where refinement is defined in a similar way as for pGCL), this behaviour shows up but it is not intrinsic to the definition — from this perspective we have a similar situation as in the relational model of [10,15].

We can see that non-determinism yields a many-to-many relation: a program can be seen as a relation that associates probability before-distributions with non-disjoint sets of probability after-distributions.

The non-deterministic choice operator is idempotent according to our definition, in accordance with the pGCL semantics we take as a guide. Although some definitions of demonic choice in the literature have this property, there are others where this property does not hold: for example if on both sides we have the same program containing a probabilistic choice and this choice is resolved independently on each side *before* the non-deterministic choice is performed, then idempotency does not hold. Nonetheless idempotency does hold if the probabilistic choice is triggered *after* the non-deterministic choice is made — this is the behaviour that we can find in our framework and in pGCL,where non-deterministic choice is history-aware, but lacks prescience [9, p.187].

We can reproduce prescient non-deterministic behaviour if we run the program twice with probabilistic choice on local variables, and then merge the outputs by means of a non-deterministic choice: this is a behaviour that has nothing to do with idempotency — we keep the actions of one program separate from the other's, so we are actually dealing with two *different* program instances that share the same specification.

We are now going to treat the well-known Monty Hall game as an example, which contains all of the main constructs of pGCL and shows the interaction between demonic and probabilistic choice.

The Monty Hall Game. In the Monty Hall game a player is challenged to guess which of the three doors in front of him hides a car. After having chosen a door among the three possible options, Monty Hall will open one of the remaining two doors: Monty Hall knows where the car is, so he is going to open one of the other two; the player is given the chance to change his guess at this point.

It is known from the literature that the player will maximize the probability of finding the car if now he changes the door he has chosen (the probability will be $2/3$) — this is Bertrand's box paradox (1889).

In fact the player can lose only if his first choice was the i-th door, which is hiding the car (and this happens with probability $1/3$), so after Monty Hall has opened the k-th door, that is one of the two hiding a goat, the switching strategy leads the player's final choice to be the j-th door, which is hiding a goat.

Nevertheless this is a winning strategy with probability $2/3$, as the chances of winning equal the chances of choosing a door hiding a goat, when all doors are closed. In fact choosing the j-th door forces Monty Hall to open the k-th door, and switching makes the player choose the i-th door.

The following is a short program, which uses the program constructs defined above to implement the game — in Figure 3 we give the definition for each variable, function and instruction that we are using:

$$P \triangleq \texttt{setup;player;host;guess}$$

The variables a, b, c have values in the set $\{1, 2, 3\}$, therefore the state space is:

$$S = \left\{\sigma \mid \sigma = \underline{v} \mapsto \underline{\boldsymbol{val}}\right\}$$

where $\underline{v} = (a, b, c)$ and $\underline{\boldsymbol{val}} \in \{1, 2, 3\} \times \{1, 2, 3\} \times \{1, 2, 3\}$.

The initial distribution is a parameter of the problem: we assume its weight is 1, but make no further assumptions on the individual weight of each state.

The first instruction is made of three assignments[8], combined via non-deterministic choice:

$$\begin{aligned} a := i \quad &= \quad \delta' = \delta\{\!|{}^{i}/_{a}|\!\} \\ \texttt{setup} \quad &= \quad \exists \pi_1, \pi_2, \pi_3 \bullet \delta' = \delta\langle\pi_1\rangle\{\!|{}^{1}/_{a}|\!\} + \delta\langle\pi_2\rangle\{\!|{}^{2}/_{a}|\!\} + \delta\langle\pi_3\rangle\{\!|{}^{3}/_{a}|\!\} \\ &\quad \wedge \ \pi_3 = \iota - \pi_1 - \pi_2 \end{aligned}$$

The second instruction is also made of three assignments, but this time they are combined via a uniform probabilistic choice:

$$\begin{aligned} b := i \quad &= \quad \delta' = \delta\{\!|{}^{i}/_{b}|\!\} \\ \texttt{player} \quad &= \quad \delta' = {}^{1}/_{3} \cdot \delta\{\!|{}^{1}/_{b}|\!\} + {}^{1}/_{3} \cdot \delta\{\!|{}^{2}/_{b}|\!\} + {}^{1}/_{3} \cdot \delta\{\!|{}^{3}/_{b}|\!\} \end{aligned}$$

$$\begin{array}{lll} a \triangleq \text{the position of the car} & & \mathcal{S}(x,y) \triangleq \min\big(\{1,2,3\} \setminus \{x,y\}\big) \\ b \triangleq \text{the player's guess} & & \mathcal{H}_m(x) \triangleq \min\big(\{1,2,3\} \setminus \{x\}\big) \\ c \triangleq \text{Monty Hall's hint} & & \mathcal{H}_M(x) \triangleq \max\big(\{1,2,3\} \setminus \{x\}\big) \end{array}$$

$$\begin{aligned} \texttt{setup} &\triangleq a := 1 \sqcap (a := 2 \sqcap a := 3) && \texttt{[1]} \\ \texttt{player} &\triangleq b := 1 \ {}_{\frac{1}{3}}\!\oplus (b := 2 \ {}_{\frac{1}{2}}\!\oplus b := 3) && \texttt{[2]} \\ \texttt{host} &\triangleq c := \mathcal{S}(a,b) \lhd (a \neq b) \rhd \big(c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a)\big) && \texttt{[3]} \\ \texttt{guess} &\triangleq b := \mathcal{S}(b,c) && \texttt{[4]} \end{aligned}$$

Fig. 3. Variables, functions and instructions for the program implementing the Monty Hall game

[8] We use the notation $\{\!|{}^{e}/_{x}|\!\}$ for the assignment `x:=e`, which leaves all other variables unchanged.

We have an if-statement in the third instruction, so we have:

$$\begin{aligned}
c := \mathcal{S}(a,b) &= \delta' = \delta\{\!|{}^{\mathcal{S}(a,b)}/_{c}|\!\} \\
c := \mathcal{H}_m(a) &= \delta' = \delta\{\!|{}^{\mathcal{H}_m(a)}/_{c}|\!\} \\
c := \mathcal{H}_M(a) &= \delta' = \delta\{\!|{}^{\mathcal{H}_M(a)}/_{c}|\!\} \\
c := \mathcal{H}_m(a) \sqcap c := \mathcal{H}_M(a) &= \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle \pi_{\mathcal{H}} \rangle \{\!|{}^{\mathcal{H}_m(a)}/_{c}|\!\} + \delta\langle \bar{\pi}_{\mathcal{H}} \rangle \{\!|{}^{\mathcal{H}_M(a)}/_{c}|\!\} \\
\texttt{host} &= \exists \pi_{\mathcal{H}} \bullet \delta' = \delta\langle a \neq b \rangle \{\!|{}^{\mathcal{S}(a,b)}/_{c}|\!\} + \\
&\quad + \delta\langle a = b \rangle \langle \pi_{\mathcal{H}} \rangle \{\!|{}^{\mathcal{H}_m(a)}/_{c}|\!\} + \delta\langle a = b \rangle \langle \iota - \pi_{\mathcal{H}} \rangle \{\!|{}^{\mathcal{H}_M(a)}/_{c}|\!\}
\end{aligned}$$

Finally the fourth instruction gives

$$b := \mathcal{S}(b,c) \quad = \quad \delta' = \delta\{\!|{}^{\mathcal{S}(b,c)}/_{b}|\!\}$$

If we compose sequentially the four instructions (and jump to conclusions, full details are available in [1]), we obtain the following expression for the final probability distribution, which describes the program output:

$$\begin{aligned}
\delta' = \sum_{i \neq j} {}^1/_3 \cdot \delta\langle \pi_i \rangle \{\!|{}^{i}/_{a}|\!\} \{\!|{}^{j}/_{b}|\!\} \langle a \neq b \rangle \{\!|{}^{\mathcal{S}(a,b)}/_{c}|\!\} \{\!|{}^{\mathcal{S}(b,c)}/_{b}|\!\} \\
+ \sum {}^1/_3 \cdot \delta\langle \pi_i \rangle \{\!|{}^{i}/_{a}|\!\} \{\!|{}^{i}/_{b}|\!\} \langle a = b \rangle \langle \pi_{\texttt{host}} \rangle \{\!|{}^{\mathcal{H}(a)}/_{c}|\!\} \{\!|{}^{\mathcal{S}(b,c)}/_{b}|\!\}
\end{aligned}$$

where i, j range over $\{1,2,3\}$ and $\pi_{\texttt{host}}$ ranges over $\{\pi_{\mathcal{H}}, \bar{\pi}_{\mathcal{H}}\}$ — and $\mathcal{H}$ will be $\mathcal{H}_m$ or $\mathcal{H}_M$ depending on $\pi_{\texttt{host}}$.

To evaluate the probability of winning, which is the probability of $a = b$, we have to evaluate $\|\delta'\langle a = b \rangle\|$; if we recall that $\iota\langle a = b \rangle$ represents the expectation of the predicate $a = b$, we can see that we are computing its expected value.

In the above expression we can distinguish two kinds of terms, and if we work on each one under the winning condition we obtain:

$$\begin{aligned}
\delta\langle \pi_i \rangle \{\!|{}^{i}/_{a}|\!\} \{\!|{}^{j}/_{b}|\!\} \langle a \neq b \rangle \{\!|{}^{\mathcal{S}(a,b)}/_{c}|\!\} \{\!|{}^{\mathcal{S}(b,c)}/_{b}|\!\} \langle a = b \rangle &= \delta\langle \pi_i \rangle \{\!|{}^{i,j}/_{a,b}|\!\} \{\!|{}^{\mathcal{S}(a,b),a}/_{c,b}|\!\} \\
\delta\langle \pi_i \rangle \{\!|{}^{i}/_{a}|\!\} \{\!|{}^{i}/_{b}|\!\} \langle a = b \rangle \langle \pi_{\texttt{host}} \rangle \{\!|{}^{\mathcal{H}(a)}/_{c}|\!\} \{\!|{}^{\mathcal{S}(b,c)}/_{b}|\!\} \langle a = b \rangle &= \epsilon
\end{aligned}$$

The terms of the second kind will give no contribution to the overall weight of $\delta'\langle a = b \rangle$ (and in fact they account for the case when the player's first guess was the right one), whereas all others contribute with ${}^1/_3 \cdot \|\delta\langle \pi_i \rangle \{\!|{}^{i,j}/_{a,b}|\!\} \{\!|{}^{\mathcal{S}(a,b),a}/_{c,b}|\!\}\|$ (and of course these account for the case when the player had first chosen a door hiding a goat).

As both remapping operations use expressions defined everywhere, and thanks to the fact that in this condition the remap operators preserves the weight of a distribution, we have that:

$$\|\delta\langle \pi_i \rangle \{\!|{}^{i,j}/_{a,b}|\!\} \{\!|{}^{\mathcal{S}(a,b),a}/_{c,b}|\!\}\| = \|\delta\langle \pi_i \rangle\|$$

Therefore we have:

$$\|\delta'\langle a = b \rangle\| = \|2 \cdot ({}^1/_3 \cdot \delta\langle \pi_1 \rangle + {}^1/_3 \cdot \delta\langle \pi_2 \rangle + {}^1/_3 \cdot \delta\langle \pi_3 \rangle)\| = {}^2/_3 \cdot \|\delta\|$$

We have assumed that the weight of the initial distribution is 1, so the weight of all winning states is $2/3$ — it is now clear why we did not need to make any other assumption, as this is all that matters, as all the variables undergo at least an assignment during the run of the program. $2/3$ is also the expected value for each of the initial states, so the pre-expectation assigning this weight to every state corresponds to the post-expectation of the predicate $\iota\langle a = b\rangle$.

4.3 Generic Choice

Now that we have given an appropriate definition of non-deterministic choice, it is worth to remark in passing that we can see how all choice constructs follow a common pattern.

The reason is that all choice constructs can be seen as a specific instance of a generic choice construct:

$$\text{choice}\big(A, B, X\big) \;\triangleq\; \exists \pi, \delta_A, \delta_B \bullet \pi \in X \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \;\wedge \delta' = \delta_A + \delta_B$$

where $X \subseteq D_w$ and D_w is the set of all weighting distributions.

We can express all our choice constructs with appropriate choices of X:

- for $X = \{\iota\langle c\rangle\}$ we have conditional choice: $A \lhd c \rhd B = \text{choice}\big(A, B, \{\iota\langle c\rangle\}\big)$
- for $X = \{p \cdot \iota\}$ we have probabilistic choice: $A \;{}_p\!\oplus B = \text{choice}\big(A, B, \{p \cdot \iota\}\big)$
- for $X = D_w$ we have non-deterministic choice: $A \sqcap B = \text{choice}\big(A, B, D_w\big)$

Moreover we can see the disjunction of two programs as another kind of choice, where $X = \{\epsilon, \iota\}$: $A \vee B = \text{choice}\big(A, B, \{\epsilon, \iota\}\big)$

Our generic choice operator allows us to define a framework with only one choice construct, where all of the usual choice operators can be seen as syntactic sugar of a particular class of generic choices; moreover we can also use this generic construct to create new kinds of choices, other than the more traditional ones—the reader can refer to [1] for some examples; the potential of this generic choice operator has still to be fully explored.

4.4 The Linkage between Other Semantic Models and Ours

The relational demonic semantics for pGCL [15, p139] is given as a function from a state to a *probabilistically closed* set[9] of distributions: $S \to \mathbb{C}S$. Kleisli lifting (See Appendix A) of that model results in a function between such sets of distributions, so $p : S \to \mathbb{C}S$ is lifted to $p^* : \mathbb{C}S \to \mathbb{C}S$. From this lifted semantics, we can extract the corresponding UTP relation (R) on distributions as follows:

$$R = \{\, (\delta, \delta') \mid \delta' \in p^*\{\delta\} \,\}$$

Things are slightly more complicated if we want to relate the `wp`-semantics from [15] to our semantic model. The way to do this is to observe that an expectation

[9] Here denoted by $\mathbb{C}S$.

is a random variable (with non-negative real values), and as such it can be represented as a distribution χ in our framework. Then if χ' represents a post-expectation and A is a program, we can define the corresponding pre-expectation χ by computing the expected final weight of each state before A is run:

$$\chi(\sigma) = \min\big(\{\|\chi' \cdot \delta'\| \mid A(\eta_\sigma, \delta')\}\big)$$

Here η_σ represents a *point distribution*, which is a distribution where all states other than σ map to zero, while σ maps to 1:

$$\eta_\sigma \;\widehat{=}\; \epsilon \dagger \{\sigma \mapsto 1\}$$

So, $A(\eta_\sigma, \delta')$ is true for all δ' that can result from running A given a point distribution about σ. For each such δ' we scale with the post-expectation, and take the minimum over those. It shall be noted that this set of δ' so obtained is a singleton set for all deterministic constructs. We extract of the pointwise minimum from that set if not a singleton, as in this case we have non-determinism, and so we have to mirror the pointwise minimum used in Figure 1.

5 Conclusion and Future Work

We have provided an encoding of the semantics of pGCL in UTP, as a homogeneous relation on the alphabet $\{\delta, \delta'\}$, where the before and after variables are distributions over program states. The key is that our semantics models probabilistic programs as predicate transformers, so allowing us to claim that "probabilistic programs are predicates too". We have shown that we can deal with variables by name, despite their being entangled in the semantic domain, and that the laws of pGCL are provable from our semantics. In addition we have formulated our semantics in such a way as to be able to view all choices as instances of a generic choice construct, and even to be able to allow disjunction back in as a form of choice.

We have shown the linkage between our semantic model and the two models that feature in [10,15]: this will lead to a formalization of the healthiness conditions, which characterise the predicates in our framework, and which we expect to be substantially the same, modulo an appropriate generalization, as in pGCL.

A further step forward to be taken is to explore the role of auxiliary variables such as `ok` and `ok′` that capture a behaviour such as termination: non-termination leads to probability sub-distributions, similar to what happens in pGCL, so we could manage without, but their introduction — together with other auxiliary variables such as `wait` and `wait′` — may prove of help in moving towards the encoding of reactive systems in this framework.

This is important, as the long term focus of this work is on a probabilistic variant of *Circus*, which requires semantic models for probabilistic process algebras like pCSP [18,4] or PTSC [20]. These will then have to be integrated with our pGCL semantics in much the same way that the theory of Reactive Designs in UTP is the basis for the semantics of *Circus*-like languages.

Acknowledgements. We wish to thank (some of) the anonymous referees who have reviewed previous versions of this paper for their insightful comments and suggestions.

References

1. Bresciani, R., Butterfield, A.: Towards a UTP-style framework to deal with probabilities. Technical Report TCD-CS-2011-09, FMG, Trinity College Dublin, Ireland (August 2011)
2. Butterfield, A. (ed.): UTP 2008. LNCS, vol. 5713, pp. 22–41. Springer, Heidelberg (2010)
3. 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)
4. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4) (2008)
5. Dunne, S., Stoddart, B. (eds.): UTP 2006. LNCS, vol. 4010, pp. 236–256. Springer, Heidelberg (2006)
6. Freitas, L., Woodcock, J., Butterfield, A.: Posix and the verification grand challenge: A roadmap. In: 13th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2008, March 31-April 3, pp. 153–162 (2008)
7. Gancarski, P., Butterfield, A.: The Denotational Semantics of slotted-Circus. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 451–466. Springer, Heidelberg (2009)
8. He, J.: A probabilistic BPEL-like language. In: Qin [22], pp. 74–100
9. He, J., Sanders, J.W.: Unifying probability. In: Dunne and Stoddart [5], pp. 173–199
10. He, J., Seidel, K., McIver, A.: Probabilistic models for the guarded command language. Science of Computer Programming 28(2-3), 171–192 (1997); Formal Specifications: Foundations, Methods, Tools and Applications
11. Hoare, C.A.R.: Programs are predicates. In: Proceedings of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages, pp. 141–155. Prentice-Hall, Upper Saddle River (1985)
12. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
13. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
14. Kozen, D.: A probabilistic pdl. J. Comput. Syst. Sci. 30(2), 162–178 (1985)
15. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). Springer, Heidelberg (2004)
16. McIver, A., Morgan, C.: Abstraction and refinement in probabilistic systems. SIGMETRICS Performance Evaluation Review 32(4), 41–47 (2005)
17. Morgan, C., McIver, A.: A probabilistic temporal calculus based on expectations. Technical Report PRG-TR-13-97, Oxford University Computing Laboratory (1997)
18. Morgan, C., McIver, A., Seidel, K., Sanders, J.W.: Refinement-oriented probability for CSP. Formal Asp. Comput. 8(6), 617–647 (1996)
19. Ndukwu, U., McIver, A.: An expectation transformer approach to predicate abstraction and data independence for probabilistic programs. CoRR (2010)

20. Ndukwu, U., Sanders, J.W.: Reasoning about a distributed probabilistic system. In: Downey, R., Manyem, P. (eds.) Fifteenth Computing: The Australasian Theory Symposium (CATS 2009). CRPIT, vol. 94, pp. 35–42. ACS, Wellington (2009)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput. 21(1-2), 3–32 (2009)
22. Qin, S. (ed.): UTP 2010. LNCS, vol. 6445, pp. 188–206. Springer, Heidelberg (2010)
23. Sherif, A., Kleinberg, R.D.: Towards a Time Model for Circus. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002)

A Keisli Composition

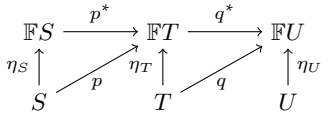
Assume a semantic model of the form $S \to \mathbb{F}S$ where $\mathbb{F}$ is a type constructor (functor). The question that naturally arises is how to compose such functions, i.e., given $p : S \to \mathbb{F}T$ and $q : T \to \mathbb{F}U$, how do we compose these to get $(p; q) : S \to \mathbb{F}U$? The standard solution for this is Kleisli lifting and composition which involves two functions with the following signatures:

$$\eta_S : S \to \mathbb{F}S \qquad \_^* : (S \to \mathbb{F}T) \to (\mathbb{F}S \to \mathbb{F}T)$$

that obey the following laws:

$$\eta_S^* = id_{\mathbb{F}S} \qquad p^* \circ \eta_S = p \qquad (q^* \circ p)^* = q^* \circ p^*$$

The intuition behind these is best understood in a diagram:



The Kleisli composition of p and q is given by $q^* \circ p$, where $\circ$ denotes regular function composition.

In this paper $\mathbb{F}S = \mathbb{P}(S \to [0, 1])$.

Behaviour-Based Cheat Detection in Multiplayer Games with Event-B

HaiYun Tian*, Phillip J. Brooke, and Anne-Gwenn Bosser

School of Computing, Teesside University, Middlesbrough, UK, TS1 3BA
{H.Tian,A.G.Bosser}@tees.ac.uk, pjb@scm.tees.ac.uk

Abstract. Cheating is a key issue in multiplayer games as it causes unfairness which reduces legitimate users' satisfaction and is thus detrimental to game revenue. Many commercial solutions prevent cheats by reacting to specific implementations of cheats. As a result, they respond more slowly to fast-changing cheat techniques. This work proposes a framework using Event-B to describe and detect cheats from server-visible game behaviours. We argue that this cheat detection is more resistant to changing cheat techniques.

Keywords: Cheat detection, multiplayer games, Event-B.

1 Introduction

Multiplayer games give players a sense of reality and engagement more so than in single-player games. They have gained considerable popularity in the entertainment world [5,19,10]. Cheating is a major concern for many game developers as it reduces the fairness of games, damages the expected game experience and thus decreases revenue [10,7,17]. "Cheating" refers to any game behaviour that players use to achieve an unfair advantage and/or a target that they are not supposed to [15]. Cheating may exceed the possible bounds of human capability, e.g., Aimbot, Spinbot, or provide "extra-sensory perception" such as seeing through opaque objects (commonly called "wallhacking") as well as learning about the hidden information (ESP) [23]. Moreover, cheating has grown to such an extent that not only are private hackers involved but also some companies commercially thrive by offering cheat techniques [11,10].

Game developers face an up-hill battle with cheat developers [17]. Many commercial solutions (e.g., DMW [13], GameGuard [14], VAC [21], etc.) act reactively by discovering and studying unknown cheat techniques then developing countermeasures, as illustrated in Figure 1. However, games can remain vulnerable to particular cheats in this defense process. This work is inspired by Laurens et al. [17], a proof-of-concept solution that calculates game behaviours for indications of cheating. This work attempts to formalise the description of game behaviours using Event-B and provides a behaviour analysis method for detecting cheating. Although we specify some behaviours within a particular game,

* Corresponding author.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 206–220, 2012.

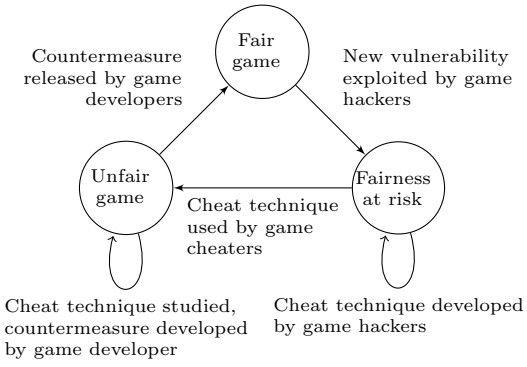


Fig. 1. Traditional cheat defence loop

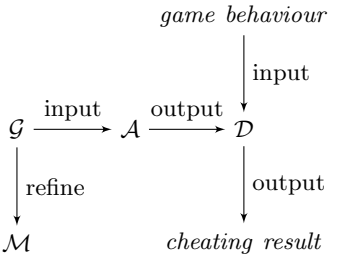


Fig. 2. $\mathcal{A}$'s environment

we are not attempting to specify a complete game. As a result, matters such as deadlock, fairness and liveness not within the scope of this work.

The rest of this paper is structured as follows. Section 2 presents our approach to modelling cheating behaviour in Event-B; section 3 introduces the actual production of cheat detectors. Section 4 validates the framework using an example game. Section 5 discusses related work before we conclude in section 6.

2 Behaviour-Based Cheat Detection with Formal Methods

The framework presented in this paper contains both a method for modelling game behaviour and a procedure for producing behaviour-based cheat detectors. This framework, illustrated in Figure 2, can be divided into two sub-systems, S_1 and S_2:

$$S_1 = (\mathcal{M}, \mathcal{G}, \mathcal{A}, \mathcal{D}) \qquad S_2 = (\mathit{game\ behaviour}, \mathcal{D}, \mathit{cheating\ results})$$

S_1 produces cheat detectors: algorithm $\mathcal{A}$ uses $\mathcal{G}$, a derivation of a base model $\mathcal{M}$, to produce a cheat detector $\mathcal{D}$. S_2 uses a detector produced by S_1 to measure the cheating behaviours of players. We next consider how we might model cheating behaviours before introducing Event-B in section 2.2.

2.1 Cheating Behaviour Modelling

For now, we consider only games based on a client/server architecture. That is, we have multiple players who send commands to and receive feedback and updates from the server.

A player's behaviour is usually a thoughtful response to the current game state. This game state is distributed by the game server to all involved players, and their responses result in the subsequent game state. We write a game state as

$$game\_state = \left\{ \begin{array}{c} player_1.local\_state, \\ \vdots, \\ player_n.local\_state \end{array} \right\}$$

i.e., a *game_state* is a collection of client states reported by all involved clients $player_{1..n}$ to their game server at an instant. Thus, a particular game behaviour can be described as

$$game\_behaviour = (game\_state, game\_state')$$

This means that a game behaviour is an ordered pair of antecedent and consequent game states. A *player*'s behaviour is also an ordered pair

$$player\_behaviour = (game\_state, game\_state'(player)).$$

player represents a player and $game\_state'(player)$ stands for the *player*'s response to the antecedent *game_state*.

We use predicates to define these game behaviours. Let *STATE* be a set containing all possible client states and *PLAYER* a set of all involved players. We give a predicate, $P1$, for describing general player behaviour below.

$$P1((game\_state, game\_state'(player))) = \\ game\_state, game\_state' \in PLAYER \rightarrow STATE \ \wedge \ player \in PLAYER$$

P1 is very abstract. Suppose the game involves three possible player responses, moving, aiming and firing. We can then refine *STATE* to $\{position, aim, fire\}$. At this level of abstraction, we can also specify a particular cheat, a "triggerbot" which automatically fires whenever an opponent is located such that the opponent is likely to be hit. This gives its users an advantage over legitimate opponents of being the first to fire. We can describe triggerbotting behaviour in the predicate $P2$ as defined below.

$$\begin{aligned} P2((game\_state, game\_state'(player))) = P1 \ \wedge \\ (\exists \ peer \in PLAYER \ \wedge \ peer \neq player \\ \wedge \ game\_state'(player).aim = game\_state(peer).location) \\ \Rightarrow game\_state'(player).fire = true \end{aligned}$$

The predicate $P2$ includes $P1$ in its conjunction: this means that all behaviour described by $P2$ are permitted by $P1$. In addition, $P2$ describes that, whenever an opponent is at the point of aim, the player immediately fires. It is very likely that fair (i.e., non-cheating) players can achieve immediate fire responses too, but we suggest that they do it less frequently than a cheater since a triggerbot allows its users to exceed the typical bounds of human capability. Most widespread game cheats do not employ an 'alien' behaviour, a behaviour out of the bounds of $\{P1\}$. Otherwise, cheat detection would not be difficult since the message from the client would be obviously unacceptable. As a result, many cheats cause more effective play than a fair player during a game period. However, a cheat that exhibits behaviours identical (in probabilistic terms) to a fair player cannot be detected by our system.

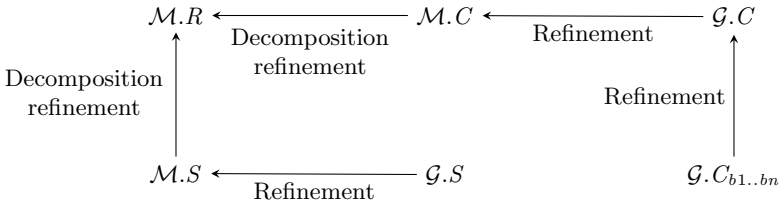


Fig. 3. Derivation from base model

Using formal languages to describe a game's behaviour can ensure the consistency in different developers' individual understanding of cheating behaviour at different abstraction levels. For example, the members of a development team (e.g., game designers, game developers, security specialists) can use a formal description to ensure a consistent description of particular cheats, as well as the overall design (in common with existing formal methods approaches).

2.2 Cheating Behaviour in Event-B

Event-B [3] is chosen in this particular work for two reasons. Firstly, many commercial games use a vast amount of data associated with world simulation, and thus modelling even an abstraction of these simulations would encounter complicated data structures. Event-B is a state-rich formalism which suits this requirement. Other options included Z [22], B [1] and Circus [9]. Secondly, Event-B has a toolkit Rodin [4,2]. Rodin contains a model editor, an obligation generator, an obligation viewer, an obligation automatic proof, comprehensive model analysis, etc., together with a friendly user interface.

Event-B models can be constructed in an arbitrary way. As our contribution involves $\mathcal{A}$, a mechanical procedure for producing cheat detectors, we need to constrain these models so that $\mathcal{A}$ is able to work. This work uses constraints on both refinement architecture and machine refinement to ensure $\mathcal{G}$'s compatibility to $\mathcal{A}$ via an abstraction $\mathcal{M}$, the base model.

We refine the machines as shown in Figure 3. The base model $\mathcal{M}$ acts as the top abstraction of client/server game systems, providing the game model with a structure suitable for subsequent processing by $\mathcal{A}$. Decomposition refinement is a technique of describing parallelism in Event-B due to Butler [8]: $\mathcal{M}$ uses it to describe synchronisation between game servers and game clients. Details of a specific game and its cheats can be added by refinements from $\mathcal{M}$ to $\mathcal{G}$ via refinement. The machine $\mathcal{G}.S$, the model of game servers, is refined from $\mathcal{M}.S$. The machine $\mathcal{G}.C$ refines $\mathcal{M}.C$, modelling the game clients of a specific game. The machines $\mathcal{G}.C_{b1..bn}$ refine $\mathcal{G}.C$, modelling different types of cheating game clients.

During this refinement, we instantiate some variables listed in Table 1. For conciseness, this report omits some details; they can be found in [20] (along with full versions of the machines presented shortly). Machine $\mathcal{G}.C$ can be defined as follows:

Table 1. Machine component variables

Var	Machine component description
$\mathcal{S}_f$	A list of machine state variables.
$\mathcal{I}_f(V)$	A list of invariants on V, implicitly conjoined.
$\mathcal{E}_f$	A label for a machine event.
$\mathcal{X}_f$	A list of event parameters.
$\mathcal{P}_f(V)$	A list of guards on V, implicitly conjoined.
$\mathcal{Q}_f(V)$	A list of actions on V.

Table 2. Example of component variable assignments (client)

Variable	Machine component
$\mathcal{E}_f$ (event name)	*play*
$\mathcal{P}_f$ (event para)	*pos*
$\mathcal{P}_f$ (event guard)	$position(cstate) = pos$

Machine $\mathcal{G}.C$
Refines $\mathcal{M}.C$
Sees $\mathcal{G}.Cxt$
Variables $buffer, clients, game\_situation, local\_state, own\_ID, \mathcal{S}_f$
Invariants $\ldots, \mathcal{I}_f(\mathcal{S}_f, buffer, clients, game\_situation, local\_state, own\_ID)$

Events
$initialisation \mathrel{\widehat{=}} \mathcal{Q}_f(\mathcal{S}_f) \ldots$ **end**
$\ldots$
$\mathcal{E}_f \mathrel{\widehat{=}}$
refine $client\_act$
any $cstate, \mathcal{X}_f$ **where**
$clientID(cstate) = own\_ID$
$cstate \in CLIENT\_STATE$
$\mathcal{P}_f(\mathcal{S}_f, \mathcal{X}_f, buffer, clients, game\_situation, local\_state, own\_ID)$
then
$local\_state := cstate$
$\mathcal{Q}_f(\mathcal{S}_f, \mathcal{X}_f, buffer, clients, game\_situation, own\_ID)$
End
End

The full text of this machine (and others given in this work) has been checked in Rodin; the version presented here is a shortened version from Rodin's LaTeX plugin.

Suppose we have a game where avatars can change their positions. Assume the model context $\mathcal{G}.Cxt$ has a fresh carrier set MAP containing all possible positions that avatars can move to in the game; a set $CLIENT\_STATE$ contains all possible client state; and one axiom $position \in CLIENT\_STATE \rightarrow MAP$.

Table 3. Component variable assignments for wallhacking example

Variable	Machine component
$\mathcal{E}_f$ (event name)	*wallhack*
$\mathcal{P}_f$ (event guard)	$\exists\, opponent\_state \in game\_situation$ $\wedge\ clientID(opponent\_state) \neq own\_ID$ $\wedge\ position(opponent\_state) \notin VISIBLE(local\_state)$ $\wedge\ (DISTANCE(pos, position(opponent\_state))$ $\quad < DISTANCE(position(local\_state), position(opponent\_state)))$

An instance of $\mathcal{E}_f$ can be made using Table 2 so that

play $\widehat{=}$
refine *client_act*
any *cstate*, *pos* **where**
$clientID(cstate) = own\_ID$
$cstate \in CLIENT\_STATE$
$position(cstate) = pos$
then
$local\_state := cstate$
End

Now consider the cheat "wallhacking". This cheat allows players to see through opaque objects. A behavioural characteristic of this cheat can be described that

Antecedent state: An opponent is not visible to a wallhacker.
Player's response: The wallhacker approach that opponent.

This characteristic can be described in Table 3 by adding a fresh guard to *play* in a refinement. This guard involves some fresh constants and variables. Briefly, *game_situation* is defined in the base model $\mathcal{M}$; it is a set containing the up-to-date game state. These constants are defined in the Event-B context. For example, the constant *VISIBLE* represents the game function that calculates the up-to-date visible zone for a game client, and *DISTANCE* calculates the distance between two map locations.

The event *wallhack* refines *play* by adding a fresh guard, which describes that a player can approach to a legally invisible opponent. A wallhacking client machine can then be composed as the following.

Machine $\mathcal{G}_{TankWar}.C_{Wallhack}$
Refines $\mathcal{G}_{TankWar}.C$
Sees $\mathcal{G}_{TankWar}.Cxt$
Variables ...
Invariant ...
Event ...
wallhack $\widehat{=}$ ... **End**
End

As more behavioural characteristics are identified, more events like *wallhack* can be added, and better cheat detection would is achieved using the algorithm $\mathcal{A}$ (described in the next section).

3 Production of Cheat Detector

This work aims at mechanically producing behaviour-based cheat detectors. We now describe our approach to detection, given the Event-B models outlined above.

Suppose we embed a specific model of cheating behaviour into a robot player, *robot* (e.g., the wallhacker behaviour or the triggerbot behaviour). Assume we now ask the detector to examine a game player, *player*. The detector first records a sequence of *player*'s behaviours, which are not necessarily adjacent in time or captured at the same interval, and then runs *robot* using the antecedent game state of each behaviour and collects *robot*'s response to the same sequence. Recall that a player's game behaviour is an ordered pair of the antecedent game state and the player's response (consequent client state). It is very likely that *robot* has a nondeterministic choice in its response. Thus, the detector collects from the *robot* a set that contains all its possible responses in that context. This is repeated until all collected *player* behaviours are used. In the end, the detector has a sequence of responses from *player* and a corresponding sequence of response sets from *robot*. The proportion of *player*'s response contained in *robot*'s corresponding response set is the rate that *player* behaves like this cheat.

To calculate whether or not *player* can make the same response as *robot*, we introduce the function *Exam* below.

$$Exam((game\_state, game\_state'), player, robot) = \begin{cases} 1 & \text{if } game\_state'(player) \in \text{robot's response set to } game\_state \\ 0 & \text{otherwise.} \end{cases}$$

The detector $\mathcal{D}$ uses *Exam* to assess monitored game behaviour. Recall that a game behaviour is a game state transition and described as (*game_state*, *game_state*′). Let *tr* be a sequence of game behaviour and k be the length of *tr*:

$$tr = \langle (tr_1.game\_state, tr_1.game\_state'), \ldots, (tr_k.game\_state, tr_k.game\_state') \rangle$$

That is, *tr* is a sequence of observations of game behaviours, or a sequence of pairs of states. The expressions $tr_i.game\_state$ and $tr_i.game\_state'$ represent the first element (antecedent game state) and the second element (consequent game state) of a particular transition tr_i.

Rather than calculating a single numeric matching rate between *player* and *robot*, our detector calculates how much *player* acts as *robot* for each leading subsequence of *tr* and returns a sequence of matching rates. This allows consideration of trends through a particular game, as it carries more information than a single final rate. For example, when a player finishes at 15%, but stays above 60% more than half time of the game, this player might be suspicious.

Let $rate_i$ be the matching rate for the subsequence $\langle tr_1, \ldots, t_k \rangle$. We can calculate $rate_i = \dfrac{\sum_{m=1}^{i} Exam(tr_m, player, robot)}{i}$ $[i \in 1..k]$. This describes that $rate_i$ is the proportion of the transitions in which *player* matches *robot* to the

total transitions that are collected until t_i. We can use $rate_i$ to define a function *rateDst* to calculate a rate sequence

$$rateDst(\langle tr_1, \ldots, tr_k \rangle, player, robot) \mathrel{\widehat{=}} rate_1, \ldots, rate_k$$

Thus,

$$rateDst(\langle tr_1, \ldots, tr_k \rangle, player, robot) = \langle \frac{\sum_{m=1}^{1} Exam(tr_1, player, robot)}{1}, \ldots, \frac{\sum_{m=1}^{k} Exam(tr_m, player, robot)}{k} \rangle$$

For example, a sequence of game behaviour (game state transition) is captured as $tr' = \langle tr_1, tr_2, tr_3, tr_4 \rangle$. Given the following

$$Exam(tr_1, player, robot) = 1,\ Exam(tr_3, player, robot) = 0,$$
$$Exam(tr_2, player, robot) = 0,\ Exam(tr_4, player, robot) = 1$$

then $rateDst(\langle tr_1, tr_2, tr_3, tr_4 \rangle, player, robot) = \langle 100\%, 50\%, 33\%, 50\% \rangle$.

So the function *rateDst* takes the input of game behaviour sequence and produces matching rates in the same sequence. The algorithm $\mathcal{A}$ uses *rateDst* to construct the detector $\mathcal{D}$, and is defined below.

Algorithm: $\mathcal{A}$
input : $\mathcal{G}$, a game model for *game*.
output: $\mathcal{D}$, a cheat detector for *game*.
begin
  $\mathcal{D} \mathrel{\widehat{=}}$ **begin**
    input : *player*, a game client.
    input : *tr*, a sequence of server-side game state transitions.
    output: $ratedsr_{c1..cn}$, matching distributions for $\mathcal{G}.C_{b1}..\mathcal{G}.C_{bn}$.
    $ratedsr_{c1} = rateDst(tr, player, \mathcal{G}.C_{b1})$
    ⋮ ⋮
    $ratedsr_{cn} = rateDst(tr, player, \mathcal{G}.C_{bn})$
  end
end

Using the example above, $\mathcal{A}$ replaces *robot* with the machine $\mathcal{G}.C_{b1..bn}$, which describes the behaviours of different cheats. Thus, a produced detector $\mathcal{D}$ is able to detect these cheats by comparing the possible behaviours specified in the formal model. As presented in Figure 2, the algorithm $\mathcal{A}$ takes a behaviour model $\mathcal{G}$ and returns a detector $\mathcal{D}$. The resulting detector can monitor players for the cheats that $\mathcal{G}$ specifies.

Importantly, it is necessary to discuss the criteria of judging a player using the output of our cheat detectors. Judging a player does not rely on the player's

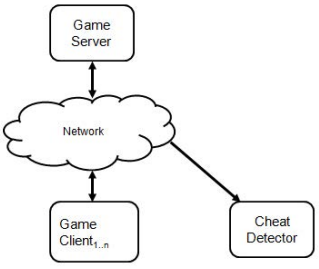


Fig. 4. Deployment of detector (1)

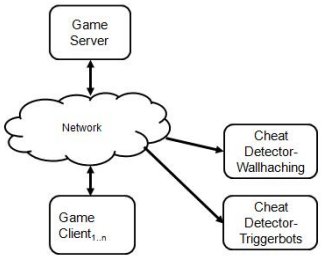


Fig. 5. Deployment of detector (2)

matching rates alone. We must consider the individual rates in the context of all the players' rates.

For example, when most players have rates between 5% and 15%, some players may always stay at about 30%: these latter players are suspects. We must also consider the cases that our detectors are wrongly specified, i.e., the predicates in the Event-B model are ineffective. This could result in no cheats being detected (false negatives) or too many fair players being identified as cheats (false positives). Another interesting case concerns most players engaging in cheating: this results in many high rates and it becomes difficult for an automated process to suitably identify them. The detection performance is determined by the quality of cheating behaviour knowledge that the framework users accumulate before embedding them in the framework as predicates.

3.1 Merits of Implementation

Our cheat detection can handle increasing amounts of work in a scalable manner. $\mathcal{A}$'s detectors only need the data packets that the game server distributes to game clients for maintaining game consistency, and requests neither extra information nor any computing service from the game server and clients. The number of detectors working for a game simultaneously has no impact on the performance of games, provided that the server broadcasts the relevant packets onto their network. When dealing with a number of cheats, rather than using a single 'giant' detector for all them as Figure 4 shows, we could produce one detector per cheat. Assume there are two cheats, wallhacking and triggerbots: a possible deployment can be as shown in Figure 5 with, one node for one cheat. When a new cheat detector is produced, it can be plugged into the network as a new independent node.

4 Validation of the Framework

This framework is validated by an experiment. An example game, *TankWar* (more fully described in [20]), was used to test the resulting detector with human volunteers. It is simple by contrast with many commercial games. But it contains

Fig. 6. Visibility zones

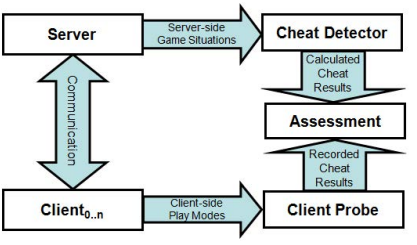


Fig. 7. Validation design

most important elements of first-person shooter games, e.g., surviving, attacking, limited vision, etc. More importantly, $\mathcal{M}$ is a generic client/server architecture game model, and it is designed for deriving behaviour model $\mathcal{G}$ for the games of this tier, even though *TankWar* is relatively small. Therefore, we suggest that it is a suitable proof-of-concept test of its applicability for larger games.

TankWar, is a real-time strategy/FPS multiplayer game, and is played by moving and shooting. Players have restricted vision as shown in Figure 6. A player's visible zone is the area to their front and is stopped by solid objects. Players can only see opponents who are in their visible zones.

Using our framework, we produce the model $\mathcal{G}_{TankWar}$, which describes wall-hacking and triggerbotting behaviour. Running $\mathcal{A}$, we obtain detector $\mathcal{D}_{TankWar}$, which is intended to detect the two cheats, as shown below.

Procedure. $D_{TankWar}$

```
1 begin
    input  : player, a game client.
    input  : tr, a sequence of server-side game state transitions.
    output: ratedsr_wallhack, ratedsr_triggerbot
2   ratedsr_wallhack = rateDst(tr, player, G.C_wallhack)
3   ratedsr_triggerbot = rateDst(tr, player, G.C_triggerbot)
4 end
```

The detector $D_{TankWar}$ is equipped with client machines $\mathcal{G}_{TankWar}.C_{wallhack}$ and $\mathcal{G}_{TankWar}.C_{triggerbot}$. To examine $\mathcal{D}_{TankWar}$, a strategy is designed as shown in Figure 7. Besides the game server and game clients, there are three more components: a cheat detector, a client monitor and an assessment module. These three components never return data to both server and clients and thus have no influence on the game experience. The client probe is a component independent from the detector. It is made only for validating the performance of the detector by recording each client's behaviour since we have full control of the clients for the experiment.

The top-right diagram of Figure 8 is an example of the detector's output, presenting the matching rates in a game for a particular player. The x-axis is a timeline and the y-axis shows matching rates. The example detector report

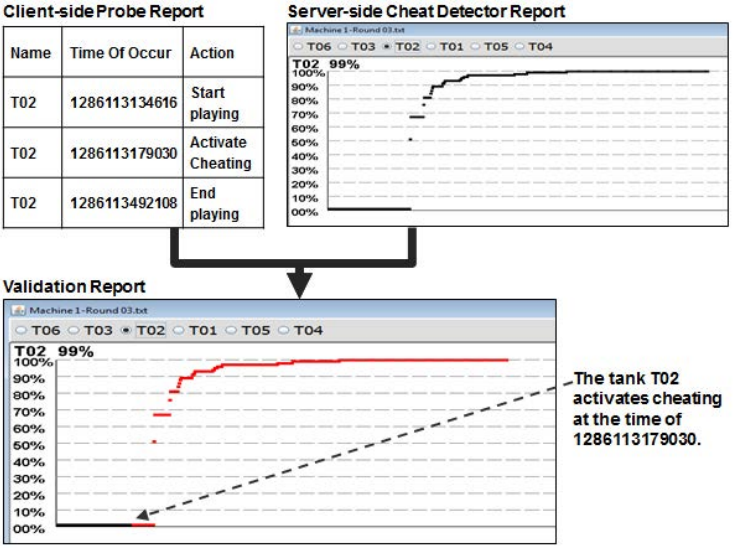


Fig. 8. Example of validation output

describes that the client T02 remained at a low rate at the beginning and increased abruptly to about 50% then gradually rose to about 90% in the end. The top-left of Figure 8 is from the client probe, consisting of three columns: user name, time and action. It reveals when and which clients activated a cheating play mode. The probe report presents that player T02 joined the game at 0s and played in a fair play mode; T02 activated cheating play mode at 45s and finished at 358s. Assessment of the detector's performance is by comparison of the reports from the client probes and the detector.

This validation involves a number of game trials: 32 games were monitored, with 16 for wallhacking and 16 for triggerbotting with five players from a pool of seven.

In most trials, a cheat finished at a rate twice or more as high as a legitimate player. There were only four exceptions, and three of them were accounted for by limitation of data (the players lost the game too quickly) and only one is a true detection failure. The success detection rate is 28 out of 32. We note that the change in detection rate over time was sometimes interesting. For example, some wallhacking detection reports show a rapid rate increase shortly after the beginning. It can be envisaged that cheat detection would perform well when game designers and developers use their knowledge about cheating to describe problems in commercial games.

5 Related Work

Recent research has proposed some novel techniques of cheat detection that do not rely on knowledge about specific game vulnerabilities as many commercial

solutions do. They consider cheats by looking at particular measurements. A work based on probability theory was proposed by Chapel et al. [10]. It proposes two complementary frameworks based on the use of the law of large numbers and the Bradley-Terry model [6], which calculate statistical indexes that suggest a player is cheating. The two frameworks are both based on the assumption that each player can be assigned a rank which determines the probability of the outcomes of their games, and determines cheating by observing the difference from resulting ranks and expected ranks. Another novel cheat detection design was proposed by Laurens et al. [17], which we have discussed in previous sections. The design statistically analyses server-side observable behaviour for indications of cheating. For example, to prevent wallhacking, the system collects data (e.g., player view and game world geometry) and then transforms the data to a measure of cheating based on, e.g., frequencies of behind-wall sight vector, distance between players and walls). Subsequently, the measurements are used by statistical algorithms to determine the probability of wallhacking.

Our work is inspired by them and has its own features. It does not rely on any particular measurement (e.g., rank, sight vector, distance). It uses behaviour models to investigate behavioural characteristics and uses the comparison of possible vs. observed behaviours to detect cheats. In common with the methods described above, its performance is resistant to fast-changing cheat implementations due to being a server-side detector.

There are other approaches. For example, it is proposed in [12] that high-level game rules can be described in temporal logic and used to verify the properties of game players at the run-time. The main difference is that our cheat detection does not reply on either rule-enforcement or rule-violation. But, it is done by simulating cheating at instant observation and matching players' behaviours with the simulation result at each instant and calculating to which extent the players are cheating during a reasonable period. In [18], a client patching mechanism is introduced that increases the difficulty of being identified and broken by hackers. The main difference is that our work uses formally-specified simulations of cheaters to detect cheating and does not need patches on game clients.

Moreover, our work has some similarity to *intrusion detection systems* (IDSes). These are primarily focused on identifying possible incidents, which are violations or imminent threats of violation of system security policies or acceptable use policies [16]. A typical IDS records observed events, and (1) matches them with some event patterns (*signatures*) corresponding to known security threats, (2) examines whether or not there are anomalous events using definitions of normal events (*profiles*) or (3) identifies unexpected sequences of events by comparing predetermined profiles of generally accepted definitions of benign protocol activity for each protocol state against the observed events.

The common feature of IDSes and this work is that they all use clients' footprints (or "server-side observable behaviour") on a server or the network to match certain 'characteristics' and calculate the indication of a threat. Typical IDSes present the characteristics (e.g., *threat signature*, *anomalous profile*, definitions of benign protocol activity) in some form of pattern or description

language and match observed behaviours against those patterns. This framework also uses a language —in this case, Event-B— to describe particular cheating behaviours. Our base model $\mathcal{M}$ does not contain any fixed cheating behaviour patterns; it has no direct concern with cheating behaviours and acts as the top abstraction. The validity of $\mathcal{G}$ and the efficiency of the resulting cheat detectors mechanically produced by $\mathcal{A}$ using $\mathcal{G}$ is determined by the quality of the cheating descriptions that are incorporated in $\mathcal{G}$. Similarly, IDSes are only as good as the signatures, profiles or definitions of benign activity.

6 Conclusions

We have demonstrated that we can use refinement in Event-B to describe some cheating behaviours in games. The resulting machines can be used to produce cheat detectors that we argue are more resistant to changing implementations of cheats. We have demonstrated the credibility of describing cheats via formal specification via experiment. This experiment tests both that we can describe the cheats in this fashion and that the resulting detector is accurate.

One might notice that this work shows a different concern from typical work in formal methods. One aim is to bring the precision of formal methods to the description of cheating behaviours. However we have not concerned ourselves with the development of games; we are not, for example, attempting to formally derive or prove the implementation of a game. As a result, it is out of scope for us to address classical issues such as deadlock, fairness, liveness and inconsistency of the games. More detailed designs may be amenable to such analysis depending on the tool support in each instance. Moreover, we are not aware of any significant game designed using such formal specification or analysis.

The major advantage of this work is that it allows game developers to proactively protect their games instead of defending passively against cheat techniques. There is no necessity of capturing behaviour in a temporally adjacent manner and/or at the same interval: for a long-running game, our detectors can randomly sample behaviour several times during a period (e.g., one or two hours), and gradually generate cheating references.

However, we must be able to describe the cheating behaviour in question in Event-B. Essentially, we are trying to describe cheats at a very abstract level such that the implementation of the cheat is inconsequential. Discovery of cheating behaviour is itself an interesting question which we have not attempted to answer here. Automatic construction of these descriptions would require some description of the rules of a game itself; thus these rules would require formal specification.

One limitation is that this work cannot efficiently detect cheats that lead only to trivial behavioural difference between fair players and cheaters. For example, some games allow virtual gifts (e.g., weapons) exchanged between players. When people abuse this feature and illegally trade, the buyers would obtain an unbalanced (unfair) power against the time they spent. Our detector might not work for this cheat since there might not be a behavioural difference between the buyers and fair players when they exchange their avatars.

Another limitation is on game architecture. The base model $\mathcal{M}$ is an abstraction of client/server architecture games, and the behaviour model $\mathcal{G}$ must be derived from it. However, this does not suit all game architectures, some of which may have strong peer-to-peer components.

This work leaves open many possible future improvements.

- Introduce a set of base models, $\mathcal{M}_{1..n}$ to allow for a broader range of game types.
- Developing easier ways to produce the predicates that describe cheating behaviour. A further development would be to automatically identify possible predicates describing cheating; but this itself would require some description of the intent and rules of any particular game. We speculate that particular patterns may arise often enough that they could be described generically.
- Extend our framework to be able to identify game players from their behaviour. This would work even if they use different user names. This can prevent people from cultivating avatars or farming games for buyers, or from changing IP address to avoid countermeasures such as IP blacklisting.

Although we have used this framework only for the classification of game users into fair-player and cheating-player groups in this paper, its application can feasibly be extended to the classification of users in other multiuser systems, such as a social-networking system. $\mathcal{A}$ could be adapted from being a creator of cheat detectors to a creator of user classifiers. A user classifier could put social network users into a variety of categories (e.g., a movie fan, a sport fan, etc.) and even sub-categories (e.g., an action movie fan, a bicycle sport fan, etc.) by calculating server visible behaviours. Thus, it might facilitate solutions for delivering relevant content to the users who are most likely to be interested.

Acknowledgements. This work was undertaken during the first author's PhD studies at Teesside University, and was funded by a research doctoral scholarship from that university. We thank the anonymous reviewers for their detailed and constructive remarks.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.-R.: A System Development Process with Event-B and the Rodin Platform (2007)
3. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press (2010)
4. Abrial, J.-R., Voisin, L.: Rodin deliverable 3.2 Event-B language (2005)
5. Bell, M.: Toward a definition of "virtual worlds". Journal of Virtual World Research 1(1) (2008)
6. Bradley, R.A., Terry, M.E.: Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. Biometrika, vol. 39 (1952)

7. Brooke, P.J., Paige, R.F., Clark, J.A., Stepney, S.: Playing the game: cheating, loopholes, and virtual identity. SIGCAS Comput. Soc. 34(2) (September 2004)
8. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
9. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
10. Chapel, L., Botvich, D., Malone, D.: Probabilistic approaches to cheating detection in online games. In: IEEE Symposium on Computational Intelligence and Games (2010)
11. de Paoli, S., Kerr, A.: We will always be one step ahead of them: A case study on the economy of cheating in MMORPGs. Journal of Virtual Worlds Research 2(4) (2010)
12. DeLap, M., Knutsson, B., Lu, H., Sokolsky, O., Sammapun, U., Lee, I., Tsarouchis, C.: Is runtime verification applicable to cheat detection? In: Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games, NetGames 2004, pp. 134–138. ACM, New York (2004)
13. DMW (2002), http://www.dmwworld.com/
14. INCA Internet (2000), http://www.inca.co.kr/
15. Choi, H.-J., Xin, J., Yan, J.: Security issues in online games. The Electronic Library 20 (2002)
16. Karen, S., Peter, M.: Guide to Intrusion Detection and Prevention Systems (IDPS). Computer Security Resource Center (2009)
17. Laurens, P., Paige, R.F., Brooke, P.J., Chivers, H.: A novel approach to the detection of cheating in multiplayer online games. In: ICECCS 2007: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 97–106. IEEE Computer Society, Washington, DC (2007)
18. Mönch, C., Grimen, G., Midtstraum, R.: Protecting online games against cheating. In: Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames 2006. ACM, New York (2006)
19. Schroeder, R.: Defining virtual worlds and virtual environments. Journal of Virtual Worlds Research 1(1), 1–3 (2008)
20. Tian, H.Y.: Formal Derivation of Behaviour-based Cheat Detectors for Multiplayer Games, PhD thesis. School of Computing, Teesside University (2012)
21. Valve Corporation (2002), http://www.valvesoftware.com/
22. Woodcock, J., Davies, J.: Using Z - Specification, Refinement, and Proof. Prentice Hall (1996)
23. Yan, J., Randell, B.: A systematic classification of cheating in online games. In: NetGames 2005: Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games, pp. 1–9. ACM, New York (2005)

Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems

Denisa Diaconescu[2], Ioana Leustean[2], Luigia Petre[1], Kaisa Sere[1], and Gheorghe Stefanescu[2]

[1] Åbo Akademi University, Finland
[2] University of Bucharest, Romania

Abstract. The state-based formal method Event-B relies on the concept of correct stepwise development, ensured by discharging corresponding proof obligations. The register-voice interactive systems (rv-IS) formalism is a recent approach for developing software systems using both structural state-based as well as interaction-based composition operators. One of the most interesting feature of the rv-IS formalism is the structuring of the components interactions. In order to study whether a more structured (rv-IS inspired) interaction approach can significantly ease the proof obligation effort needed for correct development in Event-B, we need to devise a way of integrating these formalisms. In this paper we propose a refinement-based translation from Event-B to rv-IS, exemplified with a file transfer protocol modelled in both formalisms.

1 Introduction

Event-B [2,9,14,15,16,17,18] is a state-based formalism dedicated to the refinement-based development of parallel and distributed systems. This amounts to developing an abstract model into more concrete ones, so that we are sure that a more concrete model correctly develops a more abstract one. A central advantage of Event-B is the associated Rodin tool platform [27,3] employed in discharging the proof obligations that ensure this correct development. In addition to providing a user interface for editing Event-B models, the proving process is closely integrated with the modelling process, encouraging proof-based model improvement. Event-B is currently successfully integrated in several industrial developments, for instance at Space Systems Finland [13] and at SAP [8].

The register-voice interactive systems [24,25,21,11,12,23] (rv-IS) formalism is a recent approach for developing software systems using both structural state-based as well as interaction-based composition operators. Interactive computation [29] is an important computer science topic, often related to human-computer interaction, the particular case when one of the interacting entities is human. While able to deal with such cases as well, the rv-IS formalism is more oriented to the process-to-process interaction. There are already many successful formalisms for this, including Petri nets [26], process algebras [5], π-calculus [20,19], dataflow networks [6,7], etc. The approach used in this paper integrates a dataflow-like interaction model with a classical state-based computation model.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 221–236, 2012.

One of the most interesting feature of the rv-IS formalism is the structuring of the component interactions.

Our aim is to study whether a more (rv-IS inspired) structured approach of an interactive, modular system has any effect on the correct development as designed in Event-B. More precisely, we are interested in uncovering whether the proof obligations are significantly eased when a certain structure is assumed in the model. For this, we need to devise an integration of Event-B and rv-IS, up to a level where the key features of each formalism can be easily translated into the other. We have set up the following working plan for integrating the Event-B and the rv-IS formalisms:

1. Define a notion of refinement in rv-IS models based on a combination of the refinement of state-based systems and of Broy-style refinement of dataflow-based interactive systems.
2. Define a translation EB2IS from Event-B models to structured rv-IS models.
3. Prove the translation EB2IS preserves refinement.
4. Use one of the known translations to pass from structured rv-IS models to unstructured rv-IS models, e.g, the translation in [12].
5. Define a refinement preserving translation IS2EB from unstructured rv-IS models to Event-B models.
6. Use these translations EB2IS and IS2EB to: (1) improve the discharging of proof obligations in Event-B based on rv-IS structural operators and associated decomposition techniques; (2) get tool support to develop and analyze rv-IS models.

In this paper we present a double-folded contribution. First, we introduce a refinement-preserving translation EB2IS from Event-B models to structured rv-IS models. Second, we argue our translation by analyzing an example: we present three refinement steps for modeling a simple file transfer protocol in Event-B and show the associated refined structured rv-IS models. This means we are addressing items 2. and 3. in our working plan above. We have already addressed item 1. in [10] and item 4. in [12].

We proceed as follows. In Section 2 we outline Event-B and rv-IS. In Section 3 we introduce a general translation from Event-B models to rv-IS models and briefly put forward the concept of rv-IS refinement. In Section 4 we present an example of a file transfer protocol and in Section 5 we conclude the paper.

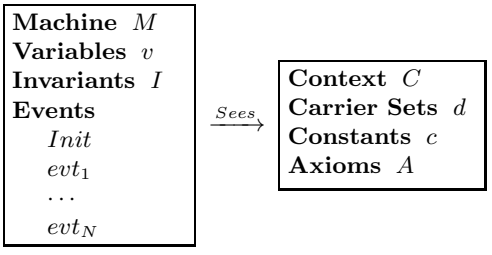
2 Preliminaries

In this section we overview the formalisms to integrate to the extent needed in this paper.

2.1 Event-B

Event-B [2] is a state-based formal method focused on the stepwise development of correct systems. This formalism is based on Action Systems [4,28] and the B-Method [1]. In Event-B, the development of a model is carried out step by step from an abstract specification to more concrete specifications.

The general form of an Event-B model is illustrated in the side figure. Models in Event-B consist of *contexts* and *machines*. A context describes the static part of a model, containing sets and constants, together with axioms about these. A machine describes the dynamic part of a model, containing variables, invariants (boolean predicates on the variables), and events, that evaluate (via event *guards*) and modify (via event *actions*) the variables. The guard of an event is an associated boolean predicate on the variables, that determines if the event can execute or not. If the event can execute, then we say it is *enabled*. The action of an event is a parallel composition of either deterministic or non-deterministic assignments. Upon executing the initializing event $Init$, computation proceeds by a repeated, non-deterministic choice and execution of an enabled event. If none of the events is enabled then the system deadlocks. The relationship *Sees* between a machine and its accompanying context denotes a structuring technique that allows the machine access to the contents of the context.



A machine M and a context C in Event-B

The semantics of events is defined using *before-after (BA) predicates* [2]. A before-after predicate describes a relationship between the system states before and after the execution of an event. The semantics of a whole Event-B model is formulated as a number of *proof obligations*, expressed in the form of logical sequents. The full list of proof obligations can be found in [2]. Every Event-B model should satisfy the event feasibility and invariant preservation properties. The feasibility of an event means that, whenever the event is enabled, its BA predicate is well-defined, i.e., there is some reachable after-state. Each event should also preserve the given model invariant. The formal semantics provides us with a foundation for establishing correctness of Event-B specifications.

System Development. Event-B employs a top-down refinement-based approach to formal system development. Development starts from an abstract system specification that models some essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into an abstract specification. These new events correspond to stuttering steps that are not visible in the abstract specification. We call such model refinement as *superposition refinement*. Moreover, Event-B formal development supports *data refinement*, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants are called *gluing invariants*.

In order to prove the correctness of each step of the development, a set of proof obligations needs to be discharged. Thus, in each development step we have mathematical proof that our model is correct. The model verification effort

and, in particular, the automatic generation and proving of the required proof obligations, are significantly facilitated by the provided tool support – the Rodin platform [27,3].

2.2 Register-Voice Interactive Systems

The rv-IS formalism is built on top of register machines, closing them with respect to a space-time duality transformation. Specifically, we use the model, the core programming language, the specification formalism and the analysis techniques developed for modeling, programming and reasoning about interactive computing systems by the last author and coworkers in the recent years, see [24,25,21,11,12,23]. In the following, we shortly overview the approach.

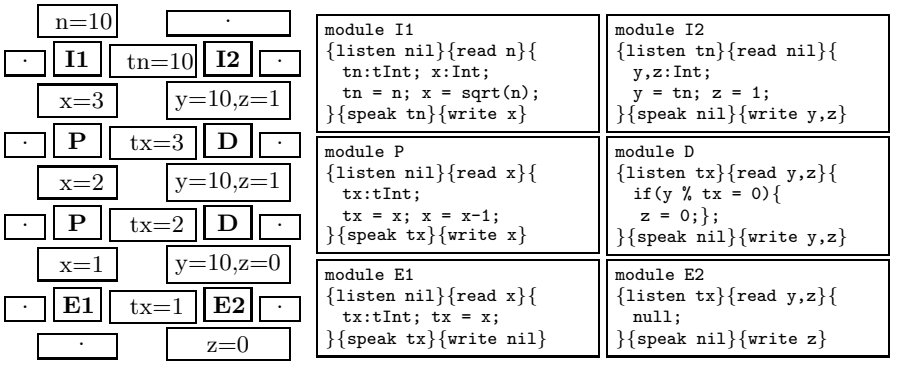


Fig. 1. A scenario and the modules of the **Prime** rv-IS program

Scenarios. A *scenario* is a two-dimensional rectangular area filled in with identifiers and enriched with data around each identifier. In our interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline. This means that a process sends a message to the right, then it resumes its execution. *(Memory) states* are placed at the north and at the south borders of the identifiers and *(interaction) classes* are placed at the west and at the est borders of the identifiers. In the the left-hand side of Fig. 1 we illustrate an rv-IS scenario for deciding whether the number 10 is prime. We explain this example in detail at the end of this section.

Spatio-temporal specifications. A spatio-temporal specification combines constraints on both spatial and temporal data. For the spatial data, we use the common data structures and their natural representations in memory. For representing temporal data we use streams: a *stream* is a sequence of data ordered in time and is denoted as $a_0 \frown a_1 \frown \ldots$, where $a_0, a_1, \ldots$ are the data laying on the stream at time $0, 1, \ldots$, respectively.

A *voice* is defined as the time-dual of a register. Voices are simple temporal structures, represented on streams, that hold natural numbers. The value of a voice can be modified in a location and then propagated within the system. A voice can be "listened" at various locations, at each location the piece of stream representing the voice displaying a particular value. Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. Most of usual data structures have natural temporal representations. Examples includes timed booleans, timed integers (denoted `tInt`), timed arrays, timed lists, etc.

The notation $\otimes$ is used for the product of memory states, while $\frown$ for the product of interaction classes; $\mathbb{N}^{\otimes k}$ denotes $\mathbb{N} \otimes \ldots \otimes \mathbb{N}$ (k terms) and $\mathbb{N}^{\frown k}$ denotes $\mathbb{N} \frown \ldots \frown \mathbb{N}$ (k terms); the associated "star" operations are denoted as $(\_\_^{\otimes})^*$ and $(\_\_^{\frown})^*$.

A simple *spatio-temporal specification* $S : (m, p) \to (n, q)$ is a relation $S \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$, where m (resp. p) is the number of input voices (resp. registers) and n (resp. q) is the number of output voices (resp. registers). More general spatio-temporal specifications may be introduced using complex interface types, not only registers and voices.

Syntax of Structured rv-Programs. The *type* of a *structured rv-program* P, denoted by

$$P : (w(P), n(P)) \to (e(P), s(P)),$$

collects the types at the west, north, east and south borders of its scenarios. In general, these are relatively complex types built up from boolean and integer types - see the concrete types used in Agapia v0.1 programming language [11].

The syntax of structured rv-programs is defined as follows:

```
P ::= null | X | P % P | P # P | P $ P | if(C) then {P} else {P}
        | while_t(C) {P} | while_s(C) {P} | while_st(C) {P}
```

The starting blocks for the construction of structured rv-programs are called *modules*. The syntax of a module is given as follows:

```
module module_name
{listen temporal_variables}{read spatial_variables}{
  code
}{speak temporal_variables}{write spatial_variables}
```

where the `read` (resp. `listen`) instruction collects the spatial (resp. temporal) input and the `write` (resp. `speak`) instruction returns the spatial (resp. temporal) output. The `code` consists in instructions similar to the C code.

The operations on structured rv-programs are briefly described below. More details and examples may be found in [24,11,12].

1. **Composition:** Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types agree. They can also be composed diagonally by mixing the horizontal and vertical composition.

   (a) For two programs $\mathtt{P}_i : (w_i, n_i) \to (e_i, s_i)$, $i = 1, 2$, the *horizontal composition* $\mathtt{P}_1 \# \mathtt{P}_2$ is well-defined only if $e_1 = w_2$; the type of the composite is $(w_1, n_1 \otimes n_2) \to (e_2, s_1 \otimes s_2)$.
   (b) Similarly, the *vertical composition* $\mathtt{P}_1 \% \mathtt{P}_2$ is well-defined only if $s_1 = n_2$; the type of the composite is $(w_1 \frown w_2, n_1) \to (e_1 \frown e_2, s_2)$.
   (c) The *diagonal composition* $\mathtt{P}_1 \$ \mathtt{P}_2$ is a derived operation - it connects the east border of $\mathtt{P}_1$ to the west border of $\mathtt{P}_2$ and the south border of $\mathtt{P}_1$ to the north border of $\mathtt{P}_2$; it is defined only if $e_1 = w_2$ and $s_1 = n_2$; the type of the composite is $(w_1, n_1) \to (e_2, s_2)$.
2. **If:** For the "if" operation, given two programs with the same type $\mathtt{P}, \mathtt{Q} : (w, n) \to (e, s)$, a new program `if(C) then {P} else {Q}` $: (w, n) \to (e, s)$ is constructed, for a condition `C` involving both, the temporal variables in w and the spatial variables in n.
3. **While:** There are three while statements, each being the iteration of the corresponding composition operation.
   (a) For a program $\mathtt{P} : (w, n) \to (e, s)$, the *temporal while* statement `while_t(C){P}` is defined if $n = s$ and C is a condition on the variables in $w \cup n$. The type of the result is $((w\frown)^*, n) \to ((e\frown)^*, n)$.
   (b) The case of *spatial while* `while_s(C){P}` is similar.
   (c) If $\mathtt{P} : (w, n) \to (e, s)$, the statement `while_st(C){P}` is defined if $w = e$ and $n = s$ and C is a condition on $w \cup n$. The type of the result is $(w, n) \to (e, s)$.

Operational Semantics of Structured rv-Programs. The operational semantics is given in terms of scenarios. Scenarios are built up with the following procedure:

1. Each cell of the associated grid has as label a module name.
2. An area around a cell may have additional information. For example, if a cell has the information `x = 2`, that means that in that area `x` is updated to be 2.
3. The scenario is built from the current rv-program by reducing it to simple compositions of spatio-temporal specifications w.r.t. the syntax of the program, until we reach basic blocks, e.g. modules.

Example. We illustrate the operational semantics by considering an example:

```
(I1 # I2) % while_t(x > 1){P # D} % (E1 # E2)
```

This is a structured rv-program **Prime** verifying if a number `n` is prime. Its modules are listed in the right-hand side of Fig. 1.

Our rv-IS program has two processes: one generates all the numbers in the set $\{\lfloor\sqrt{\mathtt{n}}\rfloor, \ldots, 2\}$ (the `P` module) and the other checks if a number is a divisor of `n` (the `D` module) as well as updates a variable `z`. Modules `I1` and `I2` are used for initializations and `E1` and `E2` for ending. At the end of the program, if the variable `z` is `1`, then the number `n` is prime.

In order to show how we can construct a scenario for the rv-IS program above we consider a concrete example for `n = 10`. The corresponding scenario is presented in the right-hand side of Fig. 1. In the first line of the scenario we initialize the processes with the needed information. Module `I1` reads the value `n = 10`, provides the first process with the square root of `n`, i.e., `x = 3`,

and declares a temporal variant of `n`, namely `tn = 10`. This is used by module `I2` to initialize the process with the initial value of `n`, namely `y = 10`; in this module we also set `z = 1` (hence initially, we assume `n` is prime). In the next step, module `P` produces a temporal data `tx = 3` (`tx` is equal with the data `x` of the first process) and decreases `x`. Module `D` verifies if `tx` is a divisor of `y` and, if it is, then it resets the value of `z` to `0`. We repeat these steps until the variable `x` becomes `1`. The last line contains ending modules that only change the interfaces.

The scenarios may be constructed in various ways. For instance, programs building the scenarios by columns [10] exhibit a dataflow computation style.

3 From Event-B to Structured rv-IS

In this section we introduce a general method for translating an Event-B system specification into an rv-IS specification. The method actually produces an rv-program, whenever the transformations used to define the actions of the events can be implemented with a code written in the rv-module code syntax. We also describe shortly our approach to the refinement of rv-IS [10]. In this paper, we are concerned with the events of a certain system, not with its invariants.

An Event-B model can be seen as a set of events of the form presented in the box on the right, where for each `i`, `Event-i` is the name of the event, `Grd-i` is the guard and `Act-i` is the action, so that `Grd-i` and `Act-i` are sets of predicates, respectively actions. We denote with `Ainit` the actions of the event `Init`.

```
Event-i
  when
    Grd-i
  then
    Act-i
  end
```

The rv-IS specification associated to an Event-B model captures not only the model, but also the semantic rules used for its execution. In order to construct a structured rv-IS specification from an Event-B model, we define a *manager* that decides which event can take place at each time. For each event `Event-i`, we construct two modules `Gi` and `Ei` - modules `Gi` are used by the manager in order to decide which event to be triggered, while modules `Ei` are used by the manager to describe the state changes caused by the event.

In Event-B the memory is shared by all the events, hence in the associated rv-IS specification we need to simulate this common memory. Therefore, after each action, the manager must update the variables of all the processes.

Table 1. The formula for EB2IS translated model

```
1:   (I # for_s(j=1,N){ID})
2: $ (Mg # for_s(j=1,N){Gj})
3: $ while_st(ten ≠ ∅) {
4:       (Me # for_s(j=1,N){Ej})
5:     $ (Mu # for_s(j=1,N){U}))
6:     $ (Mg # for_s(j=1,N){Gj})
7:   }
```

Assume that the Event-B model to translate has N events, in addition to the Init event. We define the set $\mathtt{Ev} = \{\mathtt{Ei} \mid \mathtt{i} = \overline{1,\mathtt{N}}\}$ and we denote by C the set of all the constants and by V the set of all the variables of the Event-B model.

The general format of the corresponding rv-IS specification is presented in Table 1. (The for_s statement is derived from while_s in the natural way.)

Module I contains all the initializations from the event Init in Event-B and module ID provides the same variables to all the processes involved in the program. The manager uses the modules Mg, Me and Mu to simulate the behavior in Event-B and to decide which event can take place next. In line 2, the manager constructs the set ten of enabled events by checking their guards; the module Gj checks the guard of the event Event-j. While we have at least one enabled event, we start to simulate its behavior. In line 4, the manager chooses one event from the list of enabled events at the current moment and starts to search for the process modeling the execution of this event. Module Ej modifies in the system with respect to actions Act-j if Event-j is the chosen one. In line 5, the manager updates the variables in all the processes with respect to the new modifications. After this, we repeat the procedure until no more events can occur, as described in line 6.

Table 2. Modules for EB2IS translation

```
module I
{listen nil}{read C,V}{
  Ainit; tV = V ∪ C;
}{speak tV}{write nil}
```

```
module Gi
{listen ten}{read V}{
  if(Grd-i){ten=ten∪{Event-i};};
}{speak ten}{write V}
```

```
module Mg
{listen tV}{read nil}{
  ten = ∅;
}{speak ten}{write nil}
```

```
module ID
{listen tV}{read nil}{
  V = tV;
}{speak tV}{write V}
```

```
module Ei
{listen tk,tV}{read V}{
  if(tk=Event-i){Act-i; tV=V;}
}{speak tk,tV}{write V}
```

```
module Me
{listen ten}{read nil}{
  tk :∈ ten; tV = ∅;
}{speak tk,tV}{write nil}
```

```
module U
{listen tV}{read V}{
  V = tV;
}{speak tV}{write V}
```

```
module Mu
{listen tk,tV}{read nil}{
  null;
}{speak tV}{write nil}
```

The behavior of the manager is split in the following actions: search for the 'chosen' event (lines 2 and 6), modify the system with respect to the actions of the 'chosen' event (line 4), and update the variables of all processes (line 5). In order to take the next action, the manager needs information from the previous action, therefore we must compose the parts of the program diagonally. The modules of the associated rv-IS specifications are described in Table 2.

In this translation the manager decides in an nondeterministic fashion which event can take place next; however, in module Me we can implement any method for deciding this. The manager described above chooses one single event (tk :∈ ten; tk is a single token). In a more general implementation, the manager is free to choose a set of events that can take place at a certain moment of time, by constructing tk to be a set. In such a case, we have to avoid written conflicts for updated variables occurring in more than one event. A general scenario for the program above is presented in Fig. 2.

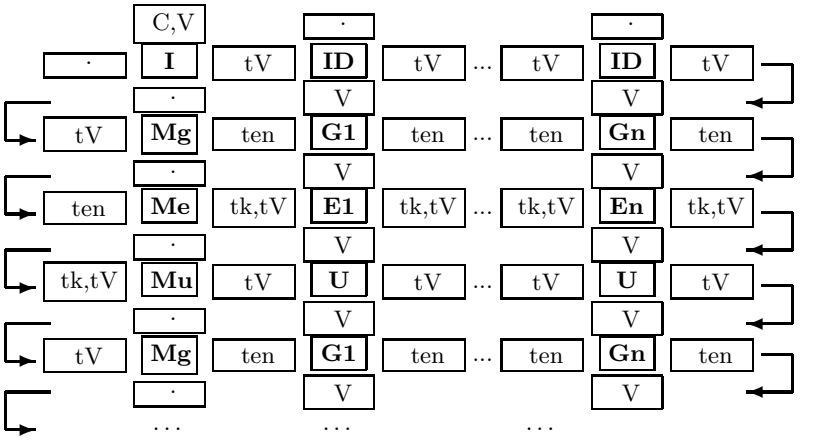


Fig. 2. Scenarios for an rv-IS obtained from an Event-B model

Refinement of register-voice interactive systems. We associate a graph $Gr(S)$ to a scenario S, with the following procedure: (1) we give a proper name to each tuple (w, n, e, s) of data surrounding a scenario cell; a cell is called an *identity* if $(e = w \vee e = n) \wedge (s = n \vee s = w)$; (2) we replace the identifiers of the cells by these names; (3) the graph $Gr(S)$ has as nodes the scenario non-identity cells and as edges connections via identity nodes of their $w/n/e/s$ ports.

Two scenarios $S1$ and $S2$ are *equal up-to-stuttering* of states and classes if the graphs $Gr(S1)$ and $Gr(S2)$ are isomorphic. $S2$ *up-to-stuttering includes* $S2$ if $Gr(S1)$ and $Gr(S2)$ have the same nodes and the edges of $Gr(S1)$ are included in the edges of $Gr(S2)$.

For two rv-IS models $IS1$ and $IS2$, we say $IS2$ is a *refinement* of $IS1$ if: (1) up to a connecting relation between the states and classes of $IS1$ and $IS2$, each scenario of $IS2$ up-to-stuttering includes a scenario of $IS1$; (2) if a scenario of $IS2$ is related to a scenario of $IS1$ and the latter may be extended in $IS1$, then the former may be extended in $IS2$.

As an example, consider the scenarios in Fig. 3. If `Y,Z,U,W` are identity nodes so that `b=d, B=C=D=E` and `F=H`, then $S1$ is up-to-stuttering equal to $S2$.

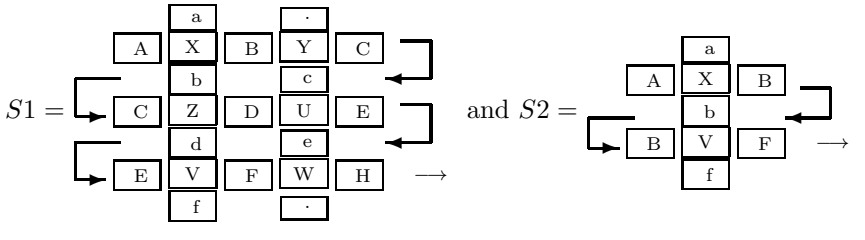


Fig. 3. Two up-to-stuttering equal scenarios

4 An Example – A Simple File Transfer Protocol

In this section we translate an Event-B model into an rv-IS specification.

The model is that of a classical file transfer protocol, also described in [2]. The file to be transferred is sequential, i.e. composed of a finite number of items arranged in a specific order. The file has to be sent from one agent - the sender, to another one - the receiver. The transferred file should be equal to the original one. The protocol is distributed, realized by two distinct modules that exchange various kinds of messages and reside in different sites.

We develop the protocol in three steps. Initially, we are interested only in the final result of the protocol, not in how it is achieved. The file in this model is transmitted in one shot and the agents do not reside on different sites. In the first refinement, we transmit the file piece by piece between the two agents. The main difference with respect to the initial model is that we separate the sender and the receiver. They are still not completely independent, since the receiver can still access the sender's memory. In the second refinement, the sender and the receiver are completely independent from each other and the receiver has no longer access to the sender's memory. In this stage, the two agents communicate only through messages: the sender is sending messages that are read by the receiver and the receiver responds to these messages by returning an acknowledgement message to the sender. The distributed nature of the protocol is therefore revealed in this final refinement step.

Initial Model. We assume a nonempty set D (the carrier set) and two constants: n is a positive number and f is a total function from $\{1, \ldots, n\}$ to D. Informally, f is the file to be transferred, the constant n represents the length of the file f, while D contains the data that can be stored in the file f. We represent the file f as a total function with elements in D. The result of the protocol is a variable g, the file transferred to the receiver. Since we construct g step by step, we model g as a partial function from $\{1, \ldots, n\}$ to D.

In the initial model, we say nothing about the internal structure of the file f. In order to transfer the file, we have an event `receive` that chooses randomly a partial function g with values in D, until this function is equal to f. When we obtain such a function g, then we can assume that the file f was sent to the receiver's site.

The Event-B events of this initial model FTP-EB1 are the following:

FTP-EB1 ::=

```
init
  g :∈ ℕ ↔ D
```

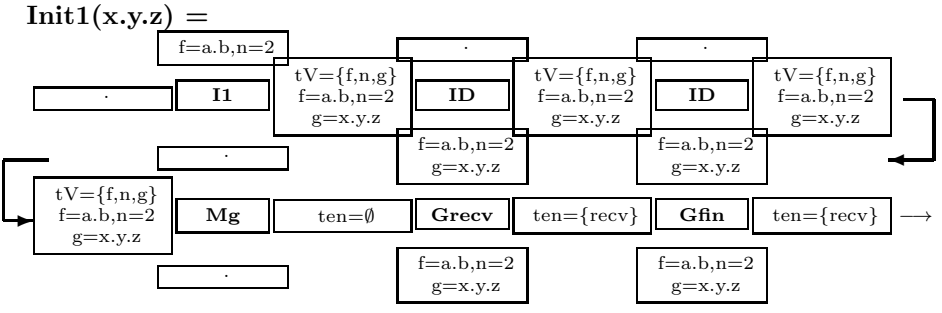
```
receive
  when
    g ≠ f
  then
    g :∈ ℕ ↔ D
  end
```

```
final
  when
    g = f
  then
    skip
  end
```

In order to construct an rv-IS specification FTP-IS1, let us consider the following set of events $\texttt{Ev} = \{\texttt{Erecv}, \texttt{Efin}\}$. The specification is presented in Table 3, where the involved modules `I, Grecv, Erecv, Gfin, Efin` are described in Table 4. In the initial model we have the modules `I, Grecv, Erecv, Gfin,`

`Efin` subscribed by 1, in the second model these modules are subscripted by 2, and in the final model these modules are subscripted by 3.

Let us analyze a simple case: suppose that `f` contains only two characters, say `f=a.b`; thus `n=2`. A typical scenario for the FTP-IS1 specification is built up using partial scenarios illustrated below. In the presentation, `g=x.y.z, g=s.t, ..., g=a.b` is just a sequence of random assignments for `g`. Alternatively, one can consider the case where the lucky assignment `g=a.b` never occurs.



The first scenario `Init1(x.y.z)` (above) is an initialization step that starts with the given data `f,n`. The random assignment `g=x.y.z` generates the initial data for all the processes associated to the events, i.e., for the `recv` and `fin` processes. In addition, the scenario starts to check the validity of the guards: in this case the guard of the `recv` event is true and `recv` is exported on the last line.

Table 3. Formulas for FTP-IS1 and FTP-IS3 specifications

```
FTP-IS1 =
  (I1 # ID # ID)
$ (Mg # Grecv1 # Gfin1)
$ while_st(ten ≠ ∅) {
        (Me # Erecv1 # Efin1)
      $ (Mu # U # U)
      $ (Mg # Grecv1 # Gfin1)
  }
```

```
FTP-IS3 =
  (I3 # ID # ID # ID)
$ (Mg # Grecv3 # Gsend3 # Gfin2)
$ while_st(ten ≠ ∅) {
        (Me # Erecv3 # Esend3 # Efin1)
      $ (Mu # U # U # U)
      $ (Mg # Grecv3 # Gsend3 # Gfin2)
  }
```

The second scenario `DoRecv1(x.y.z;s.t)` (next page) corresponds to the application of the `recv` event, resulting in a state change from `g=x.y.z` to `g=s.t`. Hopefully, after a number of such steps, the random assignment leads to `g=a.b`: in that case, the exported guard in the last line is `fin`, not `recv`. If `fin` holds, then the last scenario `DoFin1` (next page) applies. In this part, the `fin` event has no actions, so nothing changes in the states. Therefore, this repeats forever.

First refinement. In the first refinement, we modify the event `receive` in order to send concrete parts of the file f. The event `receive` will no longer produce files randomly until it obtains one equal with f. Instead, it sends one element of the file f at each step. For this we introduce a new variable r which models an index of the file f. At each step, the r-th element of f is copied in the file g of the receiver's site. The file transfer is finished when r is greater than n.

Table 4. Modules for FTP-IS1 to FTP-IS3 specifications

module I1 {listen nil}{read f,n}{ g :∈ ℕ ↔ D; tV = {f,n,g}; }{speak tV}{write nil}	module I2 {listen nil}{read f,n}{ g = ∅; r = 1; tV = {f,n,g,r}; }{speak tV}{write nil}	module I3 {listen nil}{read f,n}{ g = ∅; r = 1 ; s = 1; d :∈ D; tV = {f,n,g,r,d}; }{speak tV}{write nil}

module Grecv1
{listen ten}{read V}{
if(g ≠ f){
ten = ten ∪ {Erecv1};};
}{speak ten}{write V}
	module Grecv2
	{listen ten}{read V}{
	if(r ≤ n){
	ten = ten ∪ {Erecv2};};
	}{speak ten}{write V}
		module Grecv3
		{listen ten}{read V}{
		if(s = r+1){
		ten = ten ∪ {Erecv3};};
		}{speak ten}{write V}

module Erecv1
{listen tk,tV}{read V}{
if(tk = Erecv1){
g :∈ ℕ ↔D;
tV = V;};
}{speak tk,tV}{write V}
	module Erecv2
	{listen tk,tV}{read V}{
	if(tk = Erecv2){
	g = g ∪ {r ↦ f(r)};
	r = r+1; tV = V;};
	}{speak tk,tV}{write V}
		module Erecv3
		{listen tk,tV}{read V}{
		if(tk = Erecv3){
		g = g ∪ {r↦d}; r = r+1;
		tV = V;};
		}{speak tk,tV}{write V}

module Gfin1
{listen ten}{read V}{
if(g = f){
ten = ten ∪ {Efin1};};
}{speak ten}{write V}
	module Gfin2
	{listen ten}{read V}{
	if(r = n+1){
	ten = ten ∪ {Efin2};};
	}{speak ten}{write V}
		module Gsend3
		{listen ten}{read V}{
		if(s=r ∧ r≠n+1){
		ten = ten ∪ {Esend3};};
		}{speak ten}{write V}

module Efin1
{listen tk,tV}{read V}{
if(tk = Efin1){
tV = V;};
}{speak tk,tV}{write V}
	Efin2 = Efin1
	
	Mg, Me, Mu, U and ID
	are as in Table 2
		module Esend3
		{listen tk,tV}{read V}{
		if(tk = Esend3){
		d = f(s); s = s+1;
		tV = V;};
		}{speak tk,tV}{write V}

The refinement FTP-EB2 of our model in Event-B has the following events:

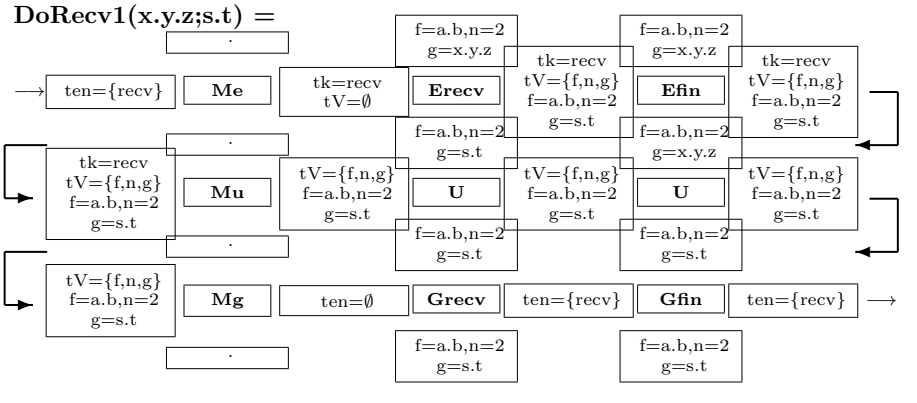
FTP-EB2 ::=

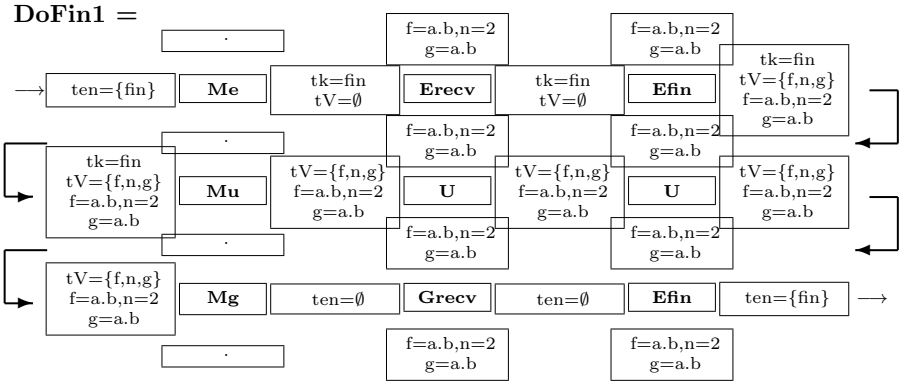
init
$g := \emptyset$
$r := 1$

receive
when
$r \leq n$
then
$g := g \cup \{r \mapsto f(r)\}$
$r := r + 1$
end

final
when
$r = n + 1$
then
skip
end

DoRecv1(x.y.z;s.t) =





The corresponding rv-IS specification FTP-IS2 uses the same formula as in the case of the initial model, but with modules `I1, Grecv1, Gfin1, Erecv1` slightly changed: they are replaced by the new modules `I2, Grecv2, Gfin2, Erecv2` listed in Table 4.

Let us analyze the above case again: suppose that `f=a.b,n=2`. The running scenario is unique (deterministic) this time and consists of an initial action, followed by `n` times repeated `recv` actions, followed by repeated `fin` actions. A detailed presentation appears in [10].

Second refinement. In the last refinement step we split the event `receive` in Event-B into two corresponding events, `send` and `receive`. The indexes s and r model the positions of the current file item (to be) sent and the next position where a file item is to be received, respectively. The event `send` models the activity of the sender, that forms a message d to be sent by copying in d the file item at position s. The event `receive` models the activity of the receiver, that stores the message d as the file item at position r. Hence, we now have a distributed file transfer protocol where the sender and the receiver communicate by sending message d and sharing variables r and s.

FTP-EB3 ::=

```
init
  g = ∅
  s = 1
  r = 1
  d :∈ D
```

```
send
  when
    s = r
    r ≠ n + 1
  then
    d = f(s)
    s = s + 1
  end
```

```
receive
  when
    s = r + 1
  then
    g = g ∪ {r ↦ d}
    r = r + 1
  end
```

```
final
  when
    r = n + 1
  then
    skip
  end
```

For presenting the associated rv-IS specification FTP-IS3, we fix the following set of events $\mathtt{Ev} = \{\mathtt{recv}, \mathtt{send}, \mathtt{fin}\}$, also adding the subscript 3 to indicate that we are at the third modeling level. The specification is presented in Table 3 and the modules in Table 4. The scenarios can be constructed in a similar way as shown for FTP-IS1. The particular case when `f=a.b`, with scenarios named `Init3`, `DoSend3(a)` and `DoRecv3(a)` is discussed in [10].

Refinement Preservation. The state space of FTP-IS1 is $S1 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}\}$, of FTP-IS2 is $S2 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}\}$ and of FTP-IS3 is $S3 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}, \mathtt{s}, \mathtt{d}\}$. The class space of FTP-IS2 is $C2 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r})\}$, departing from that of FTP-IS1 $C1 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g})\}$ by the type of `tV`. The class space of the last model FTP-IS3 is $C3 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}, \mathtt{s}, \mathtt{d})\}$. FTP-IS3 has a new event `send` and the sets used for `ten`, `tk` are larger, including this new element.

Proposition 1. *(a) FTP-IS2 is a refinement of FTP-IS1; and (b) FTP-IS3 is a refinement of FTP-IS2.*

Proof: (Outline) For (a), let $\rho = (\rho_s, \rho_c)$ be a relation between the states and classes of FTP-IS2 and FTP-IS1, where $\rho_s : S2 \to S1$ and $\rho_c : C2 \to C1$ are the natural projections that abstract `r` away. If `Scen` is a scenario in FTP-IS2, then $\rho(\mathtt{Scen})$ is a scenario in FTP-IS1. For (b), let $\rho = (\rho_s, \rho_c)$ be a relation between the FTP-IS3 and FTP-IS2, where $\rho_s : S3 \to S2$ and $\rho_c : C3 \to C2$ are the natural projections that abstract $\mathtt{s}, \mathtt{d}$, `send` away. If `Scen` is a scenario in FTP-IS3, then $\rho(\mathtt{Scen})$ is a scenario in FTP-IS2 up to sub-scenario stuttering corresponding to the application of the macro-steps associated to the `send` event and to the column corresponding to the `send` event. Indeed, this latter sub-scenarios have no visible effect on the states and classes of FTP-IS2. These arguments demonstrate condition (1) in the refinement definition at the end of Section 3.

Condition (2) in this definition is valid for case (a): if `Scen1` is a partial scenario in FTP-IS2 and the scenario $\rho(\mathtt{Scen1})$ can be extended in FTP-IS1, then the same is true for `Scen1` in FTP-IS2. A similar property holds for (b). □

5 Conclusions

Our motivation in this paper is based on one quintessential feature of Event-B and its associated Rodin platform. Modeling in Event-B is semantically justified by proof obligations. Every update of a model generates a new set of proof obligations in the background. It is this interplay between modeling and proving that sets Event-B apart from other formalisms. Without proving the required obligations, we cannot be sure of correctness of a model. The proving effort thus encourages the developer to structure formal model development in such a way that manageable proof obligations are generated at each step. This leads to very abstract initial models so that we can gradually introduce into a system model various facets of the system. Such a development method fits well when we have to describe complex algorithms.

However, it is not obvious how to structure the development of a model, what to model in the initial specification, and what other details to introduce in each of the following refinements. This is especially true when considering the generated proof obligations, because differently structured developments generate different sets of obligations. Several structuring mechanisms have been presented before for Event-B, for instance in [9], to address the complexity of system development. The problem of structuring the development has also been observed

before in the efforts to develop the `Flow`-plugin in the Rodin platform [27], to address the event ordering and enabledness conditions of a model. In this paper we bring forward the enabledness of events as well as the choice of the event to execute next, via the manager modules `Me,Mu,Mg` in rv-IS. Each event is seen as an independent process that is activated when enabled. The interactions between events are 'normalized' to sharing the variables, but in fact new values (hence interactions) occur only upon the execution of an enabled event. This puts forward a clear separation between computation and communication and is resemblant of employing Event-B||CSP in [22], to demonstrate (with the help of the same case study) an explicit approach to control flow.

The main contribution of our paper consists in the definition of a translation EB2IS from Event-B models to structured rv-IS models. Moreover, we provide evidence that the translation preserves refinement, by considering a refinement chain of relatively complex Event-B models and the corresponding translated chain of rv-IS models. As refinement is the fundamental feature of Event-B, this argues in favor of our proposed translation.

Acknowledgement. This work was supported by IST FP7 DEPLOY project, grant agreement 214158.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Design. Cambridge University Press (2010)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer 6, 447–466 (2010)
4. Back, R.J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 131–142 (1983)
5. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier (2001)
6. Broy, M.: Compositional refinement of interactive systems. Journal of the ACM 44, 850–891 (1997)
7. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science 258, 99–129 (2001)
8. Bryans, J., Wei, W.: Formal Analysis of BPMN Models Using Event-B. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 33–49. Springer, Heidelberg (2010)
9. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
10. Diaconescu, D., Leustean, I., Petre, L., Sere, K., Stefanescu, G.: Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems. TUCS Technical Reports No. 1028 (December 2011), `http://tucs.fi`

11. Dragoi, C., Stefanescu, G.: AGAPIA v0.1: A programming language for interactive systems and its typing systems. In: Proc. FINCO/ETAPS 2007. ENTCS, vol. 203, pp. 69–94. Elsevier (2008)
12. Dragoi, C., Stefanescu, G.: On Compiling Structured Interactive Programs with Registers and Voices. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 259–270. Springer, Heidelberg (2008)
13. Salehi Fathabadi, A., Rezazadeh, A., Butler, M.: Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 328–342. Springer, Heidelberg (2011)
14. Hoang, T.S., Fürst, A., Abrial, J.-R.: Event-B Patterns and Their Tool Support. In: Proc. SEFM 2009, pp. 210–219. IEEE (2009)
15. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.: Patterns for Refinement Automation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 70–88. Springer, Heidelberg (2010)
16. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting Reuse in Event B Development: Modularisation Approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
17. Kamali, M., Petre, L., Sere, K., Daneshtalab, M.: Refinement-Based Modeling of 3D NoCs. In: Sirjani, M. (ed.) FSEN 2011. LNCS, vol. 7141, pp. 236–252. Springer, Heidelberg (2011)
18. Kamali, M., Petre, L., Sere, K., Daneshtalab, M.: Formal Modeling of Multicast Communication in 3D NoCs. In: Proc. DSD 2011, pp. 634–642. IEEE (2011)
19. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
20. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes I and II. Information and Computation 100(1), 1–77 (1992)
21. Popa, A., Sofronia, A., Stefanescu, G.: High-level structured interactive programs with registers and voices. JUCS 13, 1722–1754 (2007)
22. Schneider, S., Treharne, H., Wehrheim, H.: Bounded Retransmission in Event-B||CSP: a Case Study. ENTSC 280, 69–80 (2011)
23. Sofronia, A., Popa, A., Stefanescu, G.: Undecidability Results for Finite Interactive Systems. ROMJIST 12, 265–279 (2009); Also: Arxiv, CoRR 1001.0143 (2010)
24. Stefanescu, G.: Interactive systems with registers and voices. Fundamenta Informaticae 73, 285–306 (2006)
25. Stefanescu, G.: Towards a Floyd logic for interactive rv-systems. In: Proc. ICCP 2006, pp. 169–178. TU Cluj-Napoca (2006)
26. URL, http://www.petrinets.info/
27. URL RODIN tool platform, http://www.event-b.org/platform.html
28. Waldén, M., Sere, K.: Reasoning About Action Systems Using the B-Method. FMSD 13, 5–35 (1998)
29. Wegner, P.: Interactive foundations of computing. TCS 192, 315–351 (1998)

Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B

Anton Tarasyuk[1,2], Elena Troubitsyna[2], and Linas Laibinis[2]

[1] Turku Centre for Computer Science, Turku, Finland
[2] Åbo Akademi University, Turku, Finland
{anton.tarasyuk,elena.troubitsyna,linas.laibinis}@abo.fi

Abstract. Modelling and refinement in Event-B provides a scalable support for systematic development of complex service-oriented systems. This is achieved by a gradual transformation of an abstract service specification into its detailed architecture. In this paper we aim at integrating quantitative assessment of essential quality of service attributes into the formal modelling process. We propose an approach to creating and verifying a dynamic service architecture in Event-B. Such an architecture can be augmented with stochastic information and transformed into the corresponding continuous-time Markov chain representation. By relying on probabilistic model-checking techniques, we allow for quantitative evaluation of quality of service at early development stages.

1 Introduction

The main goal of service-oriented computing is to enable rapid building of complex software by assembling readily-available services. While promising productivity gain in the development, such an approach also poses a significant verification challenge – how to guarantee correctness of complex composite services?

In our previous work we have demonstrated how to build complex service-oriented systems (SOSs) by refinement in Event-B [12,11]. We have not only formalised Lyra – an industrial model-driven approach – but also augmented it with a systematic modelling of fault tolerance. However, within this approach we could not evaluate whether the designed fault tolerant mechanisms are appropriate, i.e., they suffice to meet the desired quality of service (QoS) attributes.

To address this issue, in this paper we propose an approach to building a dynamic service architecture – an Event-B model that formally represents the service orchestration. In particular, we define the set of requirements – the formal verification conditions – that allow us to ensure that the modelled service architecture faithfully represents the dynamic service behaviour. Such an Event-B model can be then augmented with stochastic information about system failures and duration of the orchestrated services. Essentially, this results in creating a continuous-time Markov chain (CTMC) model representation and hence enables the use of existing probabilistic model checking techniques to verify the desired

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 237–252, 2012.

QoS attributes. We demonstrate how to formulate a number of widely used QoS attributes as temporal logic formulae to be verified by PRISM [14]. Overall, our approach enables an early quantitative evaluation of essential QoS attributes and rigorous verification of the dynamic aspects of the system behaviour.

The paper is organised as follows. In Section 2 we briefly describe our formal modelling framework, Event-B, and also define its underlying transition system. Section 3 discusses SOSs and the associated dynamic service architectures. In Section 4 we propose a set of necessary requirements for SOSs as well as their formalisation in Event-B. Section 5 presents a small case study that illustrates building a dynamic service architecture. In Section 6 we explain how to convert Event-B models into CTMCs and also demonstrate the use of probabilistic model checking for analysis of QoS attributes. Finally, Section 7 gives some concluding remarks as well as overviews the related work in the field.

2 Modelling in Event-B

Event-B is a formal framework derived from the (classical) B method [1] to model parallel, distributed and reactive systems [2]. The Rodin platform provides tool support for modelling and formal verification (by theorem proving) in Event-B [17]. Currently, Event-B is used in the EU project Deploy to model dependable systems from from automotive, railway, space and business domains [9].

In Event-B, a system specification is defined using the notion of an *abstract state machine*. An abstract state machine encapsulates the model state, represented as a collection of model variables, and defines operations on this state via machine *events*. The occurrence of events represents the system behaviour. In a most general form, an Event-B model can be defined as follows.

Definition 1. *An Event-B model is a tuple* $(\mathcal{C}, \mathcal{S}, \mathcal{A}, \upsilon, \mathcal{I}, \Sigma, \mathcal{E}, Init)$ *where:*

- $\mathcal{C}$ *is a set of model constants;*
- $\mathcal{S}$ *is a set of model sets;*
- $\mathcal{A}$ *is a set of axioms over* $\mathcal{C}$ *and* $\mathcal{S}$*;*
- υ *is a set of system variables;*
- $\mathcal{I}$ *is a set of invariant properties over* υ*,* $\mathcal{C}$ *and* $\mathcal{S}$*;*
- Σ *is a model state space defined by all possible values of the vector* υ*;*
- $\mathcal{E} \subseteq \mathcal{P}(\Sigma \times \Sigma)$ *is a non-empty set of model events;*
- $Init$ *is a predicate defining an non-empty set of model initial states.*

The model variables υ are strongly typed by the constraining predicates specified in $\mathcal{I}$ and initialised by the predicate $Init$. Furthermore, $\mathcal{I}$ defines important properties that must be preserved by the system during its execution.

Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, and the *guard* G_e is a predicate over the model variables. The R_e is a next-state relation called *generalised substitution*. The guard defines the conditions under which the substitution

can be performed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

If an event does not have local variables, it can be described simply as

$$e \mathrel{\widehat{=}} \textbf{when } G_e \textbf{ then } R_e \textbf{ end}.$$

Throughout the paper we will consider only such simple events. It does not make any impact on the generality of our approach because any event specified by using local variables can be always rewritten in the simple form.

Essentially, such an event definition is just a syntactic sugar for the underlying relation $e(\sigma, \sigma') = G_e(\sigma) \wedge R_e(\sigma, \sigma')$. Generally, a substitution R_e is defined by a *multiple* (possibly nondeterministic) assignment over a vector of system variables $u \subseteq v$, i.e., $u := X$, for some vector of values X. Hence the state transformation (via R_e) can be intuitively defined as $R_e(\sigma, \sigma') \Rightarrow \sigma' = \sigma[X/u]$, where $\sigma[X/u]$ is a substitution of values of the variables u in σ by the vector X. Obviously, due to presence of nondeterminism the successor state σ' is not necessarily unique.

For our purposes, it is convenient to define an Event-B model as a transition system. To describe a state transition for an Event-B model, we define two functions before and after from $\mathcal{E}$ to $\mathcal{P}(\Sigma)$ in a way similar to [8]:

$$\mathsf{before}(e) = \{\sigma \in \Sigma \mid \mathcal{I}(\sigma) \wedge G_e(\sigma)\} \quad \text{and}$$
$$\mathsf{after}(e) = \{\sigma' \in \Sigma \mid \mathcal{I}(\sigma') \wedge \big(\exists \sigma \in \Sigma \cdot \mathcal{I}(\sigma) \wedge G_e(\sigma) \wedge R_e(\sigma, \sigma')\big)\}.$$

These functions essentially return the domain and the range of an event e constrained by the model invariants $\mathcal{I}$. It is easy to see that e is enabled in σ if $\sigma \in \mathsf{before}(e)$. At any state σ, the behaviour of an Event-B machine is defined by all the enabled in σ events.

Definition 2. *The behaviour of any Event-B machine is defined by a transition relation* $\rightarrow$*:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \rightarrow \sigma'},$$

where $\mathcal{E}_\sigma = \{e \in \mathcal{E} \mid \sigma \in \mathsf{before}(e)\}$ *is a subset of events enabled in* σ*.*

Remark 1. The soundness of Definition 2 is guaranteed by the *feasibility* property of Event-B events. According to this property, such σ' should always exist for any $\sigma \in \mathsf{before}(e)$, where $e \in \mathcal{E}_\sigma$ [2].

Together Definitions 1 and 2 allow us to describe any Event-B model as a transition system with state space Σ, transition relation $\rightarrow$ and a set of initial states defined by $Init$. Next we describe the essential structure of SOSs and reflect on our experience in modelling SOSs in Event-B.

3 Service-Oriented Systems

3.1 Service Orchestration

Service-oriented computing is a popular software development paradigm that facilitates building complex distributed services by coordinated aggregation of

lower-level services (called subservices). Coordination of a service execution is typically performed by a *service director* (or *service composer*). It is a dedicated software component that on the one hand, communicates with a service requesting party and on the other hand, orchestrates the service execution flow.

To coordinate service execution, the service director keeps information about subservices and their execution order. It requests the corresponding components to provide the required subservices and monitors the results of their execution. Let us note that any subservice might also be composed of several subservices, i.e., in its turn, the subservice execution might be orchestrated by its (sub)service director. Hence, in general, a SOS might have several layers of hierarchy.

Often, a service director not only ensures the predefined control and data flow between the involved subservices but also implements fault-tolerance mechanisms. Indeed, an execution of any subservice might fail. Then the service director should analyse the failed response and decide on the course of error recovery actions. For instance, if an error deemed to be recoverable, it might repeat the request to execute the failed subservice. However, it might also stop execution of a particular subservice to implement coordinated error recovery or abort the whole service execution due to some unrecoverable error.

Let us consider a simple case when the involved subservices S_1, S_2, ..., S_n should be executed in a fixed sequential order:

$$S_1 \longrightarrow S_2 \longrightarrow S_3 .... \longrightarrow S_n$$

According the discussion above, the overall control flow can be graphically represented as shown in Fig. 1.

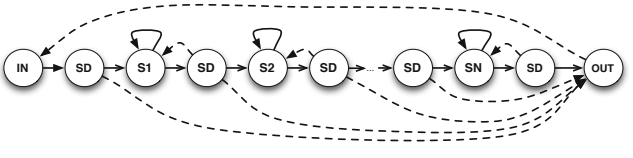


Fig. 1. Service flow in a service director

Here IN and OUT depict receiving new requests and sending service responses, while the service director SD monitors execution of the subservices and performs the required controlling or error recovery actions. These actions may involve requesting a particular subservice to repeat its execution (a dashed arrow from SD to S_i), aborting the whole service (a dashed arrow from SD to OUT), or allowing a subservice to continue its execution (a looping arrow for S_i).

Though we have considered a sequential service execution flow, the service execution per se might have any degree of parallelism. Indeed, any subservice might consist of a number of independent subservices $S_{i1}, ..., S_{ik}$ that can be executed in parallel. Such a service architecture allows the designer to improve performance or increase reliability, e.g., if parallel subservices replicate each other.

3.2 Towards Formalisation of Service Orchestration

In our previous work [12,11], we have proposed a formalisation of the service-oriented development method Lyra in the B and Event-B frameworks. In our approach, refinement formalises unfolding of architectural layers and consequently introduces explicit representation of subservices at the corresponding architectural layer. Reliance of refinement, decomposition and proof-based verification offers a scalable support for development of complex services and verification of their functional correctness. The result of refinement process is a detailed system specification that can be implemented on a targeted platform. However, before such an implementation is undertaken, it is desirable to evaluate whether the designed service meets its QoS requirements.

To enable such an evaluation, we propose to build a formal model that *explicitly* represents service orchestration, i.e., defines the *dynamic service architecture*, while suppressing unnecessary modelling details. Such a model can be augmented with probabilistic information and serve as an input for the evaluation of the desirable QoS attributes, as we will describe in Section 6.

To achieve this goal, we should strengthen our previous approach by formalising service orchestration requirements. Indeed, in [12,11] the service execution flow and possible parallelism were modelled via an abstract function *Next*. Essentially, this function served as an abstract scheduler of subservices. However, such a representation does not allow for a verification of service orchestration that is essential for building an adequate model of the dynamic service architecture.

We start our formalisation of service orchestration requirements by assuming that a service S is composed of a finite set of subservices $\{S_1, S_2, \ldots, S_n\}$ that are orchestrated by a service director. The behaviour of the service director consists of a number of activities $\{IN, OUT, SD\}$, where IN and OUT are modelling the start (i.e., receiving a service request) and the end (i.e., sending a service response) of the service execution flow. SD represents the decision making procedure performed by the service director after execution of any subservice, i.e., it computes whether to restart the execution of the current subservice, call the next scheduled subservice, or abort the service execution.

In Event-B, the subservices $S_1, ..., S_n$ as well as the service director activities IN, OUT, SD can be represented as groups of mutually exclusive events. Without losing generality, we will treat all these activities as single events.

Let us also tailor our generic definitions of before and after functions to modelling service-oriented systems. For a composite subservice S_i, i.e., a subservice that is a parallel composition of sub-subservices $S_{i1}, ..., S_{ik}$, we define

$$\mathsf{before}(S_i) = \bigcup_{j \in 1..k} \mathsf{before}(S_{ij}) \quad \text{and} \quad \mathsf{after}(S_i) = \bigcup_{j \in 1..k} \mathsf{after}(S_{ij}).$$

Moreover, we introduce a version of the function after that is "narrowed" with respect to a particular fixed pre-state σ:

$$\mathsf{after}_\sigma(e) = \{\sigma' \in \Sigma \mid \mathcal{I}(\sigma') \wedge \mathcal{I}(\sigma) \wedge G_e(\sigma) \wedge R_e(\sigma, \sigma')\}.$$

Essentially, this function gives the relational image of the next-state relation R_e for the given singleton set $\{\sigma\}$. We will rely on these definitions while postulating the service orchestration conditions that we present next.

4 Modelling the Dynamic Service Architecture

In this paper, we focus on modelling of SOSs that can provide service to only one request at any time instance. In other words, it means that once a SOS starts to serve a request, it becomes unavailable for the environment until the current service is finished or aborted. Such a discipline of service imposes the following requirements for receiving a service request and sending a service response:

(REQ1) *After receiving a service request, the service director is activated to handle it;*
(REQ2) *Once the service execution is finished, the service director is ready to receive a new service request;*
(REQ3) *Receiving a service request and sending a service response is not possible when orchestration of the service execution is still in process.*

Formally, the first two requirements can be formulated as follows:

$$\mathsf{after}(IN) \subseteq \mathsf{before}(SD) \quad \text{and} \quad \mathsf{after}(OUT) \subseteq \mathsf{before}(IN),$$

while the formalisation of (REQ3) can be defined by the following two predicates:

$$\forall e_1, e_2 \in \{IN, OUT, SD\} \cdot \mathsf{before}(e_1) \cap \mathsf{before}(e_2) = \varnothing,$$
$$\big(\mathsf{before}(IN) \cup \mathsf{before}(OUT)\big) \cap \big(\bigcup_{i \in 1..n} \mathsf{before}(S_i)\big) = \varnothing.$$

Essentially, these predicates state that IN and OUT cannot be enabled at the same time as any of subservices S_i or the service director event SD.

Moreover, the service director should follow the predefined order of the service execution. This, however, should not prevent the service director from interfering:

(REQ4) *At any moment only one of sequential subservices (i.e., only one of $S_1, ..., S_n$) can be active;*
(REQ5) *The service director has an ability to always provide a required control action upon execution of the active subservice (or any of parallel subservices);*

Formally, (REQ4) is ensured by requiring that the sets of states, where any two different subservices are enabled, are disjoint:

$$\forall i, j \in 1..n \cdot i \neq j \Rightarrow \mathsf{before}(S_i) \cap \mathsf{before}(S_j) = \varnothing,$$

while (REQ5) implies that SD can be enabled by execution of any subservice:

$$\forall i \in 1..n \cdot \mathsf{after}(S_i) \subseteq \mathsf{before}(S_i) \cup \mathsf{before}(SD).$$

Let us note that we delegate a part of the service director activity to the guards of the events modelling the subservices. Specifically, we allow a subservice to be executed in a cyclic manner without any interference from the service director, i.e., it may either block itself or continue its activity if it remains enabled.

However, if an active subservice blocks itself, it must enable the service director:

$$\forall i \in 1..n, \sigma \in \Sigma \cdot \sigma \in \mathsf{after}(S_i) \wedge \sigma \notin \mathsf{before}(S_i) \Rightarrow \sigma \in \mathsf{before}(SD)$$

that directly follows from the formalisation of (REQ5).

The next requirement concerns handling performed by the service director.

(REQ6) *The reaction of the service director depends on the result returned by the supervised subservice (or several parallel subservices) and can be one of the following:*

- *upon successful termination of the subservice, the service director calls the next scheduled subservice;*
- *in case of a recoverable failure of the subservice, the service director restarts it;*
- *in case of an unrecoverable failure (with respect to the main service) of the subservice, the service director aborts the execution of the whole service.*

Formally, it can be specified in the following way:

$$\forall i \in 1..n-1; \sigma \in \Sigma \cdot \sigma \in \mathsf{after}(S_i) \cap \mathsf{before}(SD) \Rightarrow \mathsf{after}_\sigma(SD) \subseteq \mathsf{before}(S_i) \cup \mathsf{before}(S_{i+1}) \cup \mathsf{before}(OUT)$$

and, for a special case when a subservice is the last one in the execution flow,

$$\forall \sigma \in \Sigma \cdot \sigma \in \mathsf{after}(S_n) \cap \mathsf{before}(SD) \Rightarrow \mathsf{after}_\sigma(SD) \subseteq \mathsf{before}(S_n) \cup \mathsf{before}(OUT).$$

In addition, it is important to validate that execution of any subservice cannot disable execution of the service director:

(REQ7) *None of active subservices can block execution of the service director.*

It means that, whenever SD is enabled, it stays enabled after execution of any subservice:

$$\forall i \in 1..n; \sigma \in \Sigma \cdot \sigma \in \mathsf{before}(SD) \cup \mathsf{before}(S_i) \Rightarrow \mathsf{after}_\sigma(S_i) \subseteq \mathsf{before}(SD).$$

In the general case, when the execution flow of a SOS contains parallel compositions of subservices, a couple of additional requirements are needed:

(REQ8) *All the subservices of a parallel composition must be independent of each other, i.e., their execution order does not affect the execution of the overall service;*

(REQ9) *Execution of any subservice of a parallel composition cannot block execution of any other active parallel subservice;*

In terms of model events, the independence requirement means that forward relational composition of two different events running in any order have the same range:

$$\forall i \in 1..n; j, l \in 1..k \cdot \mathsf{after}(S_{ij}; S_{il}) = \mathsf{after}(S_{il}; S_{ij}),$$

where

$$\mathsf{after}(e_i; e_j) = \{\sigma'' \in \Sigma \mid \sigma'' \in \mathsf{after}(e_j) \wedge (\exists \sigma, \sigma' \in \Sigma \cdot \sigma \in \mathsf{before}(e_i) \wedge \sigma' \in \mathsf{after}(e_i) \cap \mathsf{before}(e_j) \wedge R_{e_i}(\sigma, \sigma') \wedge R_{e_j}(\sigma', \sigma''))\}.$$

Such a definition of independence is imposed by the interleaving semantics of Event-B and the fact that the framework does not support events composition directly. Finally, we formulate the requirement (REQ9) as follows:

$$\forall i \in 1..n; j,l \in 1..k \cdot j \neq l \wedge \\ \sigma \in \mathsf{before}(S_{ij}) \cup \mathsf{before}(S_{il}) \Rightarrow \mathsf{after}_{\sigma}(S_{ij}) \subseteq \mathsf{before}(S_{il})$$

$$\text{and} \quad \forall i \in 1..n; j \in 1..k \cdot \mathsf{after}(S_{ij}) \cap \mathsf{before}(S_{ij}) \neq \varnothing,$$

where the second formula states that any subservice can continue its execution without interference from the service director.

To verify that an Event-B model of a SOS satisfies the requirements (REQ1)–(REQ9), their formalisation (based on concrete model elements) could be generated and added as a collection of model theorems. A similar approach has been applied in [8]. The generation and proof of additional model theorems can be partially automated, provided that the mapping between the model events and the subservices as well as the activities IN, OUT, and SD is supplied.

In the next section we consider a small example of a SOS. To demonstrate our approach, we formally model the system dynamic architecture in Event-B and then show that the model satisfies the formulated flow conditions.

5 Case Study

We model a simple dynamic service architecture that consists of a service director and five subservices. The latter can be structured into two composite subservices, S_1 and S_2, where S_1 is a parallel composition of the subservices S_{11}, S_{12} and S_{13}, while S_2 consists of the parallel subservices S_{21} and S_{22}. Next we define the fault assumptions and decision rules used by the service director. Besides (fully) *successful* termination of subservices, the following alternatives are possible:

- the subservice S_{11} can terminate with a *transient failure*, in which case its execution should be restarted. The total number of retries cannot exceed the predefined upperbound number MAX;
- the subservices S_{12} and S_{13} can terminate with a *permanent failure*. Moreover, each of them may return a *partially successful* result, complementing the result of the other subservice;
- the subservices S_{21} and S_{22} can also terminate with a *permanent failure*. These subservices are functionally identical, thus successful termination of one of them is sufficient for the overall success of S_2.

Fig. 2 shows the events that abstractly model the behaviour of subservices. The variables srv_{ij}, where $i \in 1..2$ and $j \in 1..3$, represent statuses of the corresponding subservices. Here, the value *nd* (meaning "not defined") is used to distinguish between the subservices that are currently inactive, and those that are active but have not yet returned any result. The value *nok* stands for a permanent failure of a subservice, while the values *ok* and *pok* represent respectively successful and partially successful subservice execution.

Variables $cnt, srv_{11}, srv_{12}, srv_{13}, srv_{21}, srv_{22}$
Invariants $cnt \in \mathbb{N} \wedge srv_{11}, srv_{21}, srv_{22} \in \{ok, nok, nd\} \wedge srv_{12}, srv_{13} \in \{ok, nok, pok, nd\}$
Events
$subsrv_{11} \mathrel{\widehat{=}}$
when $active = 1 \wedge srv_{11} = nd$
then $srv_{11} :\in \{nd, ok\}$
$cnt := cnt + 1$ **end**
$subsrv_{12} \mathrel{\widehat{=}}$
when $active = 1 \wedge srv_{12} = nd$
then $srv_{12} :\in \{ok, nok, pok\}$ **end**
$subsrv_{13} \mathrel{\widehat{=}} \cdots$
$subsrv_{21} \mathrel{\widehat{=}}$
when $active = 2 \wedge srv_{21} = nd$
then $srv_{21} :\in \{nd, ok\}$ **end**
$subsrv_{22} \mathrel{\widehat{=}} \cdots$

Fig. 2. Case study: modelling subservices in Event-B

Initially all the subservices have the status *nd*. Note that, in case of a transient failure of S_{11}, the value of srv_{11} remains *nd* and, as a result, the subservice can be restarted. The counter variable *cnt* stores the number of retries of S_{11}.

To provide the overall service, the following necessary conditions must be satisfied:

- S_{11} returns a successful result within MAX retries;
- both S_{12} and S_{13} do not fail and at least one of them returns a (fully) successful result;
- at least one of S_{21} and S_{22} does not fail.

The service director controls execution of subservices and checks preservation of these conditions. The events modelling behaviour of the service director are shown in Fig. 3. Here, the boolean variable *idle* stores the status of the overall service, i.e., whether the service director is waiting for a new service request or the service execution is already in progress. The variable *active* indicates which group of subservices or which activity of the service director is currently enabled. The value of the boolean variable *abort* shows whether one of the conditions necessary for successful completion of the service has been violated and thus the service execution should be interrupted. Finally, the variable $failed$ counts the number of such interrupted service requests. Please note that the service director activities $sd\_success_1$ and sd_fail_1 may run in parallel with $subsrv_{11}$. On the other hand, $sd\_success_2$ can be activated even if only one of $subsrv_{21}$ and $subsrv_{22}$ has been successfully executed and the other one is still running.

For our case study, it is easy to prove that the presented model satisfies the requirements (REQ1)–(REQ9). For instance, a proof of (REQ7) preservation can be split into two distinct cases, when $active = 1$ and $active = 2$. Let us consider the first case. When $\sigma \in \mathsf{before}(sd\_success_1)$, then all the subservices' events are disabled and the requirement is obviously satisfied. When $\sigma \in \mathsf{before}(sd\_fail_1)$, at least one of the disjuncts in the guard of sd_fail_1 is satisfied. In the case when $cnt > MAX$ is true, $subsrv_{11}$ can only increase the value of *cnt*, and neither $subsrv_{12}$ nor $subsrv_{13}$ can modify it. In the case when either $(srv_{12} = nok \vee srv_{13} = nok)$ or $(srv_{12} = pok \wedge srv_{13} = pok)$ is true, both $subsrv_{12}$ and $subsrv_{13}$ are disabled, and since $subsrv_{11}$ cannot affect any of these two guards the requirement is satisfied. Overall, the formal proofs for this model are simple though quite tedious.

Variables $idle, active, abort, failed$
Invariants $idle \in BOOL \wedge active \in 0..3 \wedge abort \in BOOL \wedge failed \in \mathbb{N}$
Events

$in \mathrel{\widehat{=}}$
when $active = 0 \wedge idle = TRUE$
then $idle := FALSE$ **end**

$sd_{in} \mathrel{\widehat{=}}$
when $active = 0 \wedge idle = FALSE$
then $active := 1$ **end**

$out_{success} \mathrel{\widehat{=}}$
when $active = 3 \wedge abort = FALSE$
then $active, cnt := 1, 0$
$idle := TRUE$
$srv_{11} := nd$
$srv_{12} := nd \ \ldots$ **end**

$out_{fail} \mathrel{\widehat{=}}$
when $active = 3 \wedge abort = TRUE$
then $\ldots$
$failed := failed + 1$
$abort := FALSE$ **end**

$sd\_success_1 \mathrel{\widehat{=}}$
when $active = 1 \wedge$
$srv_{11} = ok \wedge cnt \leq MAX \wedge$
$srv_{12} \neq nok \wedge srv_{13} \neq nok \wedge$
$(srv_{12} = ok \vee srv_{13} = ok)$
then $active := 2$ **end**

$sd\_fail_1 \mathrel{\widehat{=}}$
when $active = 1 \wedge (cnt > MAX \vee$
$srv_{12} = nok \vee srv_{13} = nok \vee$
$(srv_{12} = pok \wedge srv_{13} = pok))$
then $active, abort := 3, TRUE$ **end**

$sd\_success_2 \mathrel{\widehat{=}}$
when $active = 2 \wedge$
$(srv_{21} = ok \vee srv_{22} = ok)$
then $active := 3$ **end**

$sd\_fail_2 \mathrel{\widehat{=}}$
when $active = 2 \wedge$
$srv_{21} = nok \wedge srv_{22} = nok$
then $active, abort := 3, TRUE$ **end**

Fig. 3. Case study: modelling the service director in Event-B

As we have mentioned in the previous section, the verification of an Event-B model against the formulated requirements (`REQ1`)–(`REQ9`) is based on generation and proof of a number of Event-B theorems in the Rodin platform. However, for more complex, industrial-size systems, it can be quite difficult to prove such theorems in Rodin. To tackle this problem, some external mechanised proving systems, such as HOL or Isabelle, can be used. Bridging the Rodin platform with such external provers is currently under development.

The goal of building a model of dynamic service architecture is to enable quantitative evaluation of QoS attributes. In the next section we show how an Event-B machine can be represented by a CTMC and probabilistic model checking used to achieve the desired goal.

6 Probabilistic Verification in Event-B

6.1 Probabilistic Event-B

In this paper, we aim at quantitative verification of QoS of SOSs modelled in Event-B. To perform such a verification, we will transform Event-B models defining dynamic service architecture into CTMCs. The properties that we are interested to verify are the time-bounded reachability and reward properties related to a possible abort of service execution. For continuous-time models, such probabilistic properties can be specified as CSL (Continuous Stochastic Logic) formulae [3,4]. A detailed survey and specification patterns for probabilistic properties can be found in [7]. There are several examples of properties of SOSs that can be interesting for verification:

- what is the probability that at least one service execution will be aborted during a certain time interval?
- what is the probability that a number of aborted services during a certain time interval will not exceed some threshold?
- what is the mean number of served requests during a certain time interval?
- what is the mean number of failures of some particular subservice during a certain time interval?

To transform an Event-B machine into a CTMC, we augment all the events with information about probability and duration of all the actions that may occur during its execution. More specifically, we refine all the events by their probabilistic counterparts.

Let us consider a system state $\sigma \in \Sigma$ and an event $e \in \mathcal{E}$ such that $\sigma \in \mathsf{before}(e)$. Assume that R_e can transform σ to a set of states $\{\sigma'_1, \ldots, \sigma'_m\}$, where $m \geq 1$. Please recall that in Event-B, if $m > 1$ then the choice between the successor states is nondeterministic. We augment every such state transformation with a constant *rate* $\lambda_i \in \mathbb{R}^+$, where λ_i is a parameter of *the exponentially distributed sojourn time* that the system will spend in the state σ before it goes to the new state σ'_i. In such a way, we can replace a nondeterministic choice between the possible successor states by the probabilistic choice associated with the (exponential) race condition.

It is easy to show that such a replacement is a valid refinement step. Indeed, let p_i be a probability to choose a state transformation $\sigma \to \sigma'_i$, $\sigma \notin \{\sigma'_1, \ldots, \sigma'_m\}$. For $i \in 1..m$, it is convenient to define p_i as:

$$p_i = \frac{\lambda_i}{\sum\limits_{j=1}^{m} \lambda_j}.$$

The probabilities p_i define a next-state distribution for the current state σ. Refinement of the nondeterministic branching by the (discrete) probabilistic one is a well-known fact (see [15] for instance), which directly implies the validity of the refinement.

We adopt the notation $\lambda_e(\sigma, \sigma')$ to denote the transition rate from σ to σ' via the event e, where $\sigma \in \mathsf{before}(e)$ and $R_e(\sigma, \sigma')$. Augmenting all the event actions with transition rates, we can respectively modify Definition 2 as follows.

Definition 3. *The behaviour of any probabilistically augmented Event-B machine is defined by a transition relation* $\xrightarrow{\Lambda}$*:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \xrightarrow{\Lambda} \sigma'},$$

where $\Lambda = \sum\limits_{e \in \mathcal{E}_\sigma} \lambda_e(\sigma, \sigma')$.

With such a probabilistic transition relation, an Event-B machine becomes a CTMC, whereas p_i are the one-step transition probabilities of the embedded (discrete-time) Markov chain. Such an elimination of nondeterminism between

enabled events is not always suitable for modelling. However, for SOSs this assumption seems quite plausible. Indeed, the fact that execution of two or more simultaneously enabled services may "lead" to the same state usually means that all these services share the same functionality. In this situation it is natural to expect that the overall transition rate will increase and thus summing of the corresponding subservice rates looks absolutely essential. Moreover, for parallel composition of subservices, the interleaving semantics of Event-B perfectly coheres with the fact that the probability that two or more exponentially distributed transition delays elapse at the same time moment is zero.

Generally, we can assume that $\sigma \in \{\sigma'_1, \ldots, \sigma'_m\}$ and attach a rate for this *skip* transformation as well. While participating in the race, such a transition does not affect it (because it does not change the system state and due to the memoryless property of the exponential distribution). However, the skip transition can be useful for verification of specific reward properties, e.g., the number of restarts for a particular subservice, the number of lost customers in the case of buffer overflow, etc. Obviously, such a transition is excluded from the calculation of p_i.

6.2 Case Study: Quantitative Modelling and Verification

Now let us perform quantitative verification of QoS attributes of the SOS presented in our case study using the probabilistic symbolic model checker PRISM. We start by creating a PRISM specification corresponding to our Event-B model. Short guidelines for Event-B to PRISM model transformation can be found in our previous work [19]. Fig. 4 and Fig. 5 show the resulting PRISM model as well as the rates we attached to all the model transitions. The behaviour of subservices is modelled by two modules S_1 and S_2. Note that the rate of successful execution of S_{11} is decreasing with the number of retries.

In Fig. 5, the modules SD and IN_OUT model behaviour of the service director. Since the model checker cannot work with infinite sets we have bounded from above the number of interrupted service requests by the predefined constant value MAX_failed. Such a restriction is reasonable because when the number of interrupted service requests exceeds some acceptable threshold, the system is usually treated as unreliable and must be redesigned.

Various properties that can be probabilistically verified for such a system were presented in the beginning of this section. In particular, the following CSL property is used to analyse the likelihood that a service request is interrupted as time progresses:

$$\mathbf{P}_{=?}[\mathbf{F} \leq T\ abort].$$

Usually the probability to "lose" at least one request is quite high (for instance, it equals 0.99993 for 10^4 time units and rates presented in Fig. 4–5). Therefore, it is interesting to assess the probability that the number of interrupted (failed) service request will exceed some threshold or reach the predefined acceptable threshold:

$$\mathbf{P}_{=?}[\mathbf{F} \leq T\ (failed > 10)] \quad \text{and} \quad \mathbf{P}_{=?}[\mathbf{F} \leq T\ (failed = MAX\_failed)].$$

Fig. 6(a) demonstrates how these probabilities change over a period of $T = 10^4$ time units.

// successful service rates of subservices
const double $\alpha_{11} = 0.9$; **const double** $\alpha_{12} = 0.1$; **const double** $\alpha_{13} = 0.12$;
const double $\alpha_2 = 0.085$;

const double $\gamma = 0.001$; *// transient failure rate of* S_{11}

// partially successful service rates of S_{12} *and* S_{13}
const double $\beta_{12} = 0.025$; **const double** $\beta_{13} = 0.03$;

// permanent failure rates of S_{12}, S_{13}, *and* $S_{21}(S_{22})$
const double $\delta_{12} = 0.001$; **const double** $\delta_{13} = 0.002$; **const double** $\delta_2 = 0.003$;

const int $MAX = 5$; *// upperbound for retries of* S_{11}

// subservice states: $0 = nd, 1 = ok, 2 = nok, 3 = pok$
global $srv_{11} : [0..1]$ **init** 0; **global** $srv_{12} : [0..1]$ **init** 0; ... **global** $cnt : [0..100]$ **init** 0;

module S_1

$[]\ (active = 1)\&(srv_{11} = 0) \rightarrow \alpha_{11}/(cnt+1) : (srv'_{11} = 1) + \gamma : (cnt' = cnt + 1);$
$[]\ (active = 1)\&(srv_{12} = 0) \rightarrow \alpha_{12} : (srv'_{12} = 1) + \delta_{12} : (srv'_{12} = 2) + \beta_{12} : (srv'_{12} = 3);$
$[]\ (active = 1)\&(srv_{13} = 0) \rightarrow \alpha_{13} : (srv'_{13} = 1) + \delta_{13} : (srv'_{13} = 2) + \beta_{13} : (srv'_{13} = 3);$

endmodule

module S_2

$[]\ (active = 2)\&(srv_{21} = 0) \rightarrow \alpha_2 : (srv'_{21} = 1) + \delta_2 : (srv'_{21} = 2);$
$[]\ (active = 2)\&(srv_{22} = 0) \rightarrow \alpha_2 : (srv'_{22} = 1) + \delta_2 : (srv'_{22} = 2);$

endmodule

Fig. 4. Case study: modelling subservices in PRISM

const double $\lambda = 0.2$ *// service request arrival rate*
const double $\mu = 0.6$; *// service director's output rate*
const double $\eta = 1$; *// service director's handling rate*

const int $MAX\_failed = 40$; *// max acceptable threshold for the failed requests*

global $abort$: **bool init** $false$; *// 0 = nd, 1 = ok, 2 = nok, 3 = pok*
global $active : [0..3]$ **init** 1; *// 1 = IN, 2 = S1..S3, 3 = S4..S5, 4 = OUT*

module SD

$[]\ (active = 0)\&(!idle) \rightarrow \eta : (active' = 1);$
$[]\ (active = 1)\&(srv_{11} = 1)\&(cnt \leq MAX)\&(srv_{12} \neq 2)\&(srv_{13} \neq 2)\&$
$(srv_{12} = 1 \,|\, srv_{13} = 1) \rightarrow \eta : (active' = 2);$
$[]\ (active = 1)\&(srv_{12} = 2 \,|\, srv_{13} = 2 \,|\, cnt > MAX \,|\, (srv_{12} = 3 \,\&\, srv_{13} = 3)) \rightarrow$
$\eta : (active' = 3)\&(abort' = true);$
$[]\ (active = 2)\&(srv_{21} = 1 \,|\, srv_{22} = 1) \rightarrow \eta : (active' = 3);$
$[]\ (active = 2)\&(srv_{21} = 2)\&(srv_{22} = 2) \rightarrow \eta : (active' = 3)\&(abort' = true);$

endmodule

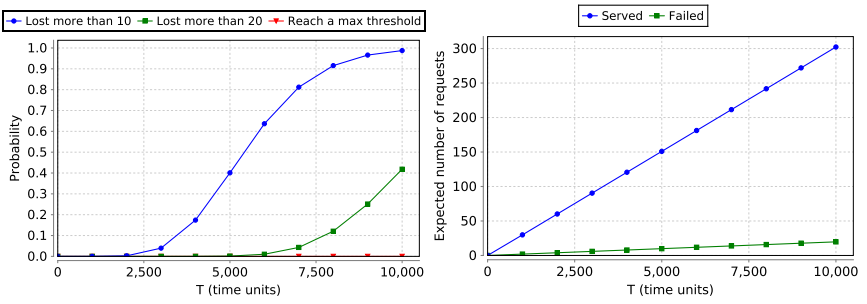
module IN_OUT

$idle$: **bool init** $true$;
$failed : [0..MAX\_failed + 1]$ **init** 0;

$[]\ (active = 0)\&(idle) \rightarrow \lambda : (idle' = false);$
$[]\ (active = 3)\&(!abort) \rightarrow \mu : (active' = 0)\&(cnt' = 0)\&(idle' = true)\&$
$(srv'_{11} = 0)\&(srv'_{12} = 0)\&(srv'_{13} = 0)\&(srv'_{21} = 0)\&(srv'_{22} = 0);$
$[]\ (active = 3)\&(abort)\&(failed \leq MAX\_failed) \rightarrow \mu : (active' = 0)\&(cnt' = 0)\&$
$(idle' = true)\&(abort' = false)\&(failed' = failed + 1)\&(srv'_{11} = 0)\&\ldots;$
$[]\ (active = 3)\&(abort)\&(failed > MAX\_failed) \rightarrow \mu : (active' = 0)\&(cnt' = 0)\&$
$(idle' = true)\&(abort' = false)\&(srv'_{11} = 0)\&\ldots;$

endmodule

Fig. 5. Case study: modelling the service director in PRISM



(a) Probabilistic properties (b) Reward-based properties

Fig. 6. Case study: results of probabilistic analysis by PRISM

The next part of the analysis is related to estimation of the failed/served requests over a period of T time units. This analysis was accomplished in PRISM using its support for reward-based properties. For each class of states corresponding to the *OUT* activity of the service director, a cost structure which assigns a cost of 1 is used. The properties

$$\mathbf{R}\{\text{`}num\_failed\text{'}\}_{=?}[\mathbf{C} \leq T] \quad \text{and} \quad \mathbf{R}\{\text{`}num\_served\text{'}\}_{=?}[\mathbf{C} \leq T]$$

are then used to compute the expected number of failed and served service requests cumulated by the system over T time units (see Fig. 6(b)).

7 Related Work and Conclusions

Modelling of SOSs is a topic of active ongoing research. Here we only overview two research strands closely related to our approach: 1) formal approaches to modelling SOSs and quantitative assessment of QoS, and 2) the approaches that facilitate explicit reasoning about the dynamic system behaviour in Event-B.

Significant research efforts have been put into developing dedicated languages for modelling SOSs and their dynamic behaviour. For instance, Orc [10] is a language specifically designed to provide a formal basis for conceptual programming of web-services, while COWS (Calculus for Orchestration of Web Services) is a process calculus for specifying and combining services [13]. Similarly to our approach, the stochastic extension of COWS relies on CTMCs and the PRISM model checker to enable quantitative assessment of QoS parameters [16]. A fundamental approach to stochastic modelling of SOSs is proposed by De Nicola et al. [6]. The authors define a structural operational semantics of MarCaSPiS – a Markovian extension of CaSPiS (Calculus of Sessions and Pipelines). The proposed semantics is based on a stochastic version of two-party (CCS-like) synchronisation, typical for service-oriented approaches, while guaranteeing associativity and commutativity of parallel composition of services.

In contrast, in our approach we rely on a formal framework that enables unified modelling of functional requirements and orchestration aspects of SOSs. We have extended our previous work on formalisation of Lyra, an UML-based approach

for development of SOSs [12,11], in two ways. First, we defined a number of formal verification requirements for service orchestration. Second, we proposed a probabilistic extension of Event-B that, in combination with the probabilistic model checker PRISM, enables stochastic assessment of QoS attributes.

There is also an extensive body of research on applying of model checking techniques for quantitative evaluation of QoS (see, e.g., [5]). We however focus on combining formal refinement techniques with quantitative assessment of QoS.

Several approaches have been recently proposed to enable explicit reasoning about the dynamic system behaviour in Event-B. Iliasov [8] has proposed to augment Event-B models with additional proof obligations derived from the provided use case scenarios and control flow diagrams. An integration of CSP and Event-B to facilitate reasoning about the dynamic system behaviour has been proposed by Schneider et al. [18]. In the latter work, CSP is used to provide an explicit control flow for an Event-B model as well as to separate the requirements dependent on the control flow information. The approach we have taken is inspired by these works. We however rely solely on Event-B to built a dynamic service architecture and verify the required service orchestration.

We can summarise our technical contribution as being two-fold. First, we have put forward an approach to defining a dynamic service architecture in Event-B. Such an Event-B model represents service orchestration explicitly, i.e., it depicts interactions of a service director with the controlled services, the order of service execution as well as fault tolerance mechanisms. Moreover, we have formally defined the conditions required for verification of a dynamic service architecture modelled in Event-B. Second, we have demonstrated how to augment such a model with stochastic information and transform it into a CTMC. It allows us to rely on probabilistic model checking techniques to quantitatively assess the desired quality of essential service attributes. In our future work, it would be interesting to extend the proposed approach to deal with dynamic reconfiguration as well as unreliable communication channels.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
3. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying Continuous Time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
4. Baier, C., Katoen, J.-P., Hermanns, H.: Approximate Symbolic Model Checking of Continuous-Time Markov Chains (Extended Abstract). In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 146–161. Springer, Heidelberg (1999)
5. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS Management and Optimization in Service-Based Systems. IEEE Trans. Softw. Eng. 37, 387–409 (2011)
6. De Nicola, R., Latella, D., Loreti, M., Massink, M.: MarCaSPiS: a Markovian Extension of a Calculus for Services. Electronic Notes in Theoretical Computer Science 229(4), 11–26 (2009)

7. Grunske, L.: Specification patterns for probabilistic quality properties. In: International Conference on Software Engineering, ICSE 2008, pp. 31–40. ACM (2008)
8. Iliasov, A.: Use Case Scenarios as Verification Conditions: Event-B/Flow Approach. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 9–23. Springer, Heidelberg (2011)
9. Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY): IST FP7 IP Project, http://www.deploy-project.eu/
10. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
11. Laibinis, L., Troubitsyna, E., Leppänen, S.: Formal Reasoning about Fault Tolerance and Parallelism in Communicating Systems. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 130–151. Springer, Heidelberg (2009)
12. Laibinis, L., Troubitsyna, E., Leppänen, S., Lilius, J., Malik, Q.A.: Formal Model-Driven Development of Communicating Systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 188–203. Springer, Heidelberg (2005)
13. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
14. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
15. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer (2005)
16. Prandi, D., Quaglia, P.: Stochastic COWS. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 245–256. Springer, Heidelberg (2007)
17. Rodin: Event-B Platform, http://www.event-b.org/
18. Schneider, S., Treharne, H., Wehrheim, H.: A CSP Approach to Control in Event-B. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 260–274. Springer, Heidelberg (2010)
19. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Quantitative Reasoning about Dependability in Event-B: Probabilistic Model Checking Approach. In: Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 459–472. IGI Global (2011)

Partially-Supervised Plants: Embedding Control Requirements in Plant Components

Jasen Markovski, Dirk A. van Beek, and Jos Baeten*

Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{j.markovski,d.a.van.beek,j.c.m.baeten}@tue.nl

Abstract. Supervisory control deals with automated synthesis of controllers based on models of the uncontrolled system and the control requirements. In this paper we share the lessons learned from synthesizing controllers for a patient support system of an MRI scanner regarding the specification of the control requirements. We learned that strictly following the philosophy of supervisory control, which partitions specifications in an uncontrolled plant and control requirements, may lead to unnecessarily complex specifications and duplication of information. In such cases, the specification can be substantially simplified by embedding part of the control requirements in so-called partially-supervised plants. To formalize the new concepts, we apply a recently developed process-theoretic approach to supervisory control. The new method for analysis of the models provides a better insight into their underlying behavior, which is demonstrated by revisiting the models of the industrial study.

1 Introduction

Modern market trends dictate lower development costs and shorter time-to-market, while increasing demands for better quality, performance, safety, and ease of use. Among else, this raises the demands on the development of control software. Traditionally, software engineers write control software based on informal specification documents, amounting to a time-consuming iterative process as the control requirements constantly change during product development. This issue gave rise to supervisory control theory of discrete-event systems [9,3], where high-level supervisory controllers are synthesized automatically based upon formal models of the hardware and control requirements.

The supervisory controller observes the discrete-event behavior of the system by receiving sensor signals from ongoing activities. Based upon these signals it makes a decision which activities are allowed to be carried out and sends back control signals to the hardware actuators. Under the assumption that the supervisory controller can react sufficiently fast on input, one can model this feedback loop as a pair of synchronizing processes. The model of the uncontrolled system, referred to as *plant*, is restricted by the model of the controller, referred to as

* Supported by Dutch NWO project ProThOS, no. 600.065.120.11N124.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 253–267, 2012.

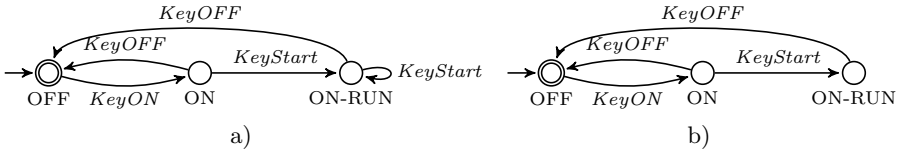


Fig. 1. Starting a car engine: a) plant component and b) control requirements

supervisor. Traditionally, the plant is modeled as a set of observable traces of events, given as a set of synchronizing components (automata), whose joint recognized language corresponds to the observed traces. The events are split into *controllable events*, which can be disabled by the supervisor in the synchronous composition (typically actuator events), and *uncontrollable events*, which must always be allowed by the supervisor (typically sensor events). The *control requirements* specify allowed behavior again as sequences of events, leading to event-based supervisory control theory [9,3].

In this paper, we revisit an industrial study regarding supervisory control of a patient support system for MRI scanners of Philips Healthcare [11]. We discuss the lessons learned from specifications of the plant and the control requirements. Namely, following the supervisory control paradigm, the plant should be modeled as unrestricted with respect to the controllable events, i.e., disabling of such events should be stated in the control requirements. However, following this paradigm may lead to duplicated specifications, slightly altered only to specify some controllable events that should be restricted. This situation usually occurs when one wants to make the specification of the (unsupervised) plant complete in the sense that all possible behavior is included, despite knowing it is irrelevant.

To provide a better intuition, we consider the process of starting a car engine. After turning the key in position ON, the engine can be started by turning the key to position start and releasing it (event *KeyStart*). Once it is started, by turning the key to position OFF, it is switched off. However, in position ON, there is no prohibition to turn the key again to "start" the already running engine. This is encountered (by accident) by almost everyone that drove in a car, observing strange noises produced by the engine. From a supervisory control point of view, the position of the key defines the behavior of the (unsupervised) plant component that models the starting of the car, depicted in Fig. 1a). The control requirements show the correct way of starting a car, depicted in Fig. 1b). Note that the requirements actually duplicate the plant component, omitting only the self loop in ON-RUN that specifies the turning of the key, while the engine is running. Moveover, every driver knows the correct way of starting a car, so we could argue that the *partially-supervised* behavior of the plant actually comprises the component in Fig. 1b). This makes for a more readable plant specification, while reducing a (superfluous) control requirement.

In the remainder, we formalize the notion of partially-supervised plants and we develop a method for analysis of the plant and control requirements that provides better insights into the underlying behavior. To this end, we employ

a recent process-theoretic approach to supervisory control that captures the standard notion of controllability by means of a behavioral preorder termed partial bisimilarity [2,10]. We retain the trace-based semantics by restricting to deterministic automata and we adapt partial bisimilarity to automata as used in supervisory control [3].

2 Supervisory Control Theory

We introduce some preliminary notions of automata and language theory as used in supervisory control theory [3]. Let $\mathcal{A} = \mathcal{C} \cup \mathcal{U}$ be the set of all events that can be observed in the plant, with $\mathcal{C}$ being the set of controllable events and $\mathcal{U}$ the set of uncontrollable events, such that $\mathcal{C} \cap \mathcal{U} = \emptyset$. We form traces and languages in a standard manner, i.e., $t \in \mathcal{A}^*$ is a trace and $L \subseteq \mathcal{A}^*$ is a language, where $\mathcal{A}^* \triangleq \{a_1 a_2 \ldots a_n \mid a_i \in \mathcal{A} \text{ for } 0 \leq i \leq n, n \in \mathbb{N}\}$ and ε denotes the unique empty trace $a_1 \ldots a_n$ for $n = 0$. By $t{\cdot}t'$ we denote the concatenation of the traces $t, t' \in \mathcal{A}^*$ and by $L{\cdot}L' \triangleq \{t{\cdot}t' \mid t \in L, t' \in L'\}$ the concatenation of languages. We omit $\cdot$ when clear from the context. We say that a language is prefix-closed if $L = \overline{L}$, where $\overline{L} \triangleq \{t \mid \text{there exists } t' \text{ such that } tt' \in L\}$.

We define a discrete-event automaton as a tuple $P = (\mathcal{S}_P, \mathcal{A}_P, \rightarrow_P, s_P, \mathcal{S}_P^m)$, where $\mathcal{S}_P$ is a set of states, $\mathcal{A}_P$ is the alphabet or the set of events used for synchronization, $\rightarrow_P \in \mathcal{S}_P \times \mathcal{A}_P \times \mathcal{S}_P$ the transition relation, s_P the initial state, and $\mathcal{S}_P^m$ is the set of marked states that denote successfully executed jobs. By $\mathcal{F}$ we denote the set of all finite automata. We define $\stackrel{*}{\rightarrow_P} \in \mathcal{S}_P \times \mathcal{A}_P^* \times \mathcal{S}_P$ as $s \stackrel{\varepsilon}{\rightarrow}{}^* s$ for all $s \in \mathcal{S}_P$, and $s \stackrel{at}{\rightarrow}{}^*_P s'$ for $a \in \mathcal{A}_P$ and $t \in \mathcal{A}_P^*$, if there exists $s'' \in \mathcal{S}_P$ such that $s \stackrel{a}{\rightarrow}_P s'' \stackrel{t}{\rightarrow}{}^*_P s'$. By $s \stackrel{t}{\rightarrow}{}^*_P$ we denote that there exists $s' \in \mathcal{S}_P$ such that $s \stackrel{t}{\rightarrow}{}^*_P s'$. Now, the recognized (prefix-closed) language of automaton P is given by $L(P) \triangleq \{t \in \mathcal{A}_P^* \mid s_P \stackrel{t}{\rightarrow}{}^*_P\}$. The recognized marked language additionally requests that the ending state is a marked state given by $L_m(P) \triangleq \{t \in \mathcal{A}_P^* \mid s_P \stackrel{t}{\rightarrow}{}^*_P s, s \in \mathcal{S}_P^m\}$. By $P_1 \mid P_2 \triangleq (\mathcal{S}_1 \times \mathcal{S}_2, \mathcal{A}_1 \cup \mathcal{S}_2, \rightarrow, (s_1, s_2), \mathcal{S}_1^m \times \mathcal{S}_2^m)$ we denote the synchronous parallel composition of $P_1 = (\mathcal{S}_1, \mathcal{A}_1, \rightarrow_1, s_1, \mathcal{S}_1^m)$ and $P_2 = (\mathcal{S}_2, \mathcal{A}_2, \rightarrow_2, s_2, \mathcal{S}_2^m)$:

$$(s', s'') \stackrel{a}{\rightarrow} \begin{cases} (\overline{s}', \overline{s}'') & \text{if } a \in \mathcal{A}_1 \cap \mathcal{A}_2, s' \stackrel{a}{\rightarrow}_1 \overline{s}', \text{ and } s'' \stackrel{a}{\rightarrow}_2 \overline{s}'' \\ (\overline{s}', s'') & \text{if } a \in \mathcal{A}_1 \setminus \mathcal{A}_2 \text{ and } s' \stackrel{a}{\rightarrow}_1 \overline{s}' \\ (s', \overline{s}'') & \text{if } a \in \mathcal{A}_2 \setminus \mathcal{A}_1 \text{ and } s'' \stackrel{a}{\rightarrow}_2 \overline{s}''. \end{cases}$$

It is easily observed that this composition is commutative and associative [3]. Note that by increasing the alphabet of an automaton with events that occur in synchronizing automata, the parallel composition would remain the same, provided that these events were added as self loops in every state.

Suppose that the plant is given by $P = (\mathcal{S}_P, \mathcal{A}, \rightarrow_P, s_P, \mathcal{S}_P^m)$ and the control requirements by $R = (\mathcal{S}_R, \mathcal{A}, \rightarrow_R, s_R, \mathcal{S}_R^m)$. If there exists $S = (\mathcal{S}_S, \mathcal{A}, \rightarrow_S, s_S, \mathcal{S}_S^m)$ such that $L(P \mid S) = L(R)$, then we say that S is a supervisor for P that achieves R. We refer to $P \mid S$ as the *supervised plant*. We ensure that S does not

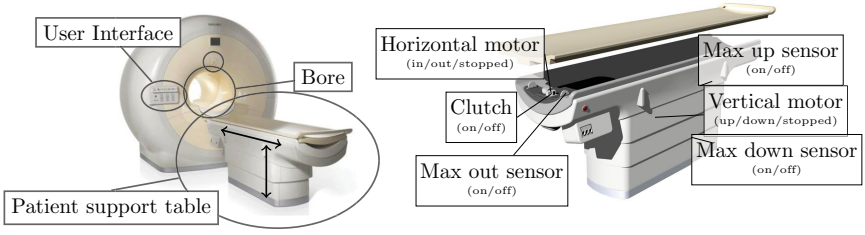


Fig. 2. Patient support system of an MRI scanner

disable uncontrollable events by requesting that R is *controllable* with respect to P, expressed by $L(R)\mathcal{U} \cap L(P) \subseteq L(R)$ [9,3]. Controllability is interpreted as follows. If we observe a desired trace in the plant followed by an uncontrollable event, then the control requirements cannot request that this uncontrollable event should be disabled after allowing that trace.

To assure, in addition, that the control is *nonblocking*, it is also required that $L(R) \subseteq \overline{L_m(P)}$. The condition ascertains that every state can reach a marked state, guaranteeing that all jobs can be successfully finished, while preventing deadlocks and livelocks. If R is controllable with respect to P and also $L(R) \subseteq \overline{L_m(P)}$, then one can guarantee the existence of a supervisor S, achieving the desired nonblocking supervised behavior R by restricting the plant P.

In general, the control requirements are not achievable and one seeks a *maximally permissive (nonblocking) supervisor*, its prefix-closed language uniquely defined for deterministic plants and control requirements as

$$M = \bigcup\{K \subseteq L(R) \cap L_m(P) \mid K \text{ is controllable with respect to } P\}.$$

In other words, the maximally permissive supervisor enables the greatest achievable nonblocking behavior that is controllable with respect to P and bounded by R. Consequently, if S is a supervisor for the plant P with respect to the control requirements R, then $L(S) \subseteq L(M) \subseteq L(R)$ and $L_m(S) \subseteq L_m(M) \subseteq L_m(R)$.

Next, we revisit the supervisor synthesis for a patient support system [11].

3 Supervisor Synthesis for a Patient Support System

The patient support system positions a patient inside an MRI scanner, see Fig. 2. The system comprises a vertical axis, a horizontal axis, and a user interface. Due to page limitations, we present only a part of the system [11]. The vertical axis consists of a lift with a motor drive and end sensors. The horizontal axis contains a removable tabletop, which can be moved in and out of the bore either by a motor drive, when the clutch is on, or by hand, otherwise. It contains sensors to detect the presence of the tabletop and its end positions. A tumble switch controls table movement and the clutch is controlled by a manual button.

The control should accomplish multiple control objectives. When the operator operates the tumble switch, the table should move up and down, or in and out

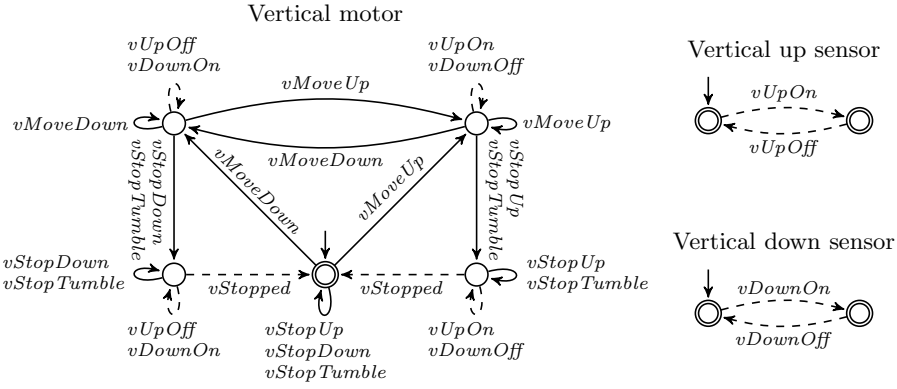


Fig. 3. Plant components for the vertical motor and end sensors

of the bore. This depends on the current position of the table and the position of the tumble switch. When the manual button is pushed, the clutch should be released such that the table can be moved manually. Finally, the table should not move beyond its end positions, and it should not collide with the magnet. Note that we do not consider faulty behavior in this paper.

This system is more difficult to control then it might appear at first sight. It contains several complex interactions of components, and the overall finite state model of the uncontrolled system contained $6.3 \cdot 10^9$ states ($6.4 \cdot 10^7$ states without user interface). Recall, that here we show just a part of this system. Nonetheless, the manufacturer estimated one week for manual adaptation of the control software to meet a change in the control requirement, while adapting them using supervisor synthesis took merely four hours [11].

We model the plant and the control requirements using automata as given in Section 2. To visualize automata, we use circles for states, full and dashed labeled arrows for controllable and uncontrollable events, respectively, incoming arrows for initial states, and doubly-lined circles for marked states. The plant and control requirements are composed out of synchronizing models for each of the components. The alphabets of the automata are comprised of the transition labels. We assume full observation of the sensors and the actuators.

Vertical Axis. The table moves up and down along the vertical axis, which comprises one actuator and two end sensors, see Fig. 3. The system should never be required to move beyond the maximally up and down position. We name the events such that their purpose becomes clear from the context. Initially, the motor is stopped and after any movement it should be able to return to its marked state. Movement is started via events *vMoveUp* and *vMoveDown*. If the motor is moving and a stop event, *vStopUp*, *vStopDown*, or *vStopTumble* is triggered, the motor slows down. When it comes to a halt, the event *vStopped* is emitted.

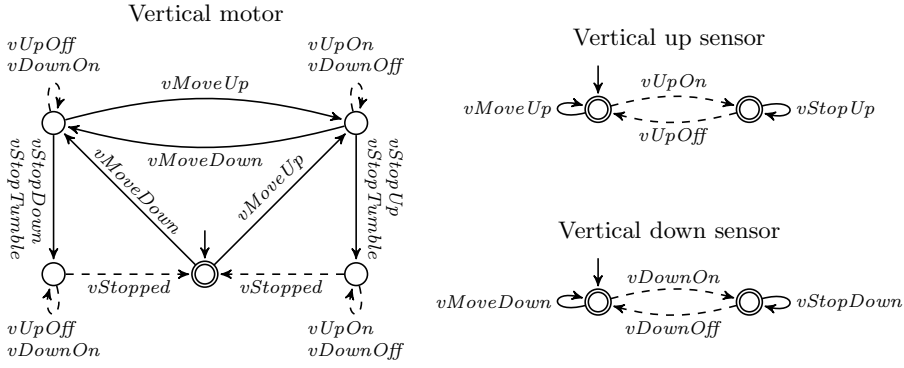


Fig. 4. Control requirements for the vertical motor and end sensors

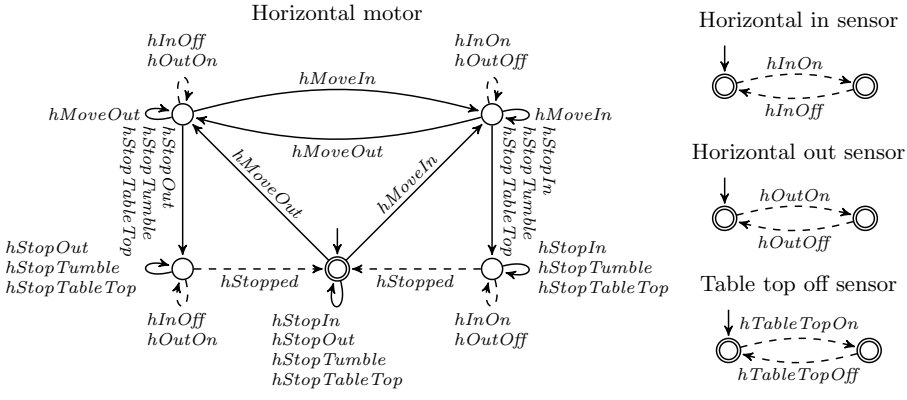


Fig. 5. Plant components for the horizontal motor, end sensors, and table top sensor

The maximally up and down sensors are active if the table is at the end sensor position, otherwise the sensors are inactive. They are modeled by means of automata with corresponding (uncontrollable) sensor events *vDownOn* and *vDownOff*, for the down sensor, and *vUpOn* and *vUpOff*, for the up sensor.

The sensors only change state when the table moves vertically. Only when the motor drive is moving the table up, the maximally down sensor can turn off, and the maximally up sensor can turn on, and vice versa. Although the end positions must be reachable, movement beyond them is not allowed. This implies that up movement is only allowed when the table is not maximally up and likewise for the down movement. Furthermore, up movement must be stopped when the table is maximally up and likewise for the down movement. In addition, once a stop event has been issued, there is no need to issue it again. These requirements lead to the models depicted in Fig. 4. Note that automata alphabets increase by adding new events, and unless self loops are added in every state, this actually restricts events in the parallel composition, cf. Section 2.

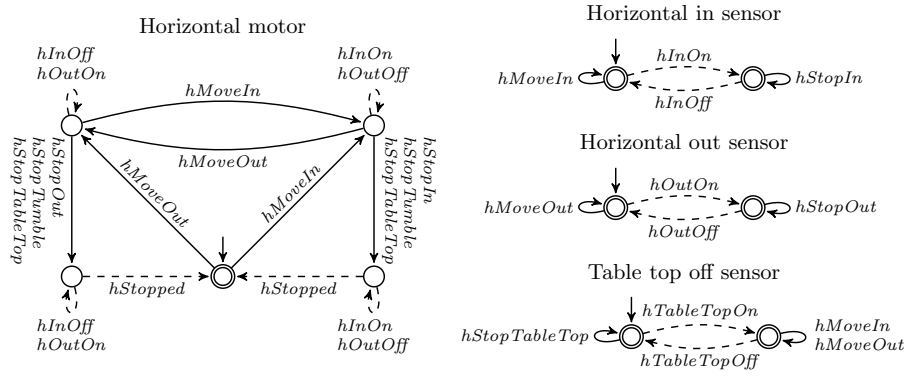


Fig. 6. Control requirements for horizontal motor, end sensors, and table top sensor

Horizontal Axis. The movement along the vertical axis is analogous to the one for the vertical axis, with the exception that the table top may be taken off by the operator, which is detected using an additional sensor. The plant components dealing with horizontal movement are depicted in Fig. 5, whereas the control requirements are depicted in Fig. 6.

User Interface. The user interface consists of a tumble switch that controls the table movement and a manual button that controls the operation mode via the clutch. In motorized mode, the tumble switch controls the movement of the table. In manual mode, the operator can move the table top by hand. When the manual button is pushed, the clutch is either applied or released, if allowed by the safety requirements of Fig. 10, leading to motorized or manual mode, respectively. The manual button push is associated to a safety timeout as manual operation is allowed only when the table top is on and it is in the topmost position, and the motors are stopped. As this takes some time, the button push might be forgotten by the operator. Fig. 7 depicts the plant components modeling the user interface.

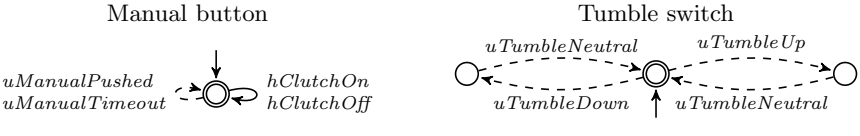


Fig. 7. Plant components for the manual button and the tumble switch

When the manual button is pressed, either the clutch is applied or released, or a timeout occurs, that invalidates the button push. When the tumble switch is down, then either downward or outward movement is allowed, whereas when the switch is up, either upward or inward movement is allowed. This is captured by the control requirements depicted in Fig. 8.

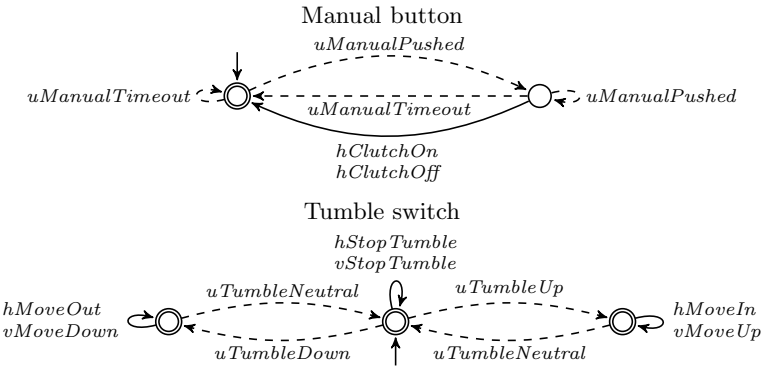


Fig. 8. Control requirements for the manual button and the tumble switch

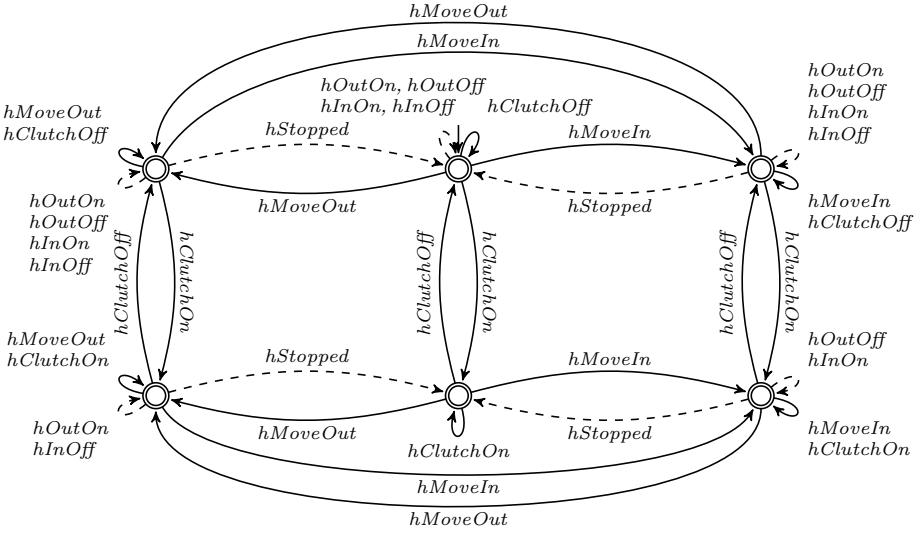


Fig. 9. Plant component relating horizontal actuator and sensor events

Finally, in Fig. 9, we capture the relationship between the horizontal motor and sensors. We note that when the clutch is off, every activation/deactivation of horizontal end sensors is possible, as the operator can move the table top unrestrictedly. We also note that the component depicted in Fig. 9 results from an interleaving (nonsynchronizing) parallel composition of a component that gives the relation between the clutch and horizontal sensor events and a component that describes motorized horizontal table movement [11].

To guarantee safe operation of the patient support system, the following safety movement restrictions apply. To ensure that the patient support table does not collide with the magnet of the MRI machine, it is required that the table can enter the magnet only if it is maximally up, and the table can move vertically

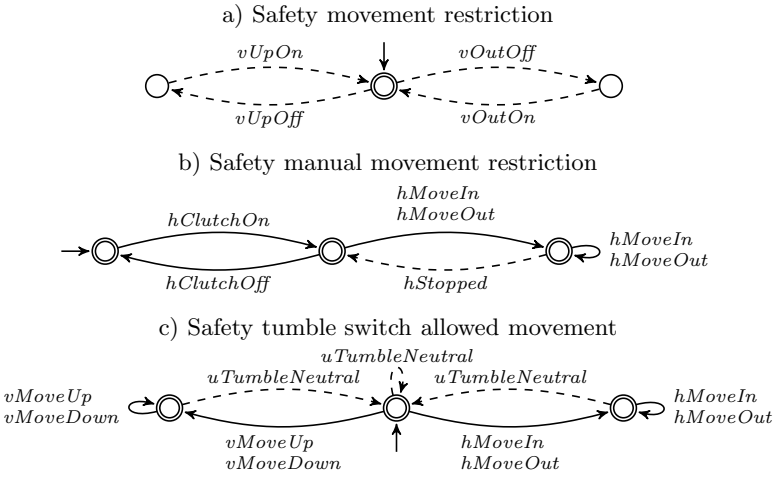


Fig. 10. Control requirements for safe patient support table movement

only if it is fully retracted (Fig. 10a). The table top can be manually operated only if the horizontal movement of the motor is stopped (Fig. 10b). In addition, to change the axis of movement, the tumble switch must be first placed in neutral position, while the motors can be activated only if the tumble switch is not in neutral position (Fig. 10c).

One cannot help to notice the duplication of information in Fig. 3 and Fig. 4, Fig. 5 and Fig. 6, and Fig. 7 and Fig. 8. Moreover, for modelers involved in this study, the plant components depicted in Fig. 3, Fig. 5, and Fig. 7, are overly simplified and actually produce the countereffect of making the plant specification unclear. This is most obvious in Fig. 9, where one can hardly deduce anything about the relationship between the horizontal actuators and sensors.

In the next chapter, we alleviate some of these issues by embedding a part of the control requirements into the plant components.

4 Process-Theoretic Approach to Controllability

We present a process-theoretic approach to supervisory control theory that enables us to manipulate more easily with the underlying notions. We define controllability from a process-theoretic perspective in terms of a so-called partial bisimilarity preorder. This preorder is meant to capture that uncontrollable events should not be disabled by the supervisor and it gives the relation between the original and the supervised plant. It requires that the unrestricted plant simulates, i.e., it is enabled to perform, every event of the supervised plant, but it is required that the supervised plant only simulates back uncontrollable events. We note that in the original process-theoretic setting of [2] we employed labeled transition systems, whereas here we adjust the behavioral (semantic) relation to accommodate deterministic automata in the vein of [3].

Definition 1. *Let* $P_1 = (\mathcal{S}_1, \mathcal{A}_1, \rightarrow_1, s_1, \mathcal{S}_1^m)$ *and* $P_2 = (\mathcal{S}_2, \mathcal{A}_2, \rightarrow_2, s_2, \mathcal{S}_2^m)$ *be two finite automata with* $\mathcal{A}_1 = \mathcal{A}_2 = \mathcal{A}' \subseteq \mathcal{A}$. *A relation* $Q \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *is a partial bisimulation between* P_1 *and* P_2 *with respect to the bisimulation action set* $B \subseteq \mathcal{A}'$ *if for all* $p_1 \in \mathcal{S}_1$ *and* $p_2 \in \mathcal{S}_2$ *such that* $(p_1, p_2) \in Q$ *it holds that:*

1. $p_1 \in \mathcal{S}_1^m$ *if and only if* $p_2 \in \mathcal{S}_2^m$;
2. *for all* $p_1' \in \mathcal{S}_1$ *and* $a \in \mathcal{A}'$ *such that* $p_1 \xrightarrow{a}_1 p_1'$, *there exists* $p_2' \in \mathcal{S}_2$ *such that* $p_2 \xrightarrow{a}_2 p_2'$ *and* $(p_1', p_2') \in Q$;
3. *for all* $p_2' \in \mathcal{S}_2$ *and* $b \in B$ *such that* $p_2 \xrightarrow{b}_2 p_2'$, *there exists* $p_1' \in \mathcal{S}_1$ *such that* $p_1 \xrightarrow{b}_1 p_1'$ *and* $(p_1', p_2') \in Q$;

We say that P_1 *is partially bisimilar to* P_2 *with respect to the bisimulation action set* B, *notation* $P_1 \preceq_B P_2$, *if there exists a partial bisimulation* Q *with respect to* B *such that* $(s_1, s_2) \in Q$. *If* $P_2 \preceq_B P_1$ *holds as well, then* P_1 *and* P_2 *are mutually partially bisimilar with respect to* B *and we write* $P_1 \leftrightarrow_B P_2$.

Note that $\preceq_B$ is a preorder relation, making $\leftrightarrow_B$ an equivalence relation for all $B \subseteq \mathcal{A}$ [10]. If $B = \emptyset$, then $\preceq_\emptyset$ coincides with strong similarity preorder and $\leftrightarrow_\emptyset$ coincides with strong similarity equivalence [5,1]. When $B = \mathcal{A}$, both $\preceq_\mathcal{A}$ and $\leftrightarrow_\mathcal{A}$ turn into strong bisimilarity [5,1].

By adopting partial bisimilarity as a behavioral (semantic) relation, we replace the original language-based equivalence, which also permits the use of nondeterministic automata. We note that partial bisimilarity also accounts for controllability of nondeterministic plants and control requirements, see [2]. Nonetheless, in the setting of this paper we only consider deterministic plants and control requirements, as they guarantee existence of a unique maximally permissive supervisor. The uniqueness is a prerequisite for the proof of main theorem that enables the embedding of the control requirements. Moreover, our experience, during execution of several other industrial studies [8,4,6,7], points out to no particular need to employ nondeterministic automata for modeling purposes.

Definition 2. *Automaton* $P = (\mathcal{S}_P, \mathcal{A}_P, \rightarrow_P, s_P, \mathcal{S}_P^m)$ *is deterministic if for all* $s, s_1, s_2 \in \mathcal{S}_P$ *and* $a \in \mathcal{A}_P$ *it holds that if* $s \xrightarrow{a} s_1$ *and* $s \xrightarrow{a} s_2$ *then* $s_1 = s_2$.

We denote the set of deterministic automata by $\mathcal{D}$. We give properties of partial bisimilarity that help to relate it to the standard notion of controllability.

Proposition 1. *Let* $P_1, P_2 \in \mathcal{D}$ *with* $P_1 = (\mathcal{S}_1, \mathcal{A}_1, \rightarrow_1, s_1, \mathcal{S}_1^m)$ *and* $P_2 = (\mathcal{S}_2, \mathcal{A}_2, \rightarrow_2, s_2, \mathcal{S}_2^m)$. *Then, the following holds:*

1. *if* $P_1 \preceq_B P_2$, *then* $P_1 \mid P \preceq_B P_2 \mid P$ *for every* $P \in \mathcal{D}$;
2. *if* $P_1 \preceq_B P_2$, *then* $P_1 \preceq_C P_2$ *for every* $C \subseteq B$;
3. $P_1 \preceq_\emptyset P_2$ *if and only if* $L(P_1) \subseteq L(P_2)$ *and* $L_m(P_1) \subseteq L_m(P_2)$;
4. *if* $\mathcal{A}_2 \subseteq \mathcal{A}_1$, *then* $P_1 \mid P_2 \preceq_\emptyset P_1$ *and* $P_2 \mid P_1 \preceq_\emptyset P_1$; *and*
5. *if* $P_1 \preceq_\mathcal{U} P_2$ *then* $L(P_1)\mathcal{U} \cap L(P_2) \subseteq L(P_1)$.

Proof. Property *1.* states that the partial bisimilarity preorder is a precongruence for the parallel composition, as shown in [2].

Property *2.* is straightforward, by following Definition 1 [2].

Property *3.* follows from Definition 1 and the definitions of recognized and marked languages, and it has been given explicitly for simulation in [5].

Property *4.* follows directly from the definition of the parallel composition and the fact that $\mathcal{A}_2 \subseteq \mathcal{A}_1$ implies that P_2 can only restrict the transitions of P_1 and, therefore, $L(P_1 \mid P_2) = L(P_2) \cap L(P_1)$ [3] implying $P_1 \mid P_2 \preceq_\emptyset P_1$ and $P_2 \mid P_1 \preceq_\emptyset P_1$ by property *2.*

Property *5.* has been previously stated in [10] in a slightly different context, but having in mind its significance, we will give another proof in this setting. Suppose that $P_1 \preceq_\mathcal{U} P_2$ holds. By Definition 1, there exists a partial bisimulation Q such that $(s_1, s_2) \in Q$. We show that $L(P_1)\mathcal{U} \cap L(P_2) \subseteq L(P_1)$ holds by contradiction. Suppose that there exists a trace $t \in L(P_1)$ such that $tu \in L(P_2)$ for some $u \in \mathcal{U}$, but $tu \notin L(P_1)$. As $t \in L(P_1)$, we have that $s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$ for $s_i \in \mathcal{S}_1$ and $t = a_1 \ldots a_n$. Then, we also have that $s_2 \equiv q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_n$ for $q_i \in \mathcal{S}_2$. Following Definition 2 we have $(s_i, q_i) \in Q$ for $1 \leq i \leq n$. However, according to Definition 1, $s_n \xrightarrow{u} s'_n$ for some $s'_n \in \mathcal{S}_1$ as $q_n \xrightarrow{u}$, leading to a contradiction. □

As in Section 2, we have that $P, R, S \in \mathcal{D}$ denote the plant, the control requirements, and the supervisor, respectively. Again, the supervised plant is given by $P \mid S$. Intuitively, controllability requires that the uncontrollable transitions of P should be bisimilar to those of $P \mid S$, so that the reachable uncontrollable parts of P and $P \mid S$ are indistinguishable. The controllable transitions of the supervised plant may only be simulated by the ones of the original plant, since some controllable transitions are suppressed by the supervisor. Then, we have that R is controllable, if $R \preceq_\mathcal{U} P$. For nonblocking behavior, we still need to ascertain that $L(R) \subseteq L_m(P)$. In case the behavior defined by the control requirements cannot be achieved, then we have that $P \mid S \preceq_\mathcal{U} P$ and $P \mid S \preceq_\emptyset R$ for some supervisor S. Note that $P \mid S \leftrightarrow_\emptyset S$, i.e., the achievable behavior is identified by the supervisor [3,9]. Also, note that for deterministic systems this is equivalent to $P \mid S \leftrightarrow_B S$ for every $B \in \mathcal{A}$ [5]. Finally, if M is the maximally permissive supervisor for P and R, then $S \preceq_\emptyset M$ for every other supervisor S of P.

Next, we employ this approach to directly manipulate plant and control requirement components, without having to unravel their recognized languages.

5 Partially-Supervised Plants

We assume that, as in Section 3, the plant and the control requirements are given as sets of parallel synchronizing components and restrictions, respectively. The following theorem states when it is possible to embed a control requirement component in the definition of the plant, without affecting the supervised behavior of the plant.

Theorem 1. *Let $P, R, X, S, T \in \mathcal{D}$ such that $P \mid X \preceq_\mathcal{U} P$, S is the maximally permissive supervisor for P with respect to $R \mid X$, and T is the maximally permissive supervisor for $P \mid X$ with respect to R. Then $S \leftrightarrow_\emptyset T \mid X$.*

Proof. As S and T are a supervisors for P and $P \mid X$, respectively, we have that

$$P \mid S \preceq_{\mathcal{U}} P \qquad \text{and} \qquad (P \mid X) \mid T \preceq_{\mathcal{U}} (P \mid X).$$

From the assumptions, we have $P \mid X \preceq_{\mathcal{U}} P$. Thus,

$$(P \mid X) \mid T \leftrightarrow_{\mathcal{U}} P \mid (T \mid X) \preceq_{\mathcal{U}} P \mid X \preceq_{\mathcal{U}} P,$$

implying that $T \mid X$ is a supervisor for P. As S is the maximally permissive supervisor for P, we have that $T \mid X \preceq_{\emptyset} S$. On the other hand, from $P \mid S \preceq_{\mathcal{U}} P$ we derive that

$$(P \mid S) \mid X \leftrightarrow_{\mathcal{U}} (P \mid X) \mid S \preceq_{\mathcal{U}} P \mid X$$

implying that S is a supervisor for $P \mid X$. As T is the maximally permissive supervisor for $P \mid X$, we have that $S \preceq_{\emptyset} T$. We show that $T \preceq_{\emptyset} T \mid X$, which implies that $S \leftrightarrow_{\emptyset} T \mid X$. Using that $(P \mid X) \mid T \leftrightarrow_{\emptyset} T$, since T is a supervisor for $P \mid X$, we derive:

$$\begin{array}{ll}
(P \mid X) \mid T \preceq_{\emptyset} P \mid X & \text{implies} \\
((P \mid X) \mid T) \mid (T \mid X) \preceq_{\emptyset} (P \mid X) \mid (T \mid X) & \text{implies} \\
(P \mid (X \mid X)) \mid (T \mid T) \preceq_{\emptyset} ((P \mid X) \mid T) \mid X & \text{implies} \\
(P \mid X) \mid T \preceq_{\emptyset} ((P \mid X) \mid T) \mid X) & \text{implies} \\
T \preceq_{\emptyset} T \mid X. &
\end{array}$$

As $S \leftrightarrow_{\emptyset} T \mid X$, we conclude that S and T deliver the same supervised behavior for P and $P \mid X$ with respect to $R \mid X$ and R, respectively. We validate that the requirements are satisfied accordingly. We have the following derivation:

$$\begin{array}{ll}
(P \mid X) \mid T \preceq_{\emptyset} R & \text{implies} \\
((P \mid X) \mid T) \mid X \preceq_{\emptyset} R \mid X & \text{implies} \\
(P \mid (X \mid X)) \mid T \preceq_{\emptyset} R \mid X & \text{implies} \\
(P \mid X) \mid T \preceq_{\emptyset} R \mid X & \text{implies} \\
P \mid (T \mid X) \preceq_{\emptyset} R \mid X, &
\end{array}$$

i.e., $T \mid X$ satisfies the control requirements for P. Also, we derive:

$$(P \mid X) \mid T \leftrightarrow_{\emptyset} (P \mid (X \mid X)) \mid T \leftrightarrow_{\emptyset} (P \mid X) \mid (T \mid X) \leftrightarrow_{\emptyset} (P \mid X) \mid S \preceq_{\emptyset} R,$$

implying that the requirements are satisfied for both supervisors.

Finally, we show that the marked behavior of $P \mid S$ and $(P \mid X) \mid T$ is equivalent, i.e., $L_m(P \mid S) = L_m((P \mid X) \mid T)$. First, note that for every $P_1, P_2 \in \mathcal{D}$, if $t \in L_m(P_1 \mid P_2)$, then $t \in L_m(P_1)$ and $t \in L_m(P_2)$. Suppose that $t \in L_m(P \mid S)$, implying that $t \in L_m(P)$ and $t \in L_m(S)$. Then, by Proposition 1, $t \in L_m(R \mid X)$ and, thus, $t \in L_m(R)$ and $t \in L_m(X)$. Now, it is not difficult to observe that $t \in L_m(P \mid T \mid X)$, as $S \leftrightarrow_{\emptyset} T \mid X$. The other direction is analogous, which completes the proof. □

Using the result of Theorem 1 we can define the notion of partially-supervised plants as specifications that embed a portion of the control requirements in them.

Such practice removes trivial and intuitive control requirements that require duplication of information and increase both the readability and meaningfulness of both the plant and control requirements.

The essential requirement of Theorem 1 is that $P \mid X \preceq_{\mathcal{U}} P$ must hold. However, such a requirement may prove difficult to check. We note that it is not necessary to take the whole plant into account, but it is sufficient to prove the claim for a portion of it. To show this, assume that $P \leftrightarrow_{\mathcal{A}} P_1 \mid P_2$ and X is such that $P_1 \mid X \preceq_{\mathcal{U}} P_1$. Then, according to Proposition 1, $(P_1 \mid X) \mid P_2 \preceq_{\mathcal{U}} P_1 \mid P_2$ holds as well, implying that $P \mid X \preceq_{\mathcal{U}} P$. Next, we characterize two cases that can be easily checked by visual inspection.

Let $E \in \mathcal{D}$ represent a totally unrestricted behavior given by $E = (\{s_E\}, \mathcal{A}, \rightarrow_E, s_E, \{s_E\})$, where $s_E \xrightarrow{a} s_E$ for every $a \in \mathcal{A}$. Then, $P \mid E \leftrightarrow_{\mathcal{A}} P$ for every $P \in \mathcal{D}$. Now, if $X \preceq_{\mathcal{U}} E$, then X is a suitable control requirement component for embedding. Visually, one needs only to verify that X does not disable any uncontrollable events in its alphabet.

A typical situation arises when the control requirements need to restrict the occurrence of controllable self loops as shown in Section 3. This requires a duplication of the plant component, while selectively adding transitions with controllable self-loop events (if not already present in the alphabet of the automaton at hand) and/or restricting their occurrences as desired, compare, e.g., Fig. 3 and Fig. 4. We show that in that case the control requirement component can be taken as a plant component.

Let $P \mid K$ be the plant, with $K = (\mathcal{S}_K, \mathcal{A}_K, \rightarrow_K, s_K, \mathcal{S}_K^m)$ a plant component. Let $L = (\mathcal{S}_K, \mathcal{A}_L, \rightarrow_L, s_K, \mathcal{S}_K^m)$ be a control requirement corresponding to this component with $\mathcal{A}_L = \mathcal{A}_K \cup C$, where $C \subseteq \mathcal{C} \setminus \mathcal{A}_K$ and $\rightarrow_L = \rightarrow_K \cup \{(s, c, s') \in \mathcal{S}_K \times C \times \mathcal{S}_K \mid s = s'\}$. Augment the transition relation of K with self loops of events in C for every state in $\mathcal{S}_K$ given by $U = \{(s, c, s) \mid s \in \mathcal{S}_K,\ c \in C\}$. Denote the augmented automaton as K'. It is straightforward that $P \mid K \leftrightarrow_{\mathcal{A}} P \mid K'$, cf. Section 2. Moreover, it is easy to see that $L \mid K' \preceq_{\mathcal{U}} K'$, so that we can replace the updated plant component K' by L, as shown above for P_1 and X.

Using the results above, we can adapt the specifications of the plant and the control requirements given in Section 3. We can replace the plant components of Figures 3, 5, and 7, with the corresponding control requirements of Figures 4, 6, and 8, respectively. The resulting replacement of the vertical and horizontal motor models of Figures 3 and 5, by the respective motor models of Figures 4 and 6, leads to a simplified plant model, which is more intuitive, and would be directly modeled by modelers with insight to the matter at hand. The substitution of the other sensor and actuator plant models of Figures 3, 5, and 7, by the corresponding control requirement models of Figures 4, 6, and 8, does eliminate duplication of information, but does not simplify the plant models themselves.

What remains are the control requirements regarding the safety of movement, depicted in Fig. 10, which can be considered as the 'meaningful' set of requirements. Here, we embed the requirement regarding the behavior of the clutch: the "Safety manual movement restriction" of Fig. 10b, in the plant component describing the actuator-sensor relationship of Fig. 9. The result of such an

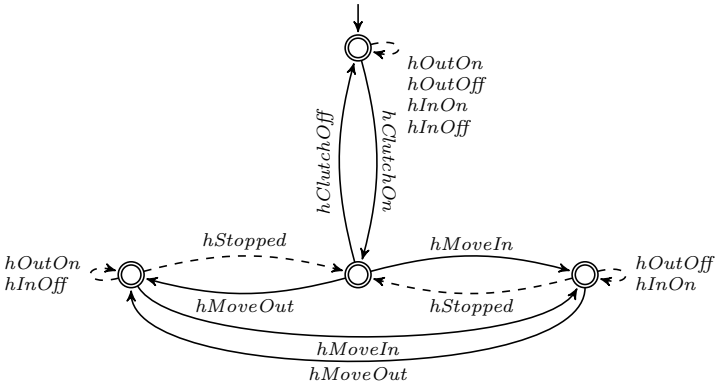


Fig. 11. Embedding the behavior of the clutch in the actuator-sensor relation

embedding is depicted in Fig. 11, which clearly shows the relation between the horizontal actuators and sensors. Namely, the horizontal motor must be stopped to allow manual operation, and in that case every sensor event is possible. In case the patient support system is in motorized mode, i.e., the clutch is applied, the sensor events can only occur consistently with the horizontal movement of the table. This behavior cannot be easily deduced from the "crowded" plant component of Fig. 9, so our analysis reveals the intended behavior of the system.

We conclude that partially-supervised plants contribute to clarity and meaningfulness of supervisory control specifications, when they are employed to eliminate trivial and cluttered control requirements. We also demonstrated that they can help better understand the underlying behavior. However, we must warn that ad-hoc modeling should not replace supervisory control, i.e., the conditions of Theorem 1 must be verified before applying the presented method. Furthermore, checking that the preconditions of the theorem hold, actually amounts to supervisory control synthesis in some cases [2]. For that purpose, we characterized two simple instances of the Theorem 1, that are often found and applied in practice without formalizing the underlying process of thought [11,8,6,4].

6 Concluding Remarks

We introduced the notion of partially-supervised plants that embed control requirements in their components. The main motivator for such an embedding is that by strictly following the principles of supervisory control, we sometimes end up with cluttered and "redundant" plant components and control requirements. Moreover, we noticed that embedding of control requirements actually occurs ad hoc during the plant modeling. We characterized when this embedding is safe and does not alter the supervised behavior. We also gave a simple characterization, which can be 'verified' visually, of the two most intuitive and most applied situations. We demonstrated the new concept in an industrial study involving supervision of a patient support system for an MRI scanner. We showed that

there is considerable improvement of the readability and the meaningfulness of the specifications of the plant and control requirements.

To show that the embedding of the control requirements does not alter the supervised behavior of the plant, we employed a process-theoretic approach that captures the notion of controllability by means of a behavioral preorder. The preorder, termed partial bisimilarity, has been adapted for deterministic automata as employed in standard supervisory control. By analyzing the proof of the main theorem, one also observes the ease of manipulation with the underlying notions, which further validates our approach to supervisory control theory.

As future work, we intend to deepen our understanding of nondeterministic partially-controlled plants, as there does not exist a unique maximally permissive supervisor. We will also investigate state-based supervisory control, where the control requirements refer to states, instead of supplying traces of events.

References

1. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge Tracts in Theoretical Computer Science, vol. 50. Cambridge University Press (2010)
2. Baeten, J.C.M., van Beek, D.A., Luttik, B., Markovski, J., Rooda, J.E.: A process-theoretic approach to supervisory control theory. In: Proceedings of ACC 2011, pp. 4496–4501. IEEE (2011)
3. Cassandras, C., Lafortune, S.: Introduction to discrete event systems. Kluwer Academic Publishers (2004)
4. Forschelen, S.T.J.: Supervisory control of theme park vehicles. Master's thesis, Systems Engineering Group, Eindhoven University of Technology (2010)
5. van Glabbeek, R.J.: The linear time–branching time spectrum I. In: Handbook of Process Algebra, pp. 3–99 (2001)
6. Leijenaar, J.F.: Supervisory Control of Document Processing Machines. Master's thesis, Systems Engineering Group, Eindhoven University of Technology (2009)
7. Markovski, J., Jacobs, K.G.M., van Beek, D.A., Somers, L.J.A.M., Rooda, J.E.: Coordination of resources using generalized state-based requirements. In: Proceedings of WODES 2010, pp. 300–305. IFAC (2010)
8. Petreczky, M., van Beek, D.A., Rooda, J.E.: Supervisor for toner error handling: a case study in supervisory control of Océ printers. SE Report 2008-011, Eindhoven University of Technology, Systems Engineering Group (2008), http://se.wtb.tue.nl/sereports
9. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25(1), 206–230 (1987)
10. Rutten, J.J.M.M.: Coalgebra, concurrency, and control. SEN Report R-9921, Center for Mathematics and Computer Science, Amsterdam, The Netherlands (1999)
11. Theunissen, R., Schiffelers, R., van Beek, D., Rooda, J.: Supervisory control synthesis for a patient support system. In: Proceedings of 10th European Control Conference, pp. 1–6. EUCA (2009)

Early Fault Detection in Industry Using Models at Various Abstraction Levels*

Jozef Hooman[1,2], Arjan J. Mooij[2], and Hans van Wezep[3]

[1] Computing Science Department, Radboud University Nijmegen, The Netherlands
[2] Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{jozef.hooman,arjan.mooij}@esi.nl
[3] Interventional X-Ray Department, Philips Healthcare, Best, The Netherlands
hans.van.wezep@philips.com

Abstract. Most formal models that are used in the industry are close to the level of code, and often ready to be used for code generation. Formal models can also be analysed and verified in order to detect any faults. As the first formal models are often such code-level models, their analysis not only reveals a lot of detailed design faults, but also the more relevant conceptual faults in the design and the requirements. Our observations are based on our experiences in an industrial development project that uses a commercial tool for formal modelling, compositional verification, and code generation. In addition to the provided tool functionality, we have introduced formal techniques to detect conceptual faults during the earlier design and requirements phases. To this end we have made additional formal models, both for the requirements and for the early designs at various abstraction levels. We have analysed these models using simulation and interactive visualization, and we have compared them using refinement checking.

1 Introduction

The formal methods that are currently the most successful in the industry are methods with commercially supported tools that provide code generation. An example is the industrial tool VDMTools [7] for the formal language VDM++ [11]. Similarly, the B-method [1], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [6]. The SCADE Suite [10] provides a formal industry-proven method for critical applications with both verification and code generation.

We report about our experiences with the industrial application of formal methods and the introduction of techniques for early fault detection [18,19]. In particular, we report about our experiences in an industrial development project at Philips Healthcare to develop control software for an interventional X-ray system. A brief description of this type of system and the developed control components can be found in Section 2.

* This publication was supported by the Allegio project, as part of the Dutch national program COMMIT, and the ITEA project Care4Me.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 268–282, 2012.

This development project uses a formal approach called Analytical Software Design (ASD) [5,21] that is supported by the commercial tool ASD:Suite [29] of the company Verum. The ASD approach uses two types of formal models: design models and interface models; both types of models are described using state machines in a tabular notation. A prerequisite for the introduction of ASD is a layered architecture with a particular communication pattern between components. Control components can be realized by an ASD design model which implements one interface model; in addition, each component can use any number of interfaces. The use of formal interface models supports concurrent software development and prevents certain integration faults.

ASD:Suite can formally verify whether each design model together with the used interface models refines the implemented interface model. The ASD approach is compositional as each component can be verified in isolation, using only the interface model of any component with which it interacts. In addition, properties like absence of deadlocks and livelocks can be checked. These formal checks can reveal subtle faults in the components, such as race conditions. An analysis of earlier applications of ASD at Philips Healthcare showed that units containing ASD components have less reported defects than other units [16].

A very important feature in industrial contexts is that ASD:Suite can use the design models to generate implementation code in a number of programming languages (C, C++, C#, Java). Code generation adds immediate value [12] to the modelling efforts that are involved in formal approaches. This also prevents the introduction of certain implementation faults related to thread creation and synchronization. Section 6 provides more details about the ASD approach.

Problem statement. In industrial development processes, the first formal models are usually close to the level of code, whereas the earlier design and requirements phases are based on informal documents. As a result, formal modelling and analysis not only reveal a lot of detailed design faults, but also the more relevant conceptual faults in the design and the requirements. These conceptual faults are often costly to repair in a detailed design phase. Moreover, as faults from several development phases are detected, it gives the impression that applying the formal approach itself is very time consuming.

Our approach. In addition to the commercial tool ASD:Suite for formal modelling, compositional verification and code generation, we have introduced formal techniques to detect conceptual faults during earlier design and requirements phases. To this end we have made additional formal models, both for the requirements and for early designs at various abstraction levels. In comparison to code-level models like ASD, such models may ignore any restrictions imposed by specific implementation technologies, they may rely on additional assumptions, and they can use simplified external interfaces.

We have analysed these models using simulation and interactive visualization, and we have compared them using refinement checking; see also the schematic overview in Fig. 8. In the next three paragraphs we describe the kinds of faults that we have thus detected in early development phases. In traditional

development approaches, these faults would probably be detected later on during the test and integration phase, where they are more expensive to repair.

Modelling and analysing requirements. Throughout the development process, we have observed that many faults are due to some unclarities in the requirements, and that such faults often lead to time-consuming redesigns. Our first priority was to increase the confidence in the requirements. In addition to the traditional documentation, we have made formal, executable models of the required system behaviour, as is also advocated by [8].

Making formal requirements models directly triggers all kinds of questions about the interpretation of the requirements. In addition, we have validated the requirements models using simulation. To support discussions with domain experts and non-technical stakeholders, we have connected the simulations to an interactive visualization of a physical view on the system. This is described in Section 3.

Modelling and analysing designs. The ASD approach imposes some restrictions on the design models, e.g., to ensure that they can be verified compositionally. To validate high-level design decisions, we have made several formal, executable models of early designs that do not (yet) adhere to these restrictions. We have validated these models using simulation. In early development phases, we have thus revealed various design faults, often related to feature interactions, that lead to deadlocks, livelocks, and functional errors.

During the development phases, we have iteratively analysed increasingly detailed design models. The use of interactive visualizations (as discussed for the requirements models) also proved useful for the design models. To prepare the final application of ASD, we have defined a pattern for simulating the ASD components in detailed design models. More details are given in Section 4.

Comparing requirements and designs. When making several formal models, it often happens that small discrepancies are introduced. To some extent, these can be discovered using the light-weight simulations discussed before. For a more profound comparison, we have built a compiler that transforms a requirements model and a design model to the input language of a refinement checker.

The use of refinement checkers to compare models has revealed subtle design faults that are difficult to find by simulation. However, exhaustive verification easily hits the state space explosion problem, and scalability to industrial sizes is still challenging. In Section 5 we explain how we have addressed this.

2 Control Components for an Interventional X-Ray System

The experiences described in this paper are based on our work in a development project at Philips Healthcare. This project concentrates on the development of control components for interventional X-ray systems as depicted in Fig. 1. Such

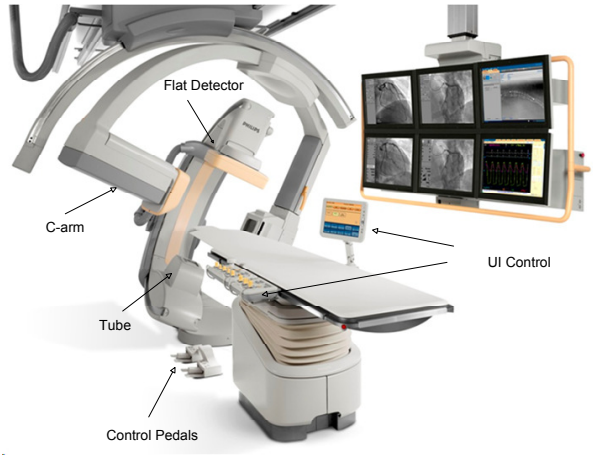


Fig. 1. Interventional X-Ray system

systems are used for minimally-invasive cardiac, vascular and neurological procedures, such as placing a stent via a catheter. During such a medical procedure, the surgeon is guided by real-time images showing, for instance, the position of the catheter inside the patient. These images are constructed from the amount of X-ray that is detected after sending X-ray beams (generated by the tube) through the patient.

The system under development consists of two X-ray planes (called lateral and frontal) that can be used in isolation or together (called biplane). Each plane can apply three types of X-ray that vary in the amount of X-ray that is emitted:

- Fluoroscopy: low dose, for interactive viewing and positioning;
- Exposure SingleShot: high dose, for recording a single image;
- Exposure Series: high dose, for recording a series of images.

The clinical users can control this system using six pedals and one hand-switch. Moreover, each of these inputs can be replicated multiple times. Three of the pedals are used to start Fluoroscopy, corresponding to the planes. For Exposure, there is one pedal to switch between the planes, and there are two pedals to start the two types of Exposure. In addition, Exposure Series can also be started using the hand-switch.

Apart from the user inputs, there can be several reasons for interrupting the X-ray beams once started, or for preventing the X-ray beams to start in the first place; such conditions are called run-conditions and start-conditions respectively. Examples include technical problems with the hardware, but also conditions related to physical safety such as an open door.

Given the earlier experiences [16,20] of Philips Healthcare with ASD, it was decided to develop the main control components using ASD. Moreover, their

external interfaces were already specified using ASD. The use of ASD is motivated by the aim to shorten the test and integration phase, which is usually long to ensure a high level of quality. Starting from the application of the ASD approach, we have experimented with the introduction of formal techniques to describe the requirements and the global design. The aim was to find faults as early as possible to improve the efficiency of the development process.

3 Modelling and Analysing Requirements

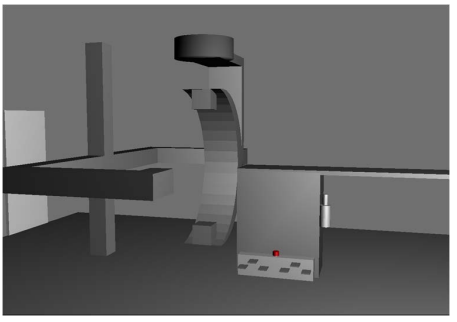
Since it is very costly to correct requirements faults during detailed design, we propose the use of formal techniques to detect such faults as early as possible. To obtain industrial acceptance and fast feedback, we have made executable models that can be simulated. In addition, the models have been simulated in combination with an interactive visualisation of the externally visible behaviour.

Executable models. To model the requirements, we have used the Parallel Object-Oriented Specification Language (POOSL) [9,28]. POOSL is a very expressive formal language with timing, predefined data types, statistical distributions and synchronous communication along channels, similar to CSP [25]. The semantics of POOSL is defined as a timed probabilistic labelled transition system. Models can be simulated by means of the tools supporting POOSL.

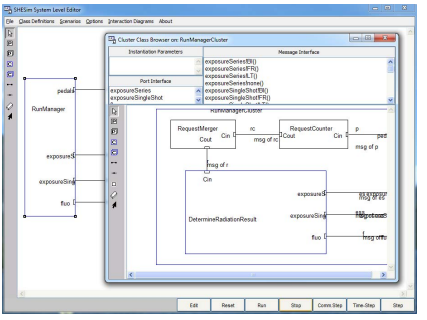
Interactive visualisation. To discuss the requirements with domain experts, we have investigated the use of interactive visualizations. The visualizations provide an attractive and understandable graphical user interface, but the logic follows from a simulation of the formal, executable requirements model.

In earlier work [23] we have used interactive 2D animations based on Flash. More recently we have started to explore the use of interactive 3D animations based on Blender [2]; see Fig. 2(a). Blender is an open source tool that combines a 3D modelling environment with an interactive game engine. Our first impression is that some situations (pedal states, active X-ray beams, etc.) are more easily understood using a realistic graphical view. In comparison to the professional animations that are used in the industry for explaining their products, our visualizations are interactive and the graphical aspects are separated from the internal logic. In this way, our interactive visualizations can also be used later in the development process to evaluate technical models of the architecture and detailed design.

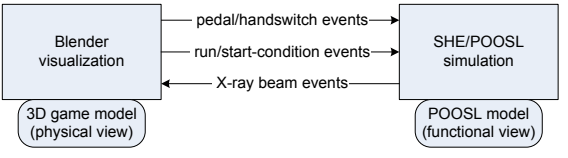
By connecting a Blender model (see Fig. 2(a)) and a POOSL model (see Fig. 2(b)) via sockets, we obtain a simulation that combines two views on the system: the physical hardware and the control logic, respectively. As shown in Fig. 2(c), inputs to the system are forwarded by Blender to POOSL, whereas resulting actions are transferred from POOSL to Blender. This combination can also be used to analyse the effectiveness of the full clinical workflows for executing the use cases. We can even imagine that such techniques can be used for training purposes, e.g., when the real system is not (yet) available.



(a) 3D visualization for simulation



(b) Modelling and simulation



(c) Coupling models

Fig. 2. Modelling and analysing requirements using Blender and POOSL

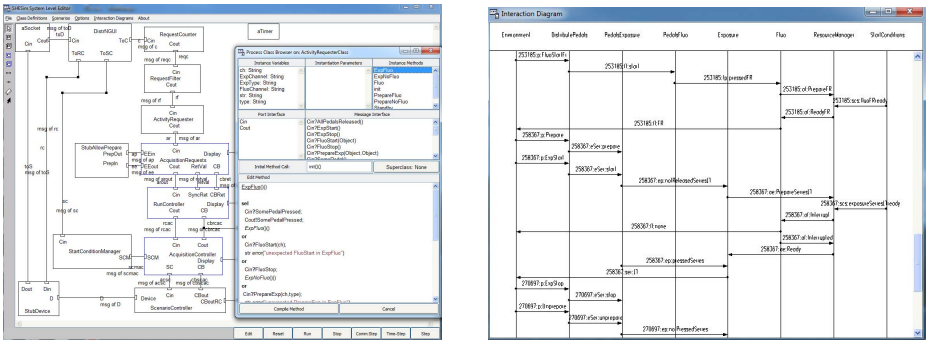
Results. We have started with modelling the types of X-ray and their relations. At any moment in time, only one type of X-ray may be active; in particular, the Exposure types have priority over Fluoroscopy. This basic behaviour was well documented, but the interpretation of the informal text was not always easy. For instance, regarding the intended result of simultaneous X-ray requests through different pedals and hand-switches. In these cases, our light-weight simulations quickly clarified the intended interpretation of the descriptions.

In a later phase, we considered error handling by adding start-conditions and run-conditions to the models. For instance, as Exposure Series can only be started after an additional preparation phase, it needs to be decided when the start-conditions should be checked, and whether any ongoing Fluoroscopy needs to be stopped at the beginning of the preparation phase. The formal, executable models turned out to be useful to make some implicit domain knowledge explicit. In this way time-consuming re-designs can be avoided.

4 Modelling and Analysing Designs

To enable a successful application of ASD, the design should meet a number of constraints. During our project, it turned out to be difficult to devise a design that meets these ASD constraints. The main limiting factors are that ASD:Suite concentrates on single components only and that it does not support simulation.

To get some insight in the essential structure and interaction between the components, we have first made several abstract design models and simulated them using POOSL. The graphical part of a POOSL model shows the structure



(a) Modelling and simulation (b) MSC generated by simulation

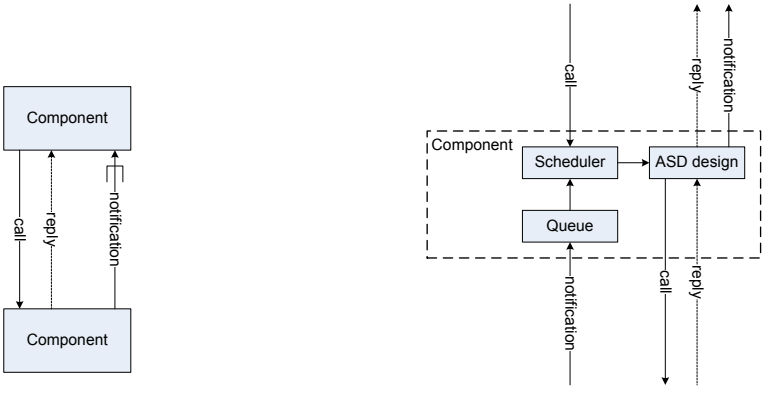
Fig. 3. Modelling and analysing designs using POOSL

of components and their interaction, as depicted in Fig. 3(a). Components can be clustered into hierarchies. At the lowest level, a component is described in a textual CSP-like language. During the simulation, the POOSL tool shows the internal state of each component, and also the interactions between the components in terms of a Message Sequence Chart (MSC); see Fig. 3(b). These features enable a developer to perform a detailed analysis of whether and how the design supports the typical use cases and their interactions.

Abstract designs. The first design model that we made focused on the component structure and the basic component interaction, without considering message parameters and error handling. After a number of iterations, a first satisfactory abstract design was obtained. An important aspect of this design is the decoupling between the fast interactions with the users of the system and the more time-consuming control of the X-ray devices in the system. This design was informally validated using simulation and inspection of the generated MSCs by industrial domain experts.

Afterwards, this design has been refined gradually with message parameters and error handling. Some restructuring was needed to keep the components small and to achieve the required behaviour. For instance, one of the early design models revealed that after a specific scenario the X-ray beam was erroneously not switched off, although all pedals were released. Clearly such important faults would also be detected immediately during the test and integration phase, but the benefit of formally analysing the design models is that the fault is detected earlier, and hence it is faster and cheaper to repair.

ASD-based designs. The application of ASD requires a layered architecture where components can interact as described in Fig. 4(a). The components in higher layers may perform function calls (resulting in a reply) on components at lower layers, but not the other way around. Components in lower layers may



(a) Component interactions (b) Modeling ASD components in POOSL

Fig. 4. ASD interaction patterns

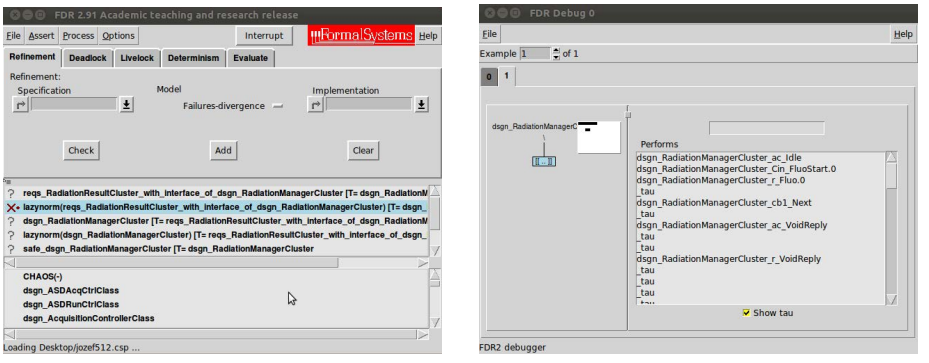
send asynchronous notifications to components in higher layers. There should be no direct interaction between components in the same layer. In this way, ASD can achieve absence of deadlocks by construction.

The POOSL models that contain ASD design models need to take this into account. Notifications have to be buffered in one queue per component. The ASD semantics also prescribes specific scheduling rules for the order in which the calls and queued notifications are processed. This semantics can be captured in POOSL by a cluster consisting of three components, as shown in Fig. 4(b).

5 Comparing Requirements and Designs

To check the consistency between the requirements and design models, we have used the formal refinement checker FDR2 [13,14]; see the screenshot in Fig. 5(a) together with a debug trace for a detected fault in Fig. 5(b). To this end, we have built a compiler that translates two models (using a subset of the POOSL language) to a CSP model, which is the input of FDR2; see also Fig. 8. An alternative refinement checker would be the Process Analysis Toolkit (PAT) [27], which is also based on a CSP dialect. In comparison to simulations, refinement checkers can automatically verify a lot of subtle scenarios.

The FDR2 tool supports, amongst others, trace refinement and failures-divergence refinement [14]. As explained in Section 6, Verum's tool ASD:Suite successfully manages to apply FDR2 to the verification of components. However, directly comparing full design and requirements models using FDR2 easily becomes infeasible. In the following, we explain how we were able to use FDR2 to find subtle discrepancies between the requirements and design models. We conclude with some typical results that we have obtained.



(a) Main window of FDR2 (b) Debug trace generated by FDR2

Fig. 5. Comparing requirements and design models using FDR2

Making the models finite state. Tools like FDR2 apply explicit state space exploration, and hence the state space of the models must be finite. The models that we consider consist of several components. In general, even if the components have an unbounded state space, it is possible that in their composition only a finite part is reachable. However, to analyse any model using FDR2, not only the state space of the entire model must be finite, but also the state space of each of the components must be finite.

The notification queues that are used in ASD designs are unbounded. Verum provides various suggestions for users of ASD:Suite to deal with this for single components, but these are not generally applicable to full design models. We have manually introduced an upperbound on the size of each queue. If the bound is violated, a special event is generated. The upperbound is valid in the design if a trace refinement indicates that this special event cannot occur.

For some external events, such as run-conditions, the bound on the queue size depends on timing aspects (the speed with which events are generated and consumed). To do partial verification in these cases, we have restricted the number of these external events that can be generated.

Making the refinement checks feasible. The requirement that the state space must be finite is just a minimum requirement. As FDR2 applies explicit state space exploration, the size of the state space requires constant attention when applying FDR2 to full models. In our project, generating the state spaces of the models is not the biggest issue. After generating the state spaces, however, FDR2 applies a normalization step before the real refinement checking begins. For complex specifications, it is known [25] that this normalization step can take a long time, and that the normalized version can be much larger than the original. To be able to find at least some traces that distinguish the requirements and design models, we have used FDR2's function "lazynorm" [26,24,25] that does not attempt to normalise the specification completely before carrying out the refinement check.

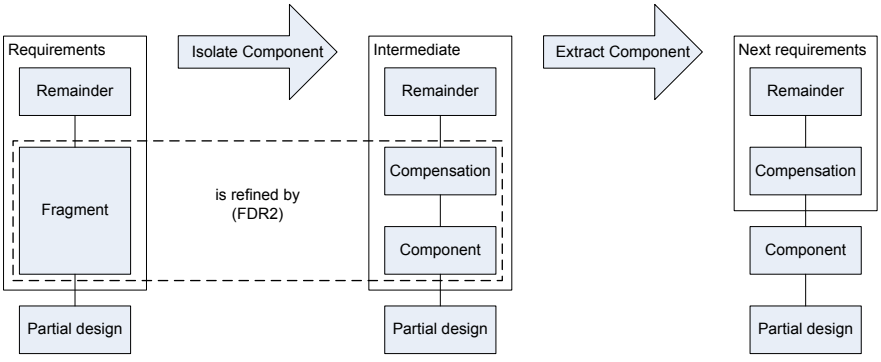


Fig. 6. Decomposition by extracting components

Most of the faults that we have detected using checks for failures-divergence refinement were actually at the level of trace refinement. Failures-divergence refinement implies trace refinement, and the latter seems easier to check. When feasibility is an issue, we have switched to checking for just trace refinement.

Decomposing models by extracting components. Another way to make refinement checking feasible in practice is to decompose a single refinement check on large models into several separate refinement checks on smaller models. In particular, we have considered a decomposition related to a typical design process that iteratively identifies design components based on the requirements. The idea is to gradually transform a requirements model into a design model; the intermediate stages combine some design components with the remaining requirements.

Consider the upper part of Fig. 6, i.e., ignore the blocks labelled "Partial design". The starting point at the left is a requirements model. After having defined a suitable design component, this component is related to a fragment of the requirements model. The aim is to show that this requirements fragment can be replaced by the composition of two blocks, viz., the component and some compensation. That is, the component and the compensation together should be a refinement of the original fragment. We have constructed the compensation manually, but ideally this would be automated using techniques like submodule construction [17,4] or equation solving [22]. If we can construct a compensation, then we have managed to isolate the component.

To finish this step, we extract the component from the requirements. Thus we obtain a model consisting of a part of the requirements (the remainder of the requirements and the compensation) and a model consisting of a part of the design (the design component). Afterwards we apply this approach again on the requirements part, as indicated by the blocks labelled "Partial design" in the full version of Fig. 6. In this way, the design model grows gradually and the formal refinements deal with a part of the original requirements only.

The validity of this decomposition approach depends on the associativity and congruence properties of composition. We have applied this approach mainly to

some conceptually simple components in the design that have a large state space, for example, because of internal counters. By extracting these components, the state space of the remaining models is reduced drastically.

Restricting the interfaces. The design models have to deal with the real technical interfaces, whereas the requirements models may make some simplifying assumptions. To compare requirements and design models, we have added some components that perform the conversions between these interfaces. In particular, we have extended the design model with components that translate the more detailed design interfaces to the simpler requirements interfaces, thus hiding some of the technical details.

In some cases, we have also restricted the possible external events. Although this means that we are not performing a full verification, again it has led to the detection of faults that were not found otherwise (see also the guiding techniques proposed in [15]). This can also be used for comparing a design model with a requirements model that is only correct under some assumptions.

Results. This kind of analysis reveals very subtle faults earlier in the development process than using traditional development approaches. For instance, we have detected some discrepancies between our models with respect to the requirement that in certain situations X-ray requests are cancelled when a pedal is released quickly after it has been pressed. In some cases, the problem turned out to be an inaccuracy in our requirements model.

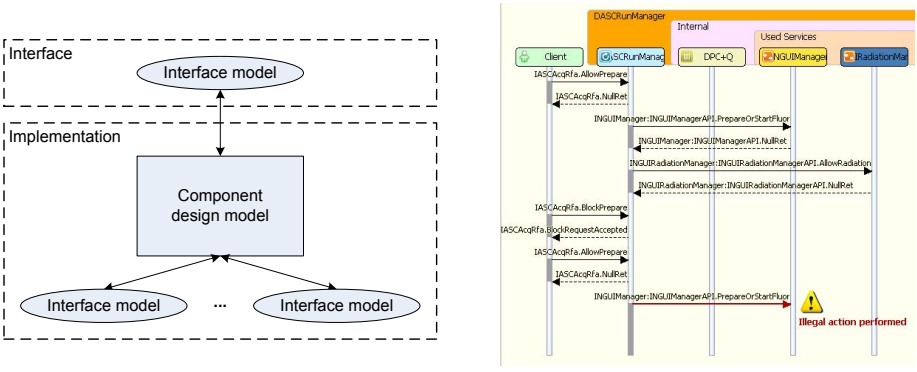
Note that our design model is not failures-divergence equivalent to the requirements model. For instance, when biplane Fluoroscopy cannot be started because of a start-condition, instead only one of the planes without a start-condition (if any) should be started. In the requirements model this is specified as a non-deterministic choice between the lateral or the frontal plane, whereas the design model implements this choice deterministically, which is fine.

6 Detailed Design with ASD

Once we had obtained confidence in the design, the individual components were realized using the ASD approach. We briefly describe the four ASD phases: interface modelling, design modelling, formal verification, and code generation. A small example that illustrates ASD can be found, for example, in [20].

Interface modelling. First, the internal and external interfaces of all components have been modelled using ASD. Such an interface model is a state machine that defines which calls and notifications are allowed and in which order. For instance, it may state that a *StartXRay* call is always followed by a *Started* or *StartFailed* notification; afterwards, a next *StartXRay* call may only occur after a *StopXRay* call. Thus, an interface model can be seen as a contract about the interaction protocol between components.

The tool ASD:Suite includes a number of basic consistency checks on the interface models. Most important is that the interface model must be complete in



(a) Verification condition (b) MSC generated by verification

Fig. 7. Verification of an implementation with respect to an interface

the sense that the response to each call or notification must be defined explicitly in all states. If a call or notification should not occur in a state, then it can be declared to be illegal.

Design modelling. After the definition of the interface models, which was a joint team effort, the components were developed concurrently. Components with data manipulations were implemented manually and tested to check compliance with their interfaces. The control components that do not involve any data manipulations were implemented using ASD design models. An ASD design model is a state machine, but, in contrast to an interface model, it must be deterministic. It defines how the component responds to calls and notifications, e.g., by performing calls and notifications to other components.

Formal verification. ASD:Suite can verify each design model for properties like absence of deadlocks and livelocks. In addition, it can verify whether each component is correct with respect to its interfaces, which means that

- no illegal calls are performed on the used interfaces; and
- the design model composed with all used interface models is a failures-divergence refinement of the implemented interface model (these are called implementation and interface respectively in Fig. 7(a)).

The motivation for these verifications is that the component interfaces are a frequent source of faults during integration. Such a verification can already eliminate such faults during the design. However, it does not guarantee the functional correctness of the components nor the entire design.

These verifications are implemented in ASD:Suite by internally transforming the models into CSP and then invoking the refinement checker FDR2; see also Fig. 8. To make the verification feasible, it is important to keep (the state space of) the components small and to limit the number of notifications.

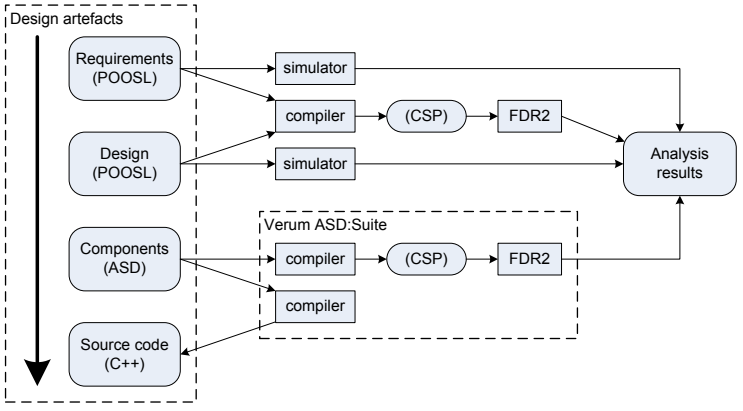


Fig. 8. Overview of models and transformations

This kind of verification typically reveals all kinds of race conditions due to the non-deterministic arrival order of events. For instance, many detected faults were related to the run-conditions, which can interrupt the normal execution flow at almost any point in time. These faults manifest themselves as refinement errors or as calls or notifications that are illegal according to the interface models. Each fault is reported as a Message Sequence Chart (MSC) covering the component and its interfaces; see Fig. 7(b).

Code generation. Finally, when the design models have been verified, ASD:Suite can generate code from the design models. Currently, ASD:Suite offers a choice between a number of programming languages (C, C++, C#, or Java).

7 Conclusions and Further Work

The first formal models that are made during development are usually close to the code level. The danger is that the formal analysis of such models reveals all kinds of faults from all previous development phases. In our industrial project, we have observed that this complicates the application of formal methods and causes delays in the project planning. To detect faults earlier, i.e., in the appropriate development phase, we have introduced additional formal models at various levels of abstraction and earlier in the development process.

In our industrial project, the formal ASD approach has been used, supported by the commercial tool ASD:Suite of the company Verum. We have added a number of techniques on top of this approach, as summarized in Fig. 8:

- modelling: to define the functionality precisely;
- simulation: to explore the functionality described in the models;
- visualization: to provide a user interface based on a physical system view;
- refinement checking: to rigorously compare pairs of models.

The main benefit of applying such techniques is early fault detection. By not only applying them to design models, but also to requirements models, developers can create a better understanding of the requirements in earlier stages of the development process.

It is well-known that unclear requirements lead to costly redesigns. This becomes very visible in the context of formal approaches such as ASD, because redesigns also require that the interface models have to be adapted, and that a large number of formal verification conditions has to be satisfied again. To use ASD more effectively, the conclusion is, also at Philips Healthcare, that more attention should be paid to the definition of the requirements.

Making formal models of the requirements, and analysing these models using simulation and interactive visualization has proved to be very useful to remove a number of unclarities. On the other hand, we have also observed that not all faults can be found practically in this way. Using refinement checking, we have detected additional faults in scenarios where a number of fast user commands are received while the system is still busy processing the first command. It is almost impossible to explore such subtle scenarios using interactive simulation.

Further work. There are various directions for further work. First of all, we are planning to evaluate the use of multiple formal models during the development of another, larger industrial system. In particular, we anticipate the need for additional kinds of analysis, such as a performance analysis for response times. It is also interesting to explore to what extent a good requirements model can formally guide the development of the design model.

On the tooling side, a compiler from (a restricted class of) POOSL design models to ASD components would be a nice extension. An all-in-one tool covering all these techniques might be interesting, but automated model transformations are our first priority. For the applicability of refinement checking, it is very relevant to investigate whether techniques such as submodule construction and equation solving, as mentioned in Section 5, can be applied at an industrial scale. It is also interesting to explore whether model-based testing [3] can be applied to the design models as an alternative for the refinement checking.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York (1996)
2. Blender, http://www.blender.org/
3. Boberg, J.: Early fault detection with model-based testing. In: Proceedings of Erlang Workshop 2008, pp. 9–20. ACM (2008)
4. von Bochmann, G.: Using First-Order Logic to Reason about Submodule Construction. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 213–218. Springer, Heidelberg (2009)
5. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: Some experiences of formal methods in practice. In: Proceedings of APSEC 2003, pp. 49–58 (2003)
6. ClearSy: Atelier B, http://www.atelierb.eu/en/
7. CSK Systems Corporation: VDMTools, http://www.vdmtools.jp/en/

8. Easterbrook, S.M., Lutz, R.R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Transactions on Software Engineering 24(1), 4–14 (1998)
9. Eindhoven University of Technology: Software/Hardware Engineering (SHE) - Parallel Object-Oriented Specification Language (POOSL), http://www.es.ele.tue.nl/poosl/
10. Esterel Technologies: SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York (2005)
12. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: The Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 237–254. Springer, Heidelberg (2007)
13. Formal Systems (Europe) Ltd: FDR2, http://www.fsel.com/
14. Formal Systems (Europe) Ltd and Oxford University Computing Laboratory: Failures-Divergence Refinement – FDR2 User Manual, 9th edn. (2010)
15. Goga, N., Romijn, J.: Guiding Spin Simulation. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 176–193. Springer, Heidelberg (2004)
16. Groote, J.F., Osaiweran, A., Wesselius, J.H.: Analyzing the effects of formal methods on the development of industrial control software. In: Proceedings of ICSM 2011, pp. 467–472. IEEE (2011)
17. Haghverdi, E., Ural, H.: Submodule construction from concurrent system specifications. Information & Software Technology 41(8), 499–506 (1999)
18. Holzmann, G.J.: Early fault detection tools. Software - Concepts and Tools 17(2), 63–69 (1996)
19. Holzmann, G.J.: Formal Methods for Early Fault Detection. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 40–54. Springer, Heidelberg (1996)
20. Hooman, J., Huis in 't Veld, R., Schuts, M.: Experiences with a Compositional Model Checker in the Healthcare Domain. In: George, C. (ed.) FHIES 2011. LNCS, vol. 7151, pp. 93–110. Springer, Heidelberg (2012)
21. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. ENTCS 128(6), 127–144 (2005)
22. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: Proceedings of LICS 1990, pp. 108–117. IEEE Computer Society (1990)
23. Li, L., Hooman, J., Voeten, J.: Connecting technical and non-technical views of system architectures. In: Proceedings of CPSCom 2010, pp. 592–599 (December 2010)
24. Roscoe, A.W., Armstrong, P.J., Pragyesh: Local Search in Model Checking. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 22–38. Springer, Heidelberg (2009)
25. Roscoe, B.: Understanding Concurrent Systems. Springer (2010)
26. Ryan, P.Y.A., Schneider, S.A., Goldsmith, M.H., Lowe, G., Roscoe, A.W.: The Modelling and Analysis of Security Protocols: the CSP Approach. Pearson Education (2000)
27. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: Proceedings of ISoLA 2008. CCIS, vol. 17, pp. 307–322. Springer (2008)
28. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P.H.A., Voeten, J.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of MEMOCODE 2007, pp. 139–148. IEEE (2007)
29. Verum Software Technologies: ASD:Suite, http://www.verum.com/

PE-KeY: A Partial Evaluator for Java Programs*

Ran Ji and Richard Bubel

Technische Universität Darmstadt, Germany
{ran,bubel}@cs.tu-darmstadt.de

Abstract. We present a prototypical implementation of a partial evaluator for Java programs based on the verification system KeY. We argue that using a program verifier as technological basis provides potential benefits leading to a higher degree of specialization. We discuss in particular how loop invariants and preconditions can be exploited to specialize programs. In addition, we provide the first results which we achieved with the presented tool.

1 Introduction

In this paper we present a prototypical implementation of a partial evaluator for Java based on the verification system KeY [1] called PE-KeY. The theoretical framework for this approach has been presented in [2].

The KeY verification system formalizes the Java programming language as proof system using a Gentzen-style sequent calculus capturing the sequential Java semantics faithfully. Although declarative, the sequent calculus used in KeY features a strong operational flavor. The calculus rules capturing the semantics of the Java programming language, are designed following closely the symbolic execution paradigm. They realize basically a symbolic interpreter, which differs from a concrete interpreter by using symbolic input values instead of concrete values. For instance, consider the Java program statement `x = y + z;` where the program variables `y` and `z` have the symbolic values y_0 and z_0 respectively. Symbolically executing the statement leads to a (symbolic) state update for program variable `x` whose new symbolic value becomes the expression (and *not* the value of) $y_0 + z_0$. In general, the symbolic execution of control flow statements like an if-statement will branch as the condition might be true or false depending on the *instantiation* of the symbolic values. Subsequently, the symbolic interpreter follows both branches in separation.

Once the Java program has been executed symbolically, the verifier ends up with a set of symbolic states. These symbolic states represent in particular all concrete states the original Java program may encounter in an actual program run. In a verification setting one has now to prove that the property of interest is valid in all these possible final states.

* This work has been supported by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 HATS project *Highly Adaptable and Trustworthy Software using Formal Models*.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 283–295, 2012.

The idea of using symbolic execution as a technology for program verification goes back to [3]. While the verification of a program is performed in an analytical manner, i.e., the program is eliminated until only first-order proof obligations remain to be proven, one can also read the proof in a program construction fashion. The tool presented here interweaves both views. While the original source program is analyzed (verified) in a first phase, the specialized program is constructed in a second phase by traversing the obtained proof tree in the opposite direction.

The potential benefit of this approach is the following: The verifier maintains a faithful representation of the symbolic state at all intermediate program states. The degree of precision of the symbolic state is crucial for possible simplifications of program expressions as well as for other specialization techniques like dead-code elimination. We argue that using a program verifier and associated techniques allows the achievement of a high degree of precision. For instance, in case of loops we are able to use unwinding and/or loop invariants. The possibility to provide (strong) loop invariants allows to maintain a highly precise description of the symbolic state even if a loop cannot be completely unwound. A similar argument holds for method calls where method contracts can be used in addition to method inlining.

Another benefit, is that (modular) program verification does not assume a single entry point and can be applied to each method in isolation. In general this limits the effect of program specialization as the parameters cannot be assumed to have static and known values. However, the state and the allowed values of the method parameters are usually restricted by method preconditions and system invariants. In a program verification environment we use these restrictions to simplify expressions and to cut off infeasible program paths. Hence, the specialized programs do not include code for these infeasible code paths. In addition our approach is not restricted to static input values but can also achieve specialization (and/or certain kinds of optimizations) using first-order constraints on the input values.

Further, the program verification calculus can itself be extended by rules performing basic partial evaluation steps like constant propagation. This extension and a logic characterization of the partial evaluation rules has been presented in [4]. Using this technique improves the obtainable degree of specialization considerably.

The paper is structured as follows: In Section 2 we describe the theoretical background of our approach. Section 3 describes the implementation and the results achieved so far. Section 4 describes related work. Finally, Section 5 concludes with an outlook of ongoing and future work.

2 Calculus

2.1 Dynamic Logic

We consider only sequential Java programs (without garbage collection) and can thus make the assumption that all programs are deterministic. In addition,

when using the notion "program" we usually mean an executable sequence of statements. If we want to refer to the context in which these statements are executed, we make this explicit by using the notion *context program*. The *context program* encompasses all interface and class declarations.

Java Dynamic Logic (JavaDL) is basically a standard first-order logic plus two modalities $\langle\cdot\rangle\cdot$ (diamond) and $[\cdot]\cdot$ (box). In this paper we use only the box modality. Given an executable sequence of Java statements `p` and an arbitrary JavaDL formula ϕ then the formula

- $\langle$`p`$\rangle\phi$ means intuitively that program `p` terminates and in its final state ϕ holds (total correctness).
- [`p`]ϕ means that if program `p` terminates then ϕ holds in its final state (partial correctness).

As mentioned in the introduction, symbolic states play a central role for our approach. The representation of symbolic states, and in particular of state changes are crucial and often a bottleneck for symbolic interpreters. JavaDL models locations (program variables, attributes, etc.) as flexible constants (or unary functions on objects), i.e., constants and functions whose value can be changed by a program. Relations between locations and restrictions on their values can be expressed using standard first-order (dynamic logic) formulas. To keep track and to represent state changes efficiently, JavaDL has one additional important feature named *updates*.

An elementary update u is an update of the form $loc := val$ where loc is a program variable and val is a term representing the value assigned to loc. Updates can be parallelized $u_1 \| \ldots \| u_n$ meaning that all locations are assigned their new values simultaneously. An update u can be applied to terms t and formulas ϕ resulting again in a term $\{u\}t$ or a formula $\{u\}\phi$. Some examples:

- {`i` := `j`}ϕ: formula ϕ is evaluated in a state where the program variable `i` is assigned `j`.
- {`i` := `j`‖`j` := `i`}ϕ where ϕ is evaluated in a state where the values of `i` and `j` have been swapped.

Update simplification is performed eagerly to achieve a compact representation of the symbolic state.

2.2 Sequent Calculus

A formula in JavaDL is proven correct in KeY using a Gentzen-style sequent calculus. A sequent is a data structure of the following shape $\Gamma \Longrightarrow \Delta$, where Γ and Δ are sets of formulas. The meaning of a sequent is the same as the meaning of the implication $\bigwedge_{\gamma\in\Gamma}\gamma \rightarrow \bigvee_{\delta\in\Delta}\delta$. The sequent calculus performs syntactic transformations of the sequents by applying rules of the form

$$\mathsf{name}\ \frac{seq_1 | \ldots | seq_n}{seq}$$

where $seq, seq_i, i \in \{1, \ldots n\}, n \geq 0$ are sequents. The sequent seq is called conclusion of the rule, while the sequents $seq_1, \ldots, seq_n$ are called premises. An example of such a rule is the andRight rule:

$$\text{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}$$

The formula $\phi \wedge \psi$ is called main formula of the sequent, ϕ and ψ are schema variables that stand for any possible JavaDL formula, Γ, Δ are schema variables that match on sets of formulas.

Application of a rule happens in an analytic manner, i.e., the rule is applied from bottom to top. A rule can be applied on a sequent s if there is a match for its conclusion. In that case the premises are instantiated accordingly and become the children of sequent s.

A sequent proof is a tree with a root r where each node $n \neq r$ is the result of a rule application on its parent node. A branch of the tree is closed if a closure rule like close/closeTrue/closeFalse

$$\frac{*}{\Gamma, \phi \Longrightarrow \phi, \Delta} \qquad \frac{*}{\Gamma \Longrightarrow true, \Delta} \qquad \frac{*}{\Gamma, false \Longrightarrow \Delta}$$

has been applied, marking the sequent as valid. A proof is called closed if all its branches are closed.

For this paper we are most interested in the rules working on program formulas. The calculus is designed to work like a symbolic interpreter. The rules match and work on the first (active) statement. One class of rules performs equivalent program transformations which stepwise decompose a complex statement into a series of atomic statements. While another class of rules translates atomic statements into a first-order logic representation.

The rule postInc

$$\frac{\Gamma \Longrightarrow \{u\}[\texttt{int t = i; i = i + 1; j = t; r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{j = i++; r}]\phi, \Delta}$$

is a representative of a pure program transformation rule transforming a complex statement in two simpler statements[1]. The assignment rule

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \{u\}\{\texttt{j} := \texttt{i}\}[\texttt{r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{j = i; r}]\phi, \Delta}$$

belongs to the second class of rules. It performs the single side-effects of an atomic assignment by moving the assignment into an update. Another representative of this class is the rule conditional

$$\frac{\Gamma, \{u\}b \Longrightarrow \{u\}[\texttt{p;r}]\phi, \Delta \quad \Gamma, \{u\}\neg b \Longrightarrow \{u\}[\texttt{q;r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{if (b) \{ p \} else \{ q \} r}]\phi, \Delta}$$

[1] The temporary variable `t` is necessary because the schema variables `i` and `j` may match on the same local variable.

which causes the proof to split. The first branch assumes that `b` holds and we have to show that ϕ holds if the `then` branch is executed or that the assumption `b` is contradictory in the current state. Analogous for the `else` branch.

2.3 Compilation Rules

While the sequent calculus is applied analytically when used for verification, it is also possible to interpret a proof in a program construction manner. We use the second reading as motivation for our approach to implement a partial evaluator on top of a proof attempt.

The main idea is to extend the box modality

$$[\,\mathtt{p} \sim \mathtt{sp_p}\,]@(obs, use)$$

to carry additional information. The extension consists of a second compartment for the specialized version $sp_{\mathtt{p}}$ of the source program `p` and two additional annotations *obs* and *use* containing sets of locations (program variables, fields, etc.). The location set *obs* keeps track of observable locations by an "outside" entity, while the location set *use* contains those locations that are read from in the continuation of the program. Without going into detail, these sets of locations are used to detect unused variable assignments and to eliminate them as soon as possible. We call these modalities, *bisimulation* modalities as `p` and $\mathtt{sp_p}$ have to be in a bisimulation relation with respect to the postcondition and the set of observable variables.

The general sequent calculus rules for the bisimulation modality are of the following form:

$$\text{ruleName}\quad \frac{\begin{array}{c}\Gamma_1 \Longrightarrow \{u_1\}[\,\mathtt{p_1} \sim \mathtt{sp_1}\,]@(obs_1, use_1)\phi_1, \Delta_1 \\ \cdots \\ \Gamma_n \Longrightarrow \{u_n\}[\,\mathtt{p_n} \sim \mathtt{sp_n}\,]@(obs_n, use_n)\phi_n, \Delta_n\end{array}}{\Gamma \Longrightarrow \{u\}[\,\mathtt{p} \sim \mathtt{sp}\,]@(obs, use)\phi, \Delta}$$

The application of sequent calculus rules for the bisimulation modality consists of two phases.

1. Symbolic execution of source program `p`. It is performed bottom-up as usual in sequent calculus rules. In addition, the observable location sets obs_i are also propagated since they contain the locations observable by $\mathtt{p_i}$ and ϕ_i that will be used in the second phase to synthesize the specialized program. Normally *obs* could contain the return variables of a method and the locations used in the continuation of the program.
2. We synthesize the target program $\mathtt{sp_i}$ and use_i by applying the rules in a top-down manner.

To obtain a specialization of a method `m(args)` we start the proof with a sequent of the form

$$\Longrightarrow pre \rightarrow [\,\mathtt{r{=}m(args)} \sim \mathtt{sp}\,]@(\{\mathtt{r}\}, use)\mathtt{POST}$$

where *pre* is the precondition of method `m` and `POST` is an unspecified predicate which can neither be proven nor disproved. This allows the easy identification of closed proof branches with infeasible paths and thus sound elimination of dead-code. The variables *sp* and *use* are placeholders for the specialized program and the used variable set which is computed in the second phase.

In the second phase, when the program is fully symbolically executed, the specialized program is synthesized by "applying" the rules in the opposite direction. This phase starts effectively with nodes where the emptyBox rule has been applied:

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \{\sqcap\}@(obs, use)\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\,\texttt{nop} \sim \texttt{nop}\,]@(obs, obs)\phi, \Delta}$$

with `nop` denoting the empty program. The rule initializes the variable set *use* to the set of observable variables *obs*. The idea is that *use* keeps track of all variables whose value has (potentially) been read.

We show only some of the bisimulation rules introduced in [2]. We use $\overline{p}$ to denote the specialized version of p and omit for space reasons the context variables Γ, Δ. The rule

$$\text{assignLocalVariable}\ \frac{\Longrightarrow \{u\}\{\texttt{l} := \texttt{r}\}[\,\omega \sim \overline{\omega}\,]@(obs, use)\phi}{\begin{array}{l}\Longrightarrow \{u\}[\,\texttt{l} = \texttt{r}; \omega \sim \texttt{l} = \texttt{r}; \overline{\omega}\,]@(obs, use - \{l\} \cup \{r\})\phi \\ \qquad\qquad \text{if } l \in use \ \&\ \texttt{r} := \texttt{r}' \notin u \\ \Longrightarrow \{u\}[\,\texttt{l} = \texttt{r}; \omega \sim \texttt{l} = \texttt{r}'; \overline{\omega}\,]@(obs, use - \{l\} \cup \{r'\})\phi \\ \qquad\qquad \text{if } l \in use \ \&\ \texttt{r} := \texttt{r}' \in u \\ \Longrightarrow \{u\}[\,\texttt{l} = \texttt{r}; \omega \sim \overline{\omega}\,]@(obs, use)\phi \qquad\qquad otherwise\end{array}}$$

describes the specialization of an assignment statement where one local variable is assigned to another one. The rule has three conclusions of which only one is taken. They differ only in the synthesized program compartments, i.e., in the analyzing (first) phase no ambiguity arises.

If the left side l of the assignment is a location with a (potential) read access before its next re-definition, i.e., $l \in use$, an assignment statement is generated. The *use* set is updated by removing the now re-defined program variable l and adding the program variable r which is read by the assignment. This explains the first of the three conclusions. But we can do better: in case the preceding update contains an elementary update with r as left-hand side and r' as right-hand side, we inline the actual value directly and generate as assignment $l = r'$. The *use* set is updated accordingly[2].

We give a brief example motivating the existence of the second conclusion. Assume we encounter the following sequent:

$$\Longrightarrow \{\ldots \| \texttt{y} := \texttt{z} + \texttt{1}\}\{\texttt{x} := \texttt{y}\}[\,\omega \sim \overline{\omega}\,]@(obs, \{x, \ldots\})\phi$$

[2] if r' is an expression all variables occurring in r' have to be added to *use*.

Since x is in the *use* set, an assignment statement has to be generated. Notice that $\mathtt{y} := \mathtt{z} + \mathtt{1}$ occurs in the update u, therefore the assignment is synthesized as `x = z + 1` according to the second case of the assignment rule. The variable `x` is removed and the variable `z` is added to the *use* set. We get as result:

$$\Longrightarrow \{\ldots \| \mathtt{y} := \mathtt{z} + \mathtt{1}\}[\mathtt{x} = \mathtt{y}; \omega \sim \mathtt{x} = \mathtt{z} + \mathtt{1}; \overline{\omega}\,]@(obs, \{z, \ldots\})\phi$$

The third conclusion of the assignment rule is used if the value of l has not been accessed by the remaining program ω and does not generate an assignment at all.

Synthesizing loops is achieved using one (or a combination) of two approaches: (i) loop unwinding to execute a fixed number of loop iterations and (ii) using the loop invariant rule for loops with no fixed bound:

whileInv

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}inv, \Delta \\ \Gamma, \{u\}\{\mathcal{V}_{mod}\}(\mathtt{b} = \mathtt{TRUE} \wedge inv) \Longrightarrow \{u\}\{\mathcal{V}_{mod}\} \\ \qquad\qquad [\,\mathtt{p} \sim \overline{\mathtt{p}}\,]@(obs \cup use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \{u\}\{\mathcal{V}_{mod}\}(\mathtt{b} = \mathtt{FALSE} \wedge inv) \Longrightarrow \{u\}\{\mathcal{V}_{mod}\}[\,\omega \sim \overline{\omega}\,]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\,\mathtt{while(b)\{p\}}\,\omega \sim \mathtt{while(b)\{\overline{p}\}}\,\overline{\omega}\,]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

to achieve program specialization in finite time.

On the logical side the loop invariant rule is as expected and has three premises. Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use *true* as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onwards (if *true* is used as invariant then it holds trivially).

Two things are of importance: the third premise executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, we must add to the variable set *obs* of the second premise the used variables of the third premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the *use case* premise of the loop invariant rule and then using the resulting use_1 set in the second premise.

3 Implementation and Experiments

The implementation of PE-KeY is a non-trivial extension based on KeY, which includes the following efforts:

- An information collector along with the symbolic execution of the source Java program. It keeps track of the observable variables and constructs the working stack that is used in the synthesize phase.
- An integrated partial evaluator which performs some simple partial evaluation operations such as constant propagation and dead code elimination. It is used in the symbolic execution phase.
- The compilation rules that are used to generate specialized program in the second phase. KeY's sequent calculus has around 1200 rules of which around 100-150 rules are used for symbolic execution of programs. Around half of them has been implemented in the current version of PE-KeY, but a considerable effort is required to get a complete coverage.
- An update analyzer used to extract symbolic values of program variables from preceding updates to achieve a higher degree of specialization.

The current version of PE-KeY supports basic Java features such as assignment, comparison, conditional, loop, method call inlining, integer arithmetics. Array data structure and field access are also supported to some extent. Multithreading and floating point arithmetics are not supported due to limitations of KeY.

PE-KeY is available at `www.key-project.org/ifm12` and runnable via Java Web Start (no installation needed). We have tried PE-KeY with a set of example programs that are available from the website given above. Although in an early stage, the examples indicate the potential of PE-KeY once full Java is supported. For instance, the (simplified) formula

$$\texttt{i>j} \rightarrow [\texttt{if(i>j) max = i; else max = j;}]\texttt{POST}$$

leads to the following specialization of the conditional statement:

```
max = i;
```

because of the precondition `i>j` and thanks to the integrated first-order reasoning mechanism in PE-KeY. For the same reason,

$$\texttt{i = 5} \rightarrow [\texttt{i++;}]\texttt{POST}$$

results in the specialized statement `i = 6`.

In fact, the program can be specialized according to the given specification from a general implementation. Fig. 1 shows a fragment of a bank account implementation. A bank account includes the current available balance and the credit line (normally fixed) that can be used when the balance is negative. Cash withdraw can be done by calling the `withdraw` method. If the withdraw amount does not exceed the available balance, the customer will get the cash without any extra service fee; if the available balance is less than the amount to be withdrawn, the customer will use the credit line to cover the difference with 5 extra cost; if the withdrawn amount could not be covered by both the available balance and the credit line, the withdraw does not succeed. In every case, the information of the new available balance will be printed (returned). This is a general

implementation of the cash withdrawal process, but some banks (or ATMs) only allow cash withdraw when the balance is above 0. In this case, the precondition of the `withdraw` method is restricted to `withdrawAmt <= availableBal`. Then with help of PE-KeY, the implementation of method `withdraw` is specialized to:

```
return availableBal - withdrawAmt;
```

```
public class BankAccount {
  int availableBal;
  int creditLn;

  BankAccount( int availableBal, int creditLn ) {
    this.availableBal = availableBal;
    this.creditLn = creditLn;
  }

  public int withdraw(int withdrawAmt) {
    if (withdrawAmt <= availableBal) {
      availableBal = availableBal - withdrawAmt;
      return availableBal;
    } else {
      if(withdrawAmt - availableBal <= creditLn) {
        availableBal = availableBal - withdrawAmt - 5;
        return availableBal;
      } else {
        return availableBal;
      }
    }
  }
  ...
}
```

Fig. 1. Code fragment of bank account

We applied our prototype partial evaluator also on some examples stemming from the JSpec test suite [5]. One of them is concerned with the computation of the power of an arithmetic expression, as shown in Fig. 2.

The interesting part is that the arithmetic expression is represented as an abstract syntax tree (AST) like structure. The abstract class `Binary` is the superclass of the two concrete binary operators `Add` and `Mult` (the strategies). The `Power` class can be used to apply a `Binary` operator `op` and a `neutral` value for `y` times to a `base` value `x`, as illustrated by the following expression.

```
power = new Power(y, new op(), neutral).raise(x)
```

The actual computation for concrete values is performed on the AST representation. To be more precise, the task was to specialize the program

```
class Power extends Object{
 int exp;
 Binary op;
 int neutral;

 Power(int exp, Binary op,
       int neutral) {
   super();
   this.exp = exp;
   this.op = op;
   this.neutral = neutral;
 }

 int raise(int base) {
   int res = neutral;
   for (int i=0; i<exp; i++) {
     res = op.eval( base, res );
   }
   return res;
 }
}
```

```
class Binary extends Object {
  Binary() { super(); }
  int eval(int x, int y) {
    return this.eval(x, y);
  }
}

class Add extends Binary {
 Add() { super(); }
 int eval(int x, int y) {
    return x+y;
 }
}

class Mult extends Binary {
 Mult() { super(); }
 int eval(int x, int y) {
   return x*y;
 }
}
```

Fig. 2. Source code of the Power example as found in the JSpec suite [5]

```
power = new Power(y, new Mult(), 1).raise(x);
```

under the assumption that the value of y is constant and equal to 16.

As input formula for PE-KeY we get:

$$y \doteq 16 \rightarrow [\texttt{power=new Power(y,new Mult(),1).raise(x);} \sim \texttt{sp}_{\texttt{res}}]@(obs, use)\texttt{POST}$$

PE-KeY then executes the program symbolically and extracts the specialized program $\texttt{sp}_{\texttt{res}}$ as `power = ((x*x)*x)*...)*x;` (see Fig. 3). The achieved result is a simple `int`-typed expression without the intermediate creation of the abstract syntax tree and should provide a significant better performance than executing the original program.

Our current implementation is still in its infancies and there are other examples of the JSpec test suite. But the already achieved results indicate that the presented approach can be scaled up to real world examples. More examples and results can be found at `www.key-project.org/ifm12`.

4 Related Work

JSpec [5] is the state-of-the-art program specializer for Java. In fact, JSpec does not support full Java but a subset without concurrency, exception, reflection.

```
==>
   y = 16
 -> \compileDiamond{
                      power=new Power (y,new Mult (),1).raise(x);
                    }\endmodality POST

Node Nr 1
Compilation Result:variableDeclaration
Used Vars:{x,heap}
Declared Vars:{x,heap,power}
Program:

power=((((((((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x;
```

Fig. 3. Specialized program computed by PE-KeY as result of the JSpec example

Our tool has similar limitations due to the restriction of KeY itself. JSpec uses an *offline* partial evaluation technique that depends on *binding time analysis*, which in general is possibly not as precise as *online* partial evaluation.

Civet [6] is a recent partial evaluator for Java based on *hybrid* partial evaluation, which performs offline-style specialization using an online approach without static binding time analysis. The programmer needs to explicitly identify which parts of the programs partial evaluation should be applied. In our approach, which is based on the program verification, the specification (or precondition) is mainly used to prove the correctness of the programs, and it also contributes to the program specialization. Our approach is similar to Civet in the sense that the specification is also user provided, but no extra annotations specially for the partial evaluation purpose is needed.

Our approach is based on symbolic execution to derive information on-the-fly, similar to *online* partial evaluation [7], the main difference being that we do not generate the specialized program during the symbolic execution phase, but synthesize it in the second phase. In principle, our first phase can obtain as much information as online partial evaluation, and the second phase can generate a more precise specialized program.

A big advantage of our approach is that the specialized program is guaranteed to be correct with respect to the original program. It is related to the work of proving the correctness of a partial evaluator by Hatcliff et al. [8,9]. The difference is that they need to encode the correctness properties into a logic programming language to perform the proof; while our approach ensures the correctness naturally by the deductive compilation rules and thus no further proof is needed.

Verifying Compiler [10] project aims at the development of a compiler that verifies the program during compilation. In contrast to this, our approach might be called the *Compiling Verifier*, since the specialized Java program is generated after verifying the source program. The correctness of the generated program is ensured by the compilation rules. Related to our work, compiler verification [11]

aims to guarantee the correctness of the target program. However, compiler verification attempts to verify the compiling program which is very expensive and hardly scales to realistic target languages and sophisticated optimizations.

Our work is closely related to rule-based compilation [12,13], but to the best of our knowledge their inference machine is by far not as powerful as the reasoning engine of KeY. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [14]) which also use a theorem prover to discharge proof obligations. They work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates the actual strategy from the translation rules. What distinguishes our work from most approaches that we know is that the starting point is a system for functional verification of Java which is used for program specialization in such a way that it becomes fully automatic.

5 Conclusions and Future Work

In this work, we presented PE-KeY, a partial evaluator for Java programs based on KeY. It works in two phases. In the first phase, symbolic execution interleaved with simple partial evaluation is performed, which is similar to online partial evaluation process. In the second phase, the specialized Java program is synthesized. A use-definition chain set is maintained to eliminate unused assignments and to avoid unnecessary statements occurring in the specialized program. The advantage of PE-KeY, among other Java partial evaluators, is that the correctness of the specialization is naturally guaranteed by the bisimulation relationship of the source and specialized programs, together with the soundness of the program logic. This has been proved in our previous work [2].

As an extension of KeY, PE-KeY does not support multi-threading and floating point in Java due to the restriction of KeY itself. However, these features are being investigated in KeY, once they are supported, PE-KeY could easily extend these features as well.

There are still some optimization opportunities for the current version of PE-KeY. For instance, on encountering a loop, the heuristics that decide whether to unwind it or not have a strong influence on the resulting specialized programs. Importing information, e.g., loop invariants or ranking functions, from other tools could also be useful. And for the method call, the method body is always inlined for the moment, however, to use the method contract instead might sometimes generate a better specialized program and is worthy to investigate in the future.

The idea of this paper is to generate specialized Java programs, however, the bisimulation modality is not restricted to source and target program being from the same language, but it can be generalized to other languages provided with corresponding observable locations. Consequentially, the approach is still sound for generating bytecode or other intermediate languages. An important future work will be to generate Java bytecode from Java source by using our approach. It will be a deductive Java compiler which guarantees a correct compilation

without further verification of the bytecode or JVM as introduced in [15]. We also plan to apply our approach to the modeling language ABS developed in the context of the HATS project [16,17].

References

1. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334, pp. 410–451. Springer, Heidelberg (2007)
2. Bubel, R., Hähnle, R., Ji, R.: Program Specialization via a Software Verification Tool. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 80–101. Springer, Heidelberg (2011)
3. King, J.C.: A program verifier. PhD thesis, CMU (1969)
4. Bubel, R., Hähnle, R., Ji, R.: Interleaving Symbolic Execution and Partial Evaluation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 125–146. Springer, Heidelberg (2010)
5. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM-TPLS 25(4), 452–499 (2003)
6. Shali, A., Cook, W.R.: Hybrid partial evaluation. In: OOPSLA, pp. 375–390 (2011)
7. Ruf, E.S.: Topics in online partial evaluation. PhD thesis, Stanford University, Stanford, CA, USA, UMI Order No. GAX93-26550 (1993)
8. Hatcliff, J.: Mechanically Verifying the Correctness of an Offline Partial Evaluator. In: Swierstra, S.D. (ed.) PLILP 1995. LNCS, vol. 982, pp. 279–298. Springer, Heidelberg (1995)
9. Hatcliff, J., Danvy, O.: A computational formalization for partial evaluation. Mathematical Structures in Computer Science 7(5), 507–541 (1997)
10. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50, 63–69 (2003)
11. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Softw. Eng. Notes 28, 2 (2003)
12. Augustsson, L.: A compiler for lazy ML. In: Proc. of the ACM Symposium LFP 1984, pp. 218–227. ACM, New York (1984)
13. Breebaart, L.: Rule-based compilation of data parallel programs. PhD thesis, Delft University of Technology (2003)
14. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A Translation Validator for Optimizing Compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
15. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer (2001)
16. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
17. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability Modelling in the ABS Language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 204–224. Springer, Heidelberg (2011)

Specification-Driven Unit Test Generation for Java Generic Classes

Francisco R. de Andrade[2], João P. Faria[2,3], Antónia Lopes[1], and Ana C.R. Paiva[2]

[1] Faculdade de Ciências da Universidade de Lisboa
[2] Faculdade de Engenharia, Universidade do Porto
[3] INESC Porto
{francisco.andrade,jpf,apaiva}@fe.up.pt, mal@di.fc.ul.pt

Abstract. Several approaches exist to automatically derive test cases that check the conformance of the implementation of abstract data types (ADTs) with respect to their specification. However, they lack support for the testing of implementations of ADTs defined by generic classes. In this paper, we present a novel technique to automatically derive, from specifications, unit test cases for Java generic classes that, in addition to the usual testing data, encompass implementations for the type parameters. The proposed technique relies on the use of Alloy Analyzer to find model instances for each test goal. JUnit test cases and Java implementations of the parameters are extracted from these model instances.

1 Introduction

Algebraic specifications have been successfully used for the formal specification of abstract data types (ADTs) and several approaches exist to automatically derive test cases that check the conformance of the implementation of ADTs with respect to their algebraic specifications (e.g., [3,5,8,11,17]). In these approaches, because ADTs are described in an axiomatic way, the derivation of tests involves choosing some instantiations of the axioms or their consequences. Then, concrete tests are generated to check if these properties hold in the context of the implementation under test (IUT).

Many data types admit different versions in different applications—e.g., *sets of strings, dates, messages*. Nowadays, the implementation of these data types in mainstream object-oriented languages, such as Java and C#, strongly relies on generic classes. However, existing methods and techniques to automatically generate test suites from specifications cannot be directly applicable in these cases.

Genericity poses new difficulties for testing. To write tests for a generic class one has to commit to a set of types for its parameters and this raises several problems. First, in the case of non trivial parameters, types for instantiating them may not be available at test time (e.g., trees of intervals as defined by edu.stanford.nlp.util.IntervalTree have intervals as parameters and no implementation for this type is available there [16]). Second, the types available for instantiating the parameters may not cover all the possibilities allowed by the parameters. For instance, partially ordered sets that have a type parameter that corresponds to partial orders can be tested with strings or integers, however, the properties of these sets that hold vacuously in total orders will not be properly tested. Third, in order to isolate the source of possible failures, one may not want to

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 296–311, 2012.

depend on the implementation of other types besides the one under test (this is a unit testing best practice). A technique that is often used to overcome these difficulties in manual test generation is the use of mock objects [14]. One of the challenges in automatic test generation for generic classes is the automatic generation of mock objects for their parameters, removing from the user the burden of providing the types for instantiating the type parameters.

In this paper we address the generation of unit test cases for Java implementations of ADTs defined by generic classes, that comprise automatically generated mock classes and mock objects that can be used to instantiate their type parameters. As illustrated in Fig. 1, we consider that ADTs are described by parameterized specifications and that the abstraction gap between the specifications and the implementations is bridged through refinement mappings. Parameterized specifications are supported by several specification languages. In contrast, refinement mappings were defined in [15] for CONGU specifications [4]. Herein, we revisit this notion and reformulate it in a more general setting.

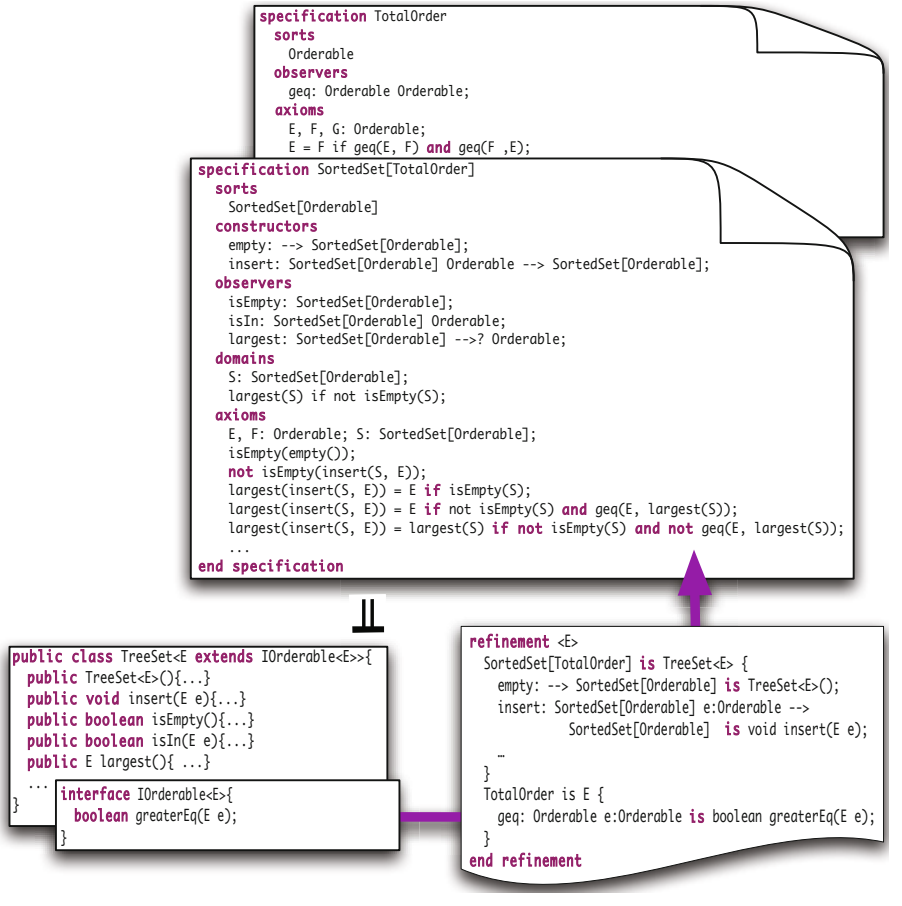


Fig. 1. The aim is to generate tests for checking if a set of classes correctly implements an ADT

Following the tradition of specification-based testing, the developed technique involves considering some abstract tests obtained through the instantiation of the axioms. The difference is that, in our case, this instantiation is not exclusively achieved at the syntactical level by substitution of axiom variables by ground terms; it also involves assigning a value to some variables according to a specific model of the parameter specification. For the generation of abstract tests, the proposed technique relies on Alloy Analyzer [13], a tool that finds finite models of relational structures. Abstract tests are then translated into JUnit tests for a given implementation of the specification. This translation takes into account the correspondence between specifications and Java types defined by the given refinement mapping. Fig. 2 presents an overview of this process.

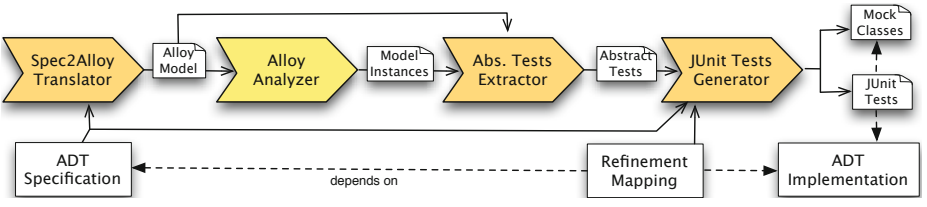


Fig. 2. Overview of the test generation process

The organisation of the paper is as follows. Sec. 2 presents the specifications we have considered in our approach and their semantics. In Sec. 3, we introduce a notion of abstract test appropriate for parameterized specifications and present a technique for the generation of these tests that relies on the encoding of specifications into Alloy and on Alloy Analyzer for finding model instances. In Sec. 4, we show how to automatically translate abstract tests to JUnit tests for a concrete implementation. Sec. 5 presents some evaluation experiments and Sec. 6 concludes the paper and discusses future work.

2 Specifications of Generic Data Types

In algebraic specification, the description of ADTs that admit different versions is supported by parameterized specifications [6]. The description of algebraic specifications in general, and parameterized specifications in particular, is supported by different languages (e.g., [2,6,7]) with significant variations in terms of syntax and semantics. In this section, we present the specifications considered in our approach and their semantics.

Preliminaries. We use Σ to represent a many-sorted signature $\langle S, F, P\rangle$, where S is the set of sorts and F and P are the sets of, respectively, operation and predicate symbols. Moreover, we use $Spec$ to represent a specification $\langle \Sigma, Ax\rangle$, where Ax is the set of axioms described by formulas in first-order logic (with equality). An example of a specification is $TotalOrder$, partially described in Fig. 1 using the language of CONGU. It has the sort $Orderable$, a single predicate symbol geq and no operation symbols. Its set of axioms includes $\forall e{:}Orderable\ \forall f{:}Orderable\ .\ geq(e,f) \wedge geq(f,e) \Rightarrow e = f$.

We use $PSpec$ to represent a parameterized specification, i.e., a pair $\langle Param, Body\rangle$ of specifications with $Param$ (the formal parameter) included in $Body$ (the body). An

example of a parameterized specification is $SortedSet[TotalOrder]$, also partially described in Fig. 1. The formal parameter is $TotalOrder$ and the body is $SortedSet$, a specification that contains what is in $TotalOrder$ and the sort $SortedSet[Orderable]$, the operation symbols $empty$, $insert$ and $largest$, the predicate symbols $isEmpty$ and $isIn$, and several axioms that express the properties of these operations and predicates.

2.1 Specifications

In this work, we restrict our attention to a set of parameterized specifications that can be described in CONGU. More concretely, we consider specifications in which operation symbols are classified as *constructors* or *observers* and that are obtained through the extension of a given specification $Spec$ with *an increment*, i.e.,: (i) a single sort s, (ii) constructors that produce elements of sort s, (iii) observers and predicate symbols that take an element of sort s as first argument, (iv) axioms that express properties of the new operation and predicate symbols only. An increment can define a specification by itself or rely on sorts, operations and predicates available in the base specification $Spec$. In any case, we use $Spec+Spec_s$ to represent the extension of $Spec$ with an increment centred on sort s and $Spec_{s_1}+...+Spec_{s_n}$ to represent a sequence of increments. For the body of $PSpec$, we also require that all increments different from $Param$ include at least one *creator*, i.e., a constructor that does not have elements of the introduced sort among its arguments. In the sequel, we use $Body-Param$ to refer to sorts, operations, predicates and axioms in $Body$ but not in $Param$. It is easy to see that $TotalOrder$ and $SortedSet$ fulfill these requirements: $TotalOrder$ is an extension of the empty specification while $SortedSet$ is an extension of $TotalOrder$ that indeed introduces one creator ($empty$) of the introduced sort ($SortedSet[Orderable]$).

In the sequel we will use $Term_\Sigma$ and $CTerm_\Sigma$ to denote, respectively, the set of ground terms and the set of canonical ground terms (i.e., terms defined exclusively in terms of constructors). For terms, canonical terms and formulas built over a set X of variables typed by sorts in Σ, we use $Term_\Sigma(X)$, $CTerm_\Sigma(X)$ and $Form_\Sigma(X)$.

In what concerns the axioms, we assume they have one of the following forms:

$$\forall x_1 : s_1 ... \forall x_n : s_n \,.\, \phi \qquad \forall x_1{:}s_1 ... \forall x_n : s_n .\, \psi_{op} \Rightarrow \mathit{defined}(op(x_1, ..., x_n))$$

where op is an operation symbol and ϕ, ψ_{op} are quantifier-free first-order logic formulas built over Σ. The first type of axioms is used for expressing usual properties of operations and predicates. The other type of axioms supports the definition of a domain condition of an operation, i.e., the condition under which the operation must be defined (these are needed because operations can be interpreted as partial functions). In $SortedSet[TotalOrder]$, all operations but $largest$ must be interpreted by total functions and, hence, their domain conditions are $true$ while $largest$ has to be defined for non empty sets, i.e., $\forall s : SortedSet[Orderable]. \neg isEmpty(s) \Rightarrow defined(largest(s))$.

We further assume there is exactly one domain condition for each operation, which allows us to define the formula $\mathit{defined}^*(t)$ that, as we will see later on, defines sufficient conditions for the term t to be defined.

Definition 1. *$\mathit{defined}^*(t)$ is the formula defined inductively in the structure of term t as follows: (1) $\mathit{defined}^*(x) = true$ if x is a variable, (2) $\mathit{defined}^*(op(t_1, ..., t_n)) = \mathit{defined}^*(t_1) \wedge ... \wedge \mathit{defined}^*(t_n) \wedge \psi_{op}[t_1/x_1, ...t_n/x_n]$ if $op : s_1, ..., s_n \rightarrow s$ is an operation with domain condition $\forall x_1{:}s_1 ... \forall x_n{:}s_n .\, \psi_{op} \Rightarrow \mathit{defined}(op(x_1, ..., x_n))$.*

2.2 Semantics

Specifications are interpreted in terms of Σ-algebras. More concretely, we take Σ-algebras as triples $A = \langle \{A_s\}_{s \in S}, \mathcal{F}, \mathcal{P} \rangle$, where A_s is the carrier set of sort s, $\mathcal{F}$ defines the interpretation of operation symbols as partial functions and $\mathcal{P}$ defines the interpretation of predicate symbols as relations. We use $[\![t]\!]^{A,\rho}$ to denote the interpretation of a term t in $Term_\Sigma(X)$ with an assignment ρ of X into A, i.e., a function that assigns a value in A_s to each variable $x{:}s$ in X. Given that operation symbols can be interpreted by partial functions, $[\![t]\!]^{A,\rho}$ might not be defined. In fact, $[\![t]\!]^{A,\rho}$ is defined if and only if $A, \rho \vDash \mathit{defined}(t)$. The interpretation of equality also has to take into account the possibility of terms not being defined. Equality is interpreted as being strong, i.e., $t_1 = t_2$ holds in a Σ-algebra A when the values of both terms are defined and equal or both are undefined. In what concerns predicates, when they are applied to undefined terms they are always false (see [2] for the rational of this choice).

There are various forms of semantic construction in the algebraic approach to specification of ADTs. For the purpose at hand, the appropriate construction is loose semantics. It associates to $\mathit{Spec}{=}\langle \Sigma, Ax \rangle$ the class of all Σ-algebras which satisfy its axioms Ax; these are called Spec-algebras. According to this semantics, an implementation of the ADT in which all specified properties hold is considered to be correct.

For parameterized specifications, loose semantics associates to $\langle Param, Body \rangle$ the class of functions $\mathcal{T}_{Body}$ that assign to each $Param$-algebra A, a $Body$-algebra $\mathcal{T}_{Body}(A)$ that coincides with A when restricted to $Param$. This means that an implementation of a parameterized specification is correct if it has all the specified properties, when instantiated with any correct implementation of its parameter.

3 Generation of Abstract Tests

The envisaged strategy for deriving test cases for implementations of generic data types encompasses the generation of tests for their parameterized specifications. We call them abstract tests because their target are abstract models (algebras). For testing Java implementations, we need to convert them into object-oriented tests (JUnit tests, in our case).

3.1 Tests for Parameterized Specifications

A test for an algebraic Spec is usually defined as a ground and quantifier-free formula that is a semantic consequence of Spec and, hence, valid in every Spec-algebra [8]. This can be generalised to parameterized specifications but the result is not interesting as specifications used as parameters are not expected to have creators and so, the corresponding set of ground terms is empty. In fact, specifications used as parameters are not expected to have constructors as they often correspond to a required "ability". For instance, in our example, $\mathit{TotalOrder}$ corresponds to a requirement for the actual parameter of a sorted set to have a comparison operation that defines a total order.

The notion of test that we found useful for parameterized specifications is one in which we fix a specific $Param$-algebra.

Definition 2. *A* closed test *for a parameterized specification* $PSpec = \langle Param, Body \rangle$ *is a tuple* $\langle A, X, \phi, \rho_P, \rho_B \rangle$ *where* A *is a* $Param$*-algebra;* X *is a finite set of variables*

typed by sorts in Body; ϕ is a quantifier-free logic formula in $Form_\Sigma(X)$; ρ_P is an assignment of X_P into A, where X_P is the set of variables in X typed by sorts in Param; ρ_B is a function that assigns a term $\rho_B(x)$ in $Term^s_\Sigma(X_P)$ to each $x{:}s$ in $X_B{=}X\backslash X_P$; such that $\mathcal{T}_{Body}(A), \rho_P \vDash \rho^*_B(\phi)$, for every $\mathcal{T}_{Body}$ in the semantics of PSpec, where $\rho^*_B(\phi)$ is the translation of formulas induced by ρ_B.

Notice that, in these tests, the instantiation of the variables in the formula is achieved through the combination of (i) a syntactic replacement of variables in X_B by terms and (ii) an assignment of variables in X_P into the fixed $Param$-algebra. In this way, we can exercise the test in any Σ_{Body}-algebra that extends A.

We are interested in tests that result from the instantiation of axioms of the form $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$. Closed tests may involve the replacement of variables by terms and their interpretation in a specific Σ_{Body}-algebra might be undefined and, hence, this instantiation needs to be conditioned by the definedeness of these terms. Because the formula $defined^*(t)$ provides a sufficient condition for the term t to be defined in any $Spec$-algebra (for details, see [1]), we can use the formula $\wedge_{x \in X_B} defined^*(\rho_B(x)) \Rightarrow \phi$.

Proposition 1. *Let $PSpec = \langle Param, Body \rangle$ be a parameterized specification and $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$ an axiom in Body. If A is a $Param$-algebra, X is a set of variables including $\{x_1{:}s_1, ..., x_n{:}s_n\}$, ρ_P is an assignment of X_P into A and ρ_B is a function that assigns a term $\rho_B(x)$ in $Term^s_\Sigma(X_P)$ to each $x{:}s$ in X_B, then*

$$\langle A, X, (\wedge_{x \in X_B} defined^*(\rho_B(x))) \Rightarrow \phi, \rho_P, \rho_B \rangle$$

is a closed test for PSpec.(See [1] for the proof.)

As an example, let us consider the axiom $\forall s{:}SortedSet[Orderable].\ \forall e{:}Orderable.\ \neg isEmpty(insert(s, e))$ of $SortedSet[TotalOrder]$. As a result of Prop. 1:

- the $TotalOrder$-algebra TO^2 with two elements, say, $Ord0$ and $Ord1$ and geq interpreted as the relation $\{(Ord1, Ord0), (Ord1, Ord1), (Ord0, Ord0)\}$
- the set of variables $\{s{:}SortedSet[Orderable], e{:}Orderable\}$
- the formula $true \Rightarrow \neg isEmpty(insert(s, e))$
- ρ_P: $\{e{:}Orderable \mapsto Ord0\}$ and ρ_B: $\{s{:}SortedSet[Orderable] \mapsto empty()\}$

defines a closed test for $SortedSet[TotalOrder]$.

3.2 Generation Technique

When tests are obtained through ground instantiation of axioms, performing a test experiment just requires evaluating a ground formula in the IUT. The generation of closed tests for parameterized specifications also involves the instantiation of axioms, but this instantiation is only partial — the instantiation of an axiom involving a set of variables X is limited to the variables in X_B. Hence, the generation of closed tests involves the generation of models for the parameter specification and evaluations in these models for the variables in X_P. In this subsection, we describe a technique for the generation of abstract test suites for parameterized specifications that can be subsequently translated into JUnit test suites for testing Java implementations.

As pointed out in [8], test thoroughness is increased by generating multiple test cases for each axiom, through the partitioning of each axiom into a finite set of "cases", either by successively unfolding the premises of equational axioms or by considering

the conjunctive terms in the Disjunctive Normal Form (DNF) of the axiom expression. In our case, since axioms are not restricted to equational ones, DNF partitioning is more directly applicable, with the advantage of not mixing together different axioms. To further assure that the different cases are disjoint, and hence avoid generating redundant tests, we take a special DNF form — the Full Disjunctive Normal Form (FDNF). The FDNF of a logical formula that consists of Boolean variables connected by logical operators is a canonical DNF in which each Boolean variable appears exactly once (possibly negated) in every conjunctive term (called a minterm) [9].

The technique consists in considering each of the axioms $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi$ in $Body{-}Param$ that do not express a domain condition and start by converting it to FDNF. Assuming that the result is $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \phi_1 \vee ... \vee \phi_k$ then, for every $1 \leq i \leq k$, the technique involves using Alloy Analyzer to find a $Body$-algebra M such that: (1) M is finite; (2) M satisfies sort generation constraints for sorts in $Body{-}Param$ (each of these sorts is constrained to be generated by the declared constructors); (3) M satisfies a stronger version of the domain condition of every operation op in $Body{-}Param$: $\forall x_1{:}s_1 ... \forall x_n{:}s_n.\ \psi_{op} \Leftrightarrow \mathit{defined}(op(x_1, ..., x_n))$; (4) M satisfies $\exists x_1{:}s_1 ... \exists x_n{:}s_n.\ \phi_i$.

Only axioms in $Body{-}Param$ are considered because these are the axioms that express the properties of the generic data type that we are interested to check in the IUT (the other axioms concern properties that are expected to hold in actual parameters). In what concerns the constraints imposed on the finding of the $Body$-algebra (the model instances of the Alloy specification): condition 1 is a requirement imposed by the model finder tool, which limits search to finite models; condition 2 excludes models that have junk in the carrier sets as we will need to subsequently convert the elements of this model to arbitrary Σ_{Body}-algebras that extend $M|_{Param}$ (the restriction of M to $Param$); condition 3 avoids the generation of some models that define an evaluation for terms that are undefined in other $Body$-algebras; condition 4 ensures we get from the model finder tool an assignment ρ of the variables in ϕ_i into M satisfying it. Since the formula is in FDNF, all variables of the axiom occur in ϕ_i and, hence, ρ is an assignment of $X{=}\{x_1{:}s_1, ..., x_n{:}s_n\}$ into M.

Because M restricted to sorts in $Body{-}Param$ is a generated model, for each x in X_B, there exists (i) a canonical term $t_x \in CTerm_{\Sigma}(Y_x)$ for some set Y_x of variables typed by sorts in $Param$ and disjoint from X, and (ii) an assignment ρ_x of Y_x into M such that $[\![t_x]\!]^{M,\rho_x} = \rho(x)$. This family of terms and assignments can be used to define a closed test for $PSpec$ as follows:

$$\langle M|_{Param}, X', \phi', \rho_P, \rho_B \rangle$$

- $M|_{Param}$ is the restriction of M to $Param$
- $X' = \cup_{x \in X_B} Y_x \cup X$
- ϕ' is the formula $(\wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x))) \Rightarrow \phi$
- ρ_P coincides with ρ for X_P and with ρ_x for Y_x, for every $x \in X_B$
- ρ_B is the function that maps each x in X_B into t_x

The correctness of the proposed technique is an immediate consequence of Prop. 1.

Proposition 2. *The tuple* $\langle M|_{Param}, X', \phi', \rho_P, \rho_B \rangle$ *is a closed test for* $PSpec$.

Consider, for instance, the axiom of $SortedSet[TotalOrder]$:

$$\forall s{:}SortedSet[Orderable].\forall e{:}Orderable.$$
$$\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \Rightarrow largest(insert(s, e)) = largest(s)$$

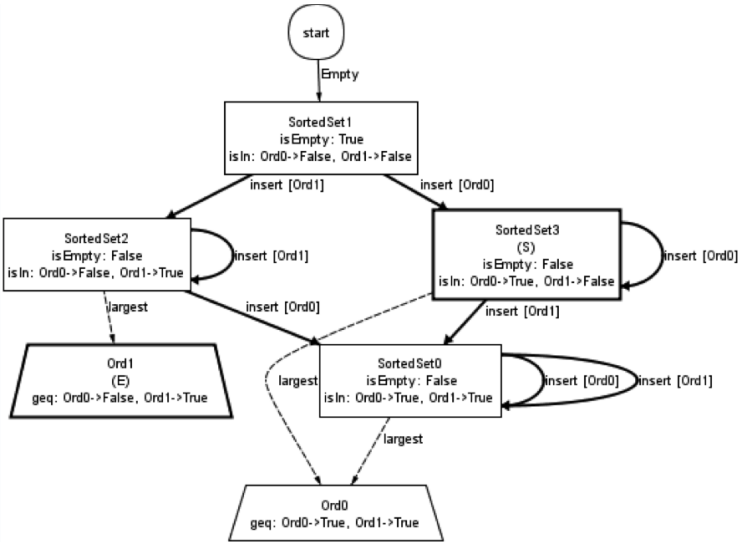


Fig. 3. A model instance defining a $SortedSet$-algebra and an assignment to variables e and s

One minterm of the corresponding FDNF is $\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \wedge largest(insert(s, e))$=$largest(s)$. The application of the technique just described involves using Alloy Analyzer to obtain a $SortedSet$-algebra that satisfies this minterm (and fulfills the other three requirements described before). Fig. 3 presents an example of one of these algebras (referred as SS^2 in the sequel), represented as an Alloy model instance. In fact, it also defines ρ:$\{e$:$Orderable \mapsto Ord1$, s:$SortedSet[Orderable] \mapsto SortedSet3\}$. The last step is to find a pair $\langle t_s, \rho_s \rangle$ that represents $SortedSet3$. Through the analysis of SS^2, we find the term $insert(empty(), f)$ and the assignment ρ_s: $\{f$:$Orderable \mapsto Ord0\}$. As a result, we obtain the closed test $\langle TO^2, X', \phi', \rho_P, \rho_B \rangle$ for $SortedSet[TotalOrder]$, where

- X'=$\{s$:$SortedSet[Orderable], e$:$Orderable\} \cup Y_s$ with $Y_s = \{f$:$Orderable\}$
- ϕ' is $defined^*(insert(empty(), f)) \Rightarrow$ $(\neg isEmpty(s) \wedge \neg geq(e, largest(s)) \Rightarrow largest(insert(s, e))$=$largest(s))$, with $defined^*(insert(empty(), f)) = true \wedge true \wedge true$
- ρ_P is the assignment $\{e \mapsto Ord1, f \mapsto Ord0\}$
- ρ_B is the function $\{s \mapsto insert(empty(), f)\}$.

The number of tests generated for each axiom is in general smaller than the number of minterms in the corresponding FDNF since a minterm may not be satisfiable by $Body$-algebras. In particular, this happens in minterms that require the satisfaction of the negation of the definedness condition of some term. This is the case of the minterm $isEmpty(s) \wedge \neg geq(e, largest(s)) \wedge largest(insert(s, e))$=$largest(s)$.

3.3 From Algebraic Specifications to Alloy and Back

The technique just described requires the ability to generate Alloy models from a parameterized specification $PSpec$=$\langle Param, Body \rangle$ and model finding commands for

```
sig Orderable  {
    geq: Orderable -> one BOOLEAN/Bool
}
sig SortedSet {
    insert:Orderable -> one SortedSet,
    isEmpty:one BOOLEAN/Bool,
    isIn:Orderable ->one BOOLEAN/Bool,
    largest: lone Orderable
}
one sig start{
    empty: one SortedSet
}
fact SortedSetConstruction{
    SortedSet in (start.empty).*{x: SortedSet, y: x.insert[Orderable]}
}
fact domainSortedSet0{
  all S:SortedSet |
     S.isEmpty != BOOLEAN/True implies one S.largest else no S.largest
}
fact axiomSortedSet4{
    all E:Orderable, S:SortedSet |
       (S.isEmpty = BOOLEAN/False and E.geq[S.largest] = BOOLEAN/False)
          implies (S.insert[E].largest = S.largest)
}
// ... other axioms of Orderable and SortedSet
run run_axiomSortedSet4_0{
    some E:Orderable, S:SortedSet |
       S.isEmpty = BOOLEAN/False and E.geq[S.largest] = BOOLEAN/False
       and S.insert[E].largest = S.largest
} for 6 but exactly 2 Orderable
// ... other run commands for other minterms and axioms
```

Fig. 4. Excerpt of the Alloy model and run commands for $SortedSet[TotalOrder]$

Alloy Analyzer (Alloy "run" commands) and, in the end, to extract abstract closed tests from the model instances found by Alloy Analyzer.

Encoding of Algebraic Specifications in Alloy. The encoding of $PSpec$ in Alloy takes into account the sorts, operations, predicates and axioms in $Body$ and, at the same time, has to ensure conditions 2 and 3 of Sec. 3.2: there is no junk in the parameterized sorts and partial operations are defined if and only if their domain condition holds.

Due to space limitation we explain the translation rules (presented in full detail in [1]) using our running example. Fig. 4 shows an excerpt of the Alloy model produced for $SortedSet[TotalOrder]$. Sorts are translated into Alloy signatures. A special signature $start$ with a single instance is defined to represent the root of the graph view of each model instance found by Alloy Analyzer (see Fig. 3), holding fields corresponding to the creators of all sorts (e.g., $empty$). Other operations and predicates are encoded as fields of the signature corresponding to their first argument. To allow this encoding for predicates without further arguments (e.g., $isEmpty$), predicates are handled as operations of return type $Boolean$. Partial operations (e.g., $largest$) originate fields with $lone$ multiplicity (0 or 1) and a fact encoding their (strong) domain condition. To exclude junk for a sort s, a fact is introduced (e.g., $SortedSetConstruction$ fact in Fig. 4) imposing that all its instances are generated by applying constructors (a creator followed by other constructors). When constructors have extra arguments that also have to be constructed, it is necessary to ensure that all instances can be constructed in an acyclic way, e.g., by imposing in the construction fact that, in each step, it is possible to construct an instance y by using only instances x_1, ..., x_n that precede y in a partial

ordering (to be found by Alloy Analyzer) of all instances. Axioms are straightforwardly encoded as Alloy facts (e.g., $axiomSortedSet4$ in Fig. 4).

Generation of Model Finding Commands. In order to find a model instance that satisfies each minterm of the FDNF of each axiom in $Body-Param$, a "run" command that encodes condition 4 of Sec. 3.2 is generated. This is illustrated in the bottom of Fig. 4 for the same axiom and minterm used in the example of Sec. 3.2. The exploration bounds can be configured by the user. In the example of Fig. 4, we are searching for models with at most 6 instances of each signature and exactly 2 instances of $Orderable$.

Extraction of Abstract Tests from the Model Instances Found. When a "run" command is executed, each model instance found by Alloy Analyzer can be visualized as a graph as illustrated in Fig. 3. From this instance an abstract test can be extracted as partially explained in Sec. 3.2. The canonical term to be assigned to each variable x in X_B (e.g., S in Fig. 3) is obtained by following a path from the $start$ node to the node assigned to that variable (e.g., $SortedSet3$). When constructors have extra parameters that have also to be constructed, only paths obeying the partial ordering of all instances imposed by the construction fact are considered. In the example, the extracted Alloy expression is $start.empty.insert[Ord0]$. Since a canonical term cannot contain elements of carrier sets, values of parameter sorts ($Ord0$ in this case) are replaced by variables in the expression and their values are recorded in an assignment (ρ_P).

Prop. 1 ensures the correctness of the test generation technique in abstract terms. Obviously, the preservation of this correctness result depends on how specifications are encoded into Alloy. Concretely, it is necessary to ensure that all model instances of the generated model, restricted to the elements of $Param$, define a $Param$-algebra. For the encoding technique presented in this section, all model instances of the generated Alloy model define a $Body$-algebra.

4 From Abstract Tests to JUnit Tests

In this work, we focus on Java implementations of ADTs. Hence, we consider implementations of parameterized specifications to be sets of Java classes and interfaces, some of them defining generic types. The challenge we address in this section is the translation of the abstract tests generated for the parameterized specification with the help of Alloy Analyzer to concrete JUnit tests. The goal of these tests is to exercise the IUT, instantiating its parameters with mock classes and mock objects derived from the abstract tests.

The translation of abstract into concrete tests requires that a correspondence between what is specified algebraically and what is programmed is defined. We assume this correspondence is defined in terms of a refinement mapping. This notion, defined in the context of CONGU specifications in [15], is formulated in a more general setting.

4.1 Refinement Mappings

The correspondence between specifications and Java types as well as between operations/predicates and methods can be described in terms of what we have called a *refinement mapping*. We will restrict our attention to the set of specifications described in

Sec. 2.1. Hence, in the rest of this section we will consider a parameterized specification $PSpec$ with $Body$ defined by $\mathcal{B} = Spec_{s_1}+...+Spec_{s_n}+Spec_p+Spec_{t_1}+...+Spec_{t_k}$ in which $Spec_p$ corresponds to the parameter specification. Moreover, to ease the presentation, we will consider that $Spec_p$ has a single sort and all increments $Spec_{t_i}$ depend on $Spec_p$ (i.e., they cannot be moved to a position on the left of $Spec_p$). For the same reason, we also consider only Java generic types with a single parameter.

Definition 3. *A* refinement mapping *from $\mathcal{B}$ to a set $\mathcal{C}$ of Java types consists of a type variable V and an injective refinement function $\mathcal{R}$ that maps:*

- *each s_i to a non-generic type defined by a Java class in $\mathcal{C}$;*
- *each t_i to a generic type with a single parameter, defined by a Java class in $\mathcal{C}$;*
- *p to the type variable V;*
- *each operation/predicate of $Spec_s$, with $s \in \{s_1,...,s_n,t_1,...,t_k\}$, to a method of the corresponding Java type $\mathcal{R}(s)$ with a matching signature: **(i)** every n-ary creator corresponds to an n-ary constructor; **(ii)** every other (n+1)-ary operation/predicate symbol corresponds to an n-ary method (object* **this** *corresponds to the first parameter of the operation/predicate); **(iii)** every predicate symbol corresponds to a boolean method; **(iv)** every operation with result sort s corresponds to a method with any return type, void included, and every operation with a result sort different from s corresponds to a method with the corresponding return type; **(v)** the i-th parameter of the method that corresponds to an operation/predicate symbol has the type corresponding to its (i+1)-th parameter sort;*
- *each operation/predicate of $Spec_p$ to a matching method signature such that, for $1 \leq i \leq k$, we can ensure that any type K that can be used to instantiate the parameter of the generic type $\mathcal{R}(t_i)$ possesses all methods defined by $\mathcal{R}$ for type variable V after appropriate renaming — the replacement of all instances of V by K.*

Fig. 1 partially shows a refinement mapping from $SortedSet[TotalOrder]$ to the Java types {TreeSet<E>,IOrderable<E>}, using CONGU refinement language. We can check if the last condition above holds by inspecting whether any bounds are declared in the class TreeSet for its parameter E, and whether those bounds are consistent with the methods that were associated to parameter type E by the refinement mapping — **boolean** greaterEq(E e). This is indeed the case: the parameter E of TreeSet is bounded to extend IOrderable<E>, which, in turn, declares the method **boolean** greaterEq(E e).

4.2 Mock Classes and JUnit Tests

In order to test generic classes against their specifications, finite mock implementations of their parameters are automatically generated, comprising *mock classes*, that are independent of the generated abstract tests, and *mock objects*, instances of mock classes that are created and set up in each test method according to a specific abstract test.

Mock Classes. For the parameter sort p, a *mock class* named pMock is generated. This class will be used to instantiate the parameter of all generic types $\mathcal{R}(t_i)$ and, hence, has to implement all the interfaces that bound these parameters. For instance, in our example, the class OrderableMock was generated (see Fig. 5) implementing IOrderable<OrderableMock> because the parameter E of TreeSet is bounded to extend

```
public class OrderableMock implements IOrderable<OrderableMock> {
      private HashMap<OrderableMock, Boolean> greaterEqMap =
            new HashMap<OrderableMock, Boolean>();
      public boolean greaterEq(OrderableMock o) {return greaterEqMap.get(o);}
      public void add_greaterEq(OrderableMock o, boolean result) {
            greaterEqMap.put(o, result);
      }
}
```

Fig. 5. Mock class generated from the refinement mapping in Fig.1

IOrderable<E>. The mock class defines extensional implementations of all interface methods that correspond to operations or predicates of the parameter specification (in our example, just the method greaterEq). More concretely, for each interface method m, the mock class provides: a hash map mMap, to store the method return values for allowed actual parameters; an add_m method, to be used by the test setup code to define the above return values; and an implementation of m itself, that simply retrieves the value previously stored in the hash map.

JUnit Tests: Axiom Tester Method. Each axiom $\forall x_1{:}s_1 ... \forall x_n{:}s_n \,.\, \phi$ in $Body-Param$ not defining a domain condition is encoded as a method (reused by all test methods generated for that axiom) with the axiom variables as parameters and a body that evaluates and checks the value of ϕ for the given parameter values (see axiomSortedSet4Tester in Fig. 6). In the case of a variable x_k of a parameterized sort s_k, since operations of s_k may be mapped to methods with side effects (see the case of insert in Figs. 1 and 6), a factory object (of type Factory<s_k>) is expected as parameter instead of an object of type s_k, to allow the creation of as many copies as needed of x_k (a copy for each occurrence of x_k in ϕ) without depending on the implementation of clone. This way, methods with side effects can be invoked on one copy without affecting the other copies. Sub-expressions involving operations mapped to **void** methods are evaluated in separate instructions (see insert in Fig. 6). Equality is evaluated with the equals method.

JUnit Tests: Test Methods Encoding Abstract Tests. For each abstract test

$$\langle A, X, \wedge_{x \in X_B} \mathit{defined}^*(\rho_B(x)) \Rightarrow \phi, \rho_P, \rho_B \rangle$$

generated according to the technique described in Sec. 3.2, a concrete JUnit test method is generated comprising three parts (for an example, see Fig. 6):

- **Mock Objects:** Creation of mock objects (instances of mock classes) for the values in the carrier set of A, and addition of tuples for the functions and relations in A.
- **Factory Objects:** Creation of a factory object of type Factory<s> for each variable $x : s$ in X_B, that constructs an object of type s upon request according to the term $\rho_B(x)$ and the mapping ρ_P. The verification of the condition $\mathit{defined}^*(\rho_B(x))$ is performed incrementally in each step of the construction sequence, by checking the domain condition before applying any operation with a defined domain condition and issuing a warning in case it does not hold (not needed in the example).
- **Axiom Verification**: Invocation of the method that checks ϕ, passing as actual parameters the factory objects prepared in the previous step (for the variables in X_B) and the values defined in ρ_P (for the remaining variables).

```
private interface Factory<T> {T create();}
private void axiomSortedSet4Tester(Factory<TreeSet<OrderableMock>> sFact,OrderableMock e) {
      TreeSet<OrderableMock> s_0 = sFact.create();
      TreeSet<OrderableMock> s_1 = sFact.create();
      if(!s_0.isEmpty() && !e.greaterEq(s_0.largest())) {
            s_1.insert(e);
            assertTrue(s_1.largest().equals(s_0.largest()));
      }
}
@Test public void test0_axiomSortedSet4_0(){
      // mock objects for the parameter
      final OrderableMock ord0 = new OrderableMock();
      final OrderableMock ord1 = new OrderableMock();
      ord0.add_greaterEq(ord0, true);
      ord0.add_greaterEq(ord1, false);
      ord1.add_greaterEq(ord0, true);
      ord1.add_greaterEq(ord1, true);
      // factory objects for the axiom var's of parameterized type
      Factory<TreeSet<OrderableMock>> sFact =
         new Factory<TreeSet<OrderableMock>>() {
            public TreeSet<OrderableMock> create() {
                  TreeSet<OrderableMock> s = new TreeSet<OrderableMock>();
                  s.insert(ord0);
                  return s; }
      };
      // checking the axiom
      axiomSortedSet4Tester(sFac, ord1);
}//… other axioms and test cases
```

Fig. 6. Excerpt of JUnit test code generated corresponding to the model instance shown in Fig. 3

5 Evaluation

To assess the efficacy (defect detection capability of the generated test cases) and efficiency (time spent) of the proposed technique, an experiment was conducted using different specifications. Herein, we report on the results of the experiment with our running example. We started by generating abstract test cases for the specification *SortedSet*. We measured the time spent by Alloy Analyzer on finding model instances for the several run commands (axiom cases) and the number of run commands for which instances were found. For the ones that Alloy Analyzer could not find instances, a manual analysis was conducted to determine whether they could be satisfied with other search settings (exploration bounds). After that, JUnit test cases generated from abstract tests and the refinement mapping to TreeSet and IOrderable were executed to check the correctness of the implementation and of the test suite. Subsequently, a mutation analysis was performed to assess the quality of the test suite. Mutants not killed by the test suite were manually inspected to determine if they were equivalent to the original code, and additional test cases were added to kill the non-equivalent ones. A test coverage analyses was also performed as a complementary test quality assessment technique. The experiment was conducted on a portable computer with a 32 bits Intel Core 2 Duo T6600 @ 2.20 GHz processor with 2.97 GB of RAM, running Microsoft's Windows 7. The results are summarized in Table 1.

In terms of efficiency, we concluded that the time spent in finding model instances ($\sim$ 2 minutes) is not a barrier for the adoption of the proposed approach. The percentage of axiom cases for which a model instance was not found was significant (44%). A manual analysis showed that these cases were not satisfiable. Mutation analysis revealed some parts of the implementation of *equals* and *largest* that were not adequately

Table 1. Experimental results for the SortedSet example

Item	Sorted Set
Size of algebraic specification ($Body - Param$)	25 lines [(1)]
Total number of axioms	9
With instances found in all axiom cases	5
With instances found in some axiom cases	4
Total number of axiom cases (minterms)	36
Number of cases for which instances were found [(2)]	20 (56%)
Number of cases for which no instances were found [(2)(3)]	16 (44%)
Time spent by Alloy analyzer finding instances [(2)]	129 sec
Number of JUnit test cases generated [(4)]	20
Size of Java implementation under test	77 lines [(1)]
Number of failed test cases	0
Total number of mutants generated (with Jumble [12])	41
Killed by the original test suite	35 (85%)
Not killed by the original test suite	6 (15%)
Equivalent to original implementation	0
Not equivalent to original implementation [(5)]	6
Coverage of Java implementation under test (measured with Eclemma [10])	96,9%
Number of added test cases to kill all mutants (and achieve 100% code coverage)	3

[(1)] Ignoring comments and blank lines.[(2)] In this experiment, the exploration was limited to at most 12 instances per sort, but exactly 3 $Orderable$. [(3)] Manual analysis showed that these cases were not satisfiable. [(4)] Only one test case was generated for each satisfiable axiom case (corresponding to the first instance retrieved by Alloy Analyzer). [(5)] Related to method invocation outside the domain and to insufficient testing of $equals$ (lack of inequality cases).

exercised, due to the fact that conditions for inequality are not explicitly specified and consequently not tested in this example, and due to the fact that the behaviour of operations outside their domain (in the example, the behaviour of *largest* over an empty set) is not specified and consequently not tested.

6 Conclusions and Future Work

Although test generation from algebraic specifications has been thoroughly investigated, existing approaches are based on flat specifications. In this paper, we have discussed testing from parameterized specifications and put forward a notion of a closed test appropriate for these specifications, which generalises the standard notion of test as a quantifier-free ground formula. Then, based on closed tests, we presented an approach for the generation of unit tests for Java implementations of generic ADTs from specifications in which the generated test code includes finite implementations (mocks) of the parameters. This paper addresses the foundational aspects of the approach. A tool that fully automates the test generation from specifications is currently under development. The tool supports the translation of CONGU specifications to Alloy, the translation of model instances found by Alloy Analyzer to JUnit as well as the tuning of exploitation bounds. We envisage the tool can also provide support for other related problems, like the automatic generation of actual parameters for methods when the type parameters are interfaces (e.g., comparator in TreeSet<E>(Comparator<E> comparator)).

The proposed approach relies on a translation of specifications into Alloy and on the capability of Alloy Analyzer to find model instances that satisfy given properties—in our case, the minterms of the FDNF representation of each axiom. In the conducted experiments, Alloy Analyzer was able to find model instances for all theoretically satisfiable axiom cases in a moderate time. Mutation testing and code coverage analyses showed that the generated test cases were of high quality, because they were able to kill all the mutants and cover all the code apart from behaviours that were not explicitly specified (behaviour outside operation domains and conditions for inequality).

Although Alloy Analyzer has scalability limitations due to the time required to find instances of complex models, we did not find that to be an issue for unit testing ADTs. The fact that Alloy Analyzer only performs model-finding over restricted scopes consisting of a user-defined finite number of objects is what imposes a limitation of the approach presented: the inability to generate tests for ADTs that do not admit finite models, such as unbounded stacks (since the domain of push is true, it is always possible to create a bigger stack). To overcome that problem, we are currently working on an extension of the approach to automatically handle that kind of specifications, that encompasses transforming constructors into partial functions in the Alloy model and inserting definedeness guard conditions in the axioms that use those constructors.

As future work, we also intend to extend the approach to rule out automatically by static analysis unsatisfiable axiom cases and generate tests outside operations' domains and for properties not explicitly included in specifications such as those related with the fact that equality is a congruence. This will reduce the dependence of the approach on the correct implementation of equals.

Acknowledgement. This work was partially supported by FCT under contract PTDC/EIA-EIA/103103/2008.

References

1. Andrade, F., Faria, J.P., Lopes, A., Paiva, A.: Specification-driven unit test generation for Java generic classes (2011), http://paginas.fe.up.pt/~jpf/research/TR-QUEST-2011-01.pdf
2. Bidoit, M., Mosses, P.: CASL User Manual. LNCS, vol. 2900, pp. 3–9. Springer, Heidelberg (2004)
3. Chen, H.Y., Tse, T.H., Chen, T.Y.: TACCLE: a methodology for object-oriented software testing at the class and cluster levels. ACM Trans. Softw. Eng. Methodol. 10, 56–109 (2001)
4. Crispim, P., Lopes, A., Vasconcelos, V.T.: Runtime Verification for Generic Classes with CONGU2. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 33–48. Springer, Heidelberg (2011)
5. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. 3, 101–130 (1994)
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations und Initial Semantics. Monographs in Theoretical Computer Science (EATCS), vol. 6. Springer, Heidelberg (1985)
7. Futatsugi, K., Goguen, J.A., Jouannaud, J.-P., Meseguer, J.: Principles of OBJ2. In: Proceedings of the 12th POPL, pp. 52–66. ACM, New York (1985)

8. Gaudel, M.-C., Le Gall, P.: Testing Data Types Implementations from Algebraic Specifications. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 209–239. Springer, Heidelberg (2008)
9. Hein, J.L.: Discrete Structures, Logic, and Computability. Jones & Bartlett Publishers (2009)
10. Hoffmann, M.R.: Ecclema: Java code coverage tool for Eclipse, http://www.eclemma.org/
11. Huges, M., Stotts, D.: Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In: Proc. ISSTV, pp. 53–61. ACM (1996)
12. Irvine, S.A., Pavlinic, T., Trigg, L., Cleary, J.G., Inglis, S., Utting, M.: Jumble Java byte code to measure the effectiveness of unit tests (2007), http://jumble.sourceforge.net/
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
14. Mackinnon, T., Freeman, S., Craig, P.: Endotesting: Unit testing with mock objects. In: eXtreme Programming and Flexible Processes in Software Engineering – XP 2000 (2000)
15. Nunes, I., Lopes, A., Vasconcelos, V.T.: Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 115–131. Springer, Heidelberg (2009)
16. The Stanford Natural Language Processing Group, http://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/util/package-tree.html
17. Yu, B., King, L., Zhu, H., Zhou, B.: Testing Java components based on algebraic specifications. In: Proc. International Conference on Software Testing, Verification and Validation, pp. 190–198. IEEE (2008)

Specifying UML Protocol State Machines in Alloy*

Ana Garis[1], Ana C.R. Paiva[2], Alcino Cunha[3], and Daniel Riesco[1]

[1] Universidad Nacional de San Luis, Argentina
{agaris,driesco}@unsl.edu.ar
[2] DEI-FEUP, Universidade do Porto, Portugal
apaiva@fe.up.pt
[3] HASLab, INESC TEC and Universidade do Minho, Portugal
alcino@di.uminho.pt

Abstract. A UML *Protocol State Machine* (PSM) is a behavioral diagram for the specification of the external behavior of a class, interface or component. PSMs have been used in the software development process for different purposes, such as requirements analysis and testing. However, like other UML diagrams, they are often difficult to validate and verify, specially when combined with other artifacts, such as *Object Constraint Language* (OCL) specifications. This drawback can be overcome by application of an off-the-shelf formal method, namely one supporting automatic validation and verification. Among those, we have the increasingly popular Alloy, based on a simple relational flavor of first-order logic. This paper presents a model transformation from PSMs, optionally complemented with OCL specifications, to Alloy. Not only it enables automatic verification and validation of PSMs, but also a smooth integration of Alloy in current software development practices.

Keywords: UML, OCL, Protocol State Machines, Alloy.

1 Introduction

UML state machine diagrams can be used to describe the dynamic behavior of a system or part of it. There are two variants, namely *Behavioral State Machines* and *Protocol State Machines* (PSMs) [16]. While the former is used to express behavior of various elements (e.g., class instances), the latter is a way to define the allowed behavior of classifiers; namely, classes, interfaces and components. Therefore, PSMs enable the specification of a lifecycle for objects or an order of invocation of its operations and to express usage protocols. PSMs typically omit implementation details and allow the use of the *Object Constraint Language* (OCL) [17] to specify state invariants and transitions' pre- and post-conditions. As such, PSMs are well-suited to be integrated in a *Model Driven Engineering* (MDE) context, allowing the specification of the allowed behavior of a classifier

* Work partially supported by FCT (Portugal) under contract PTDC/EIA-EIA/103103/2008.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 312–326, 2012.

in a highly abstract way. PSMs have been used for the specification of dynamic views during the analysis phase, and they have been exploited for the generation of class contracts, test code and test cases [19,18,3,21].

Since UML is the industry de facto language for modeling, there exist myriad tools supporting it. In particular, UML integration in the *Model Driven Architecture* (MDA), the MDE initiative of the OMG, led to an explosion of UML based MDE tools, such as code generators and reverse engineering frameworks. Unfortunately, in part due to the fact that UML has only an informally given semantics, most of these tools do not offer adequate support for *Verification and Validation* (V&V).

Formal methods have been successfully applied in the formalization and V&V of UML state machines [22,23,12,5,21]. However, the consistency between these and other UML specification artifacts has rarely been addressed. Moreover, most of these formalizations rely on traditional formal methods, that are avoided by software developers due to the inherent complexity that makes them hard to learn and use.

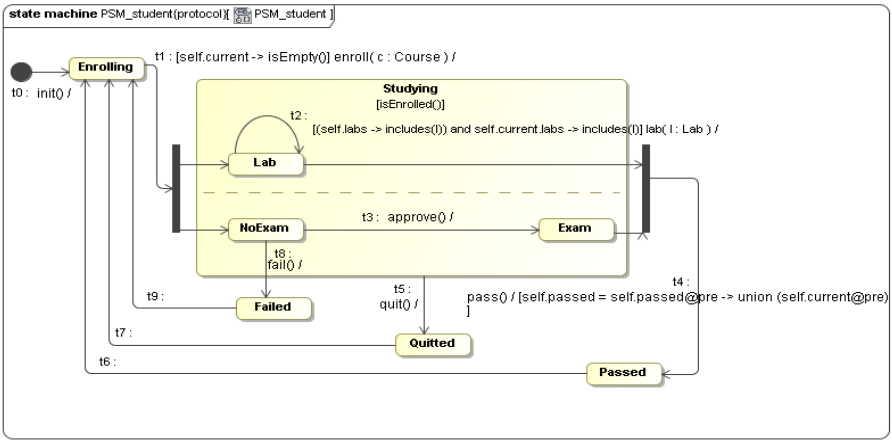
The objective of this paper is precisely to tackle both these issues: we show how both PSMs and Class Diagrams (CDs) enhanced with OCL can be formalized in Alloy [10] lightweight formal modeling language; and we present an approach to develop V&V tasks using Alloy Analyzer. This formalization allows us to simulate and verify the consistency between UML artifacts and to perform other V&V activities, such as detect unreachable states or invalid transitions. The formalization of PSMs is implemented using the model transformation language ATL [4]. For the formalization of OCL, we use the UML2Alloy tool [2] and the approach presented in [1] but adapted to support dynamic behavior.

The rest of the paper is structured as follows. Section 2 shows a case study in order to explain our proposal. Section 3 describes preliminary concepts referred to PSMs and Alloy. Section 4 presents our approach. Section 5 discusses the related work. Finally, Section 6 summarizes the contributions and exposes some ideas for future work.

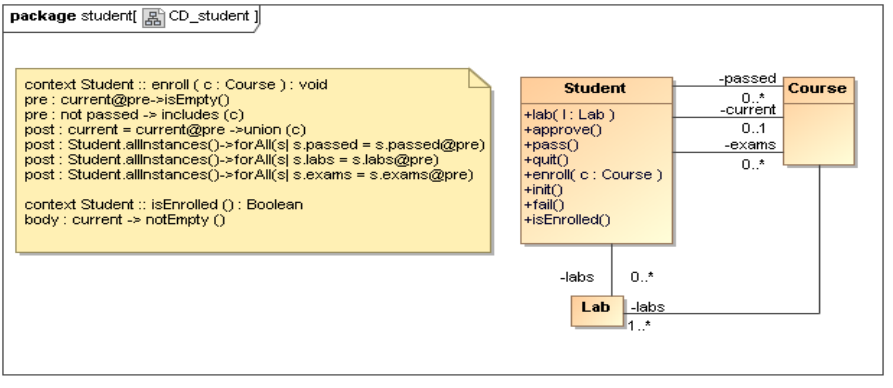
2 Case Study

Figure 1(a) shows an example of a PSM. It is a simplified model of a student coursing a career. The PSM describes the intended behavior of the class `Student`, specified also in CD enriched with OCL shown in Figure 1(b). Initially, the student is not enrolled to any course. The `enroll` operation enrolls the student in a course, and enables him to attend the course exam, while performing `lab` assignments. If he is `approved` in the exam and completes all mandatory lab assignments he can `pass` the course. If he `fail`s the exam, he also fails the course. At any time, he can `quit` the course. After failing, quitting, or passing a course he returns to the `Enrolling` state where he can enroll in another (or the same) course.

Note that the transitions labeled with operations `enroll`, `lab`, and `pass` have attached pre- and post-conditions defined in OCL. These constrain when such



(a) Protocol State Machine



(b) Class Diagram enriched with OCL

Fig. 1. Coursing case study

operations can be invoked and their effect on the modeled student state (namely, associations `passed`, `current`, `exams`, and `labs`). Likewise, the `Studying` composite state is characterized by an invariant, forcing the student to be enrolled in order to attend the exam and the labs. The `Student` class operations are further specified in OCL. Due to space limitation we only show the specification of the operation `enroll` and the predicate `isEnrolled` in figure 1(b). Note that the OCL specification includes frame conditions, such as `Student.allInstances()->forAll(s|s.passed=s.passed@pre)`, stating which attributes should remain unchanged when executing an operation. It is not consensual whether such frame-conditions must be specified, and some authors assume an implicit invariability assumption, stating that what is not mentioned in a post-condition should remain unchanged. However, such assumption may lead to ambiguities in post-condition interpretation [11], and we require

them to be explicitly specified, however, this step will be automated to release the user of this, potential, tedious task.

When PSMs are combined with UML static models such as CDs, both annotated with OCL, the V&V task substantially increases in complexity. Namely, it is no longer trivial to manually check consistency and detect specification errors, such as unreachable states. For instance, is it possible to ensure that every student has the opportunity to pass a course, by eventually reaching the `Passed` state? And upon reaching such state, are the `Student` attributes consistent, namely, is the course part of the `exams` association that stores the exams successfully completed by the student? We will show how an Alloy formalization of PSMs, CDs, and OCL, enables automatic verification of properties such as these using the Alloy Analyzer.

3 Preliminary Concepts

We present preliminary concepts related to PSMs and Alloy. Section 3.1 explains syntactic and semantics issues of PSMs. Section 3.2 introduces Alloy, and describes how UML models enriched with OCL can be formalized using an Alloy idiom tailored for dynamic specification.

3.1 UML Protocol State Machines

The abstract syntax of a PSM is shown in Figure 2. A PSM is modeled using a directed graph where the nodes represent *states* and the arrows *transitions* between states. A transition is an expression of the form `[precondition] operation / [postcondition]`. Pre- and post-conditions can be informally defined, however UML prescribes the use of OCL for their formal specification. State invariants can be associated to each state. A state invariant should be satisfied whenever the related state is active. There exist three kind of states: *simple*, *composite* or *submachine*. The first one is a state without sub-states (neither regions nor submachines), the second one can be composed of two or more orthogonal regions, and the third one allows the specification of inner state machine (submachines). For instance, the PSM in Figure 1(a) shows a composite state, `Studying`, with two regions which we will name them as `R1` and `R2`. A region may optionally have a final state and an initial pseudostate. Other examples of pseudostates are: *join*, *fork*, *junction* and *choice*.

A transition is enabled when its source state is active, the source state invariant holds and the pre-condition associated to its operation is true. Transitions are triggered by events which represent invocation of operations. When the same operation is referred by more than one transition, if it is invoked, different transitions will be enabled resulting in a *conflict*. The UML standard [16] prioritizes firing using the state hierarchy: transitions from deeper sub-states have higher priority over the ones from including composite states.

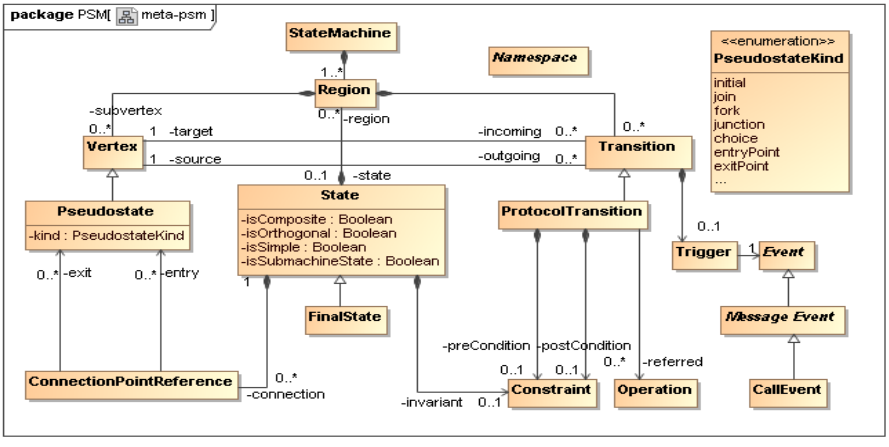


Fig. 2. Metamodel of UML Protocol State Machines

3.2 Alloy

Alloy [10] is a formal modeling language based on a relational flavor of first-order logic. Alloy is supported by the Alloy Analyzer, a SAT (satisfiability problem) based tool that enables automatic model V&V. Alloy Analyzer is inspired by model checkers, but it is implemented as a solver, performing verification within a bounded scope.

The abstract syntax of Alloy language is described in the metamodel presented in Figure 3. An Alloy module consists of a module header, a set of imports and zero or more paragraphs. The *module header* is a name of the module where paragraphs are defined. The *import* keyword specifies the inclusion of other modules. A *paragraph* can either be a signature declaration, a constraint, an assertion or a command.

A *signature declaration* denotes a set of atoms. An atom is a unity with three basic properties: it is indivisible, immutable and uninterpreted. Signature declarations can introduce *fields*. A field represents a relation among signatures. *Facts*, *predicates* and *functions* describe invariants, named constraints, and named expressions, respectively. The difference between a fact and a predicate is that the first one always holds while the second one only holds when invoked. *Assertions* allow the expression of properties that are expected to hold as consequence of specified facts. Finally, *commands* instruct the Alloy Analyzer to perform particular analysis using two possible instructions: *run* and *check*. The first checks model consistency by requesting a valid instance, and the latter verifies an assertion by searching for a counterexample. Both commands optionally define a scope, bounding the number of instances allowed for each signature.

Specifying OCL Annotated Class Diagrams in Alloy. Alloy's logic is quite generic and does not commit to a particular specification style [10]. There

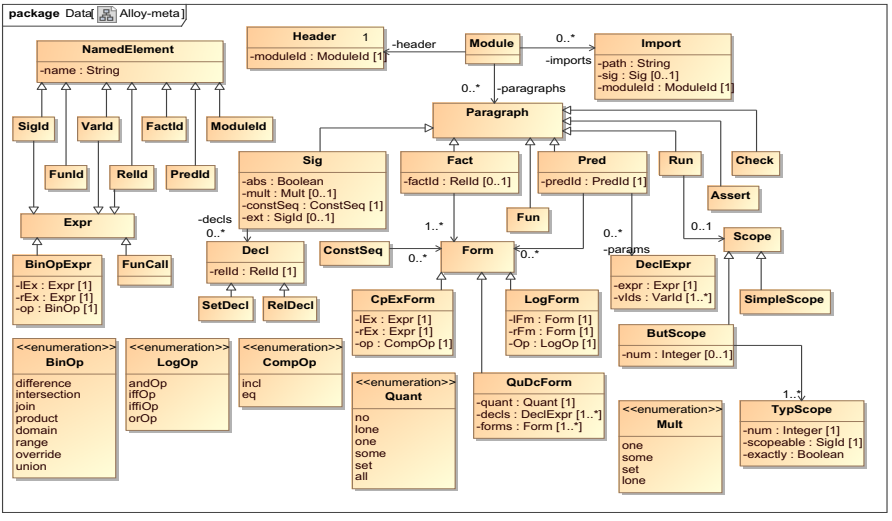


Fig. 3. Metamodel of Alloy

is also no predefined way to model dynamic behavior, since instances can only be populated with immutable atoms. A standard way to circumvent this is to introduce a signature denoting the overall state of the system, and model operations as predicates that specify the relationship between pre- and post-states. Two variants of this idiom are known respectively as *global state* and *local state*. In the first one, all mutable fields are defined in the global state signature. In the second one, an extra column at the end of each mutable field is added locally to represent the state signature (usually denoted `Time`). The local state is often simpler than other competing idioms for modeling the dynamics of complex systems [24] and well-suited for modeling state machines [10]. In this idiom, operations are modeled as predicates that specify the relationship between pre- and post-states. To be more specific, an operation `op` is specified using a predicate `pred op[...,t,t':Time] {...}` with two special parameters `t` and `t'` denoting, respectively, the pre- and post-state. Predicates of the form `pred q [...,t:Time] {...}` are used to specify boolean queries. A formal characterization of this idiom can be found in [8], together with a translation to UML CDs enriched with OCL.

Figure 4 shows how the OCL annotated CD of Figure 1(b) can be specified in Alloy using the local state idiom. The `passed`, `current`, `labs` and `exams` associations are modeled as mutable relations in `Student`. Cardinality constraints at association ends yield corresponding multiplicities in field declarations. For example, the keyword `lone` in the field `current` limits the cardinality of the set `Course` to zero or one instances, when it is associated to `Time`. OCL pre- and post-conditions in operations are modeled by relational expressions evaluated at state `t` and `t'`, respectively. In Alloy *everything is a relation*. Therefore, the relational composition operator can be used to various purposes. In particular,

```
module student
sig Time {}
sig Student {
  passed  : Course     -> Time, current : Course lone -> Time,
  labs    : Laboratory -> Time, exams   : Course -> Time      }
sig Course { labs : some Laboratory }
sig Laboratory {}
fact { labs in Course lone -> Laboratory}
pred enroll [s : Student, c : Course, t,t' : Time] {
     no s.current.t
     c not in s.passed.t
     current.t' = current.t + s -> c
     passed.t' = passed.t
     labs.t' = labs.t
     exams.t' = exams.t }
pred isEnrolled [s : Student, t : Time] { some s.current.t }
```

Fig. 4. Coursing example in Alloy

when `t` is composed with a mutable field, it denotes its value at the pre-state. For example, `s.current.t` denotes the course of a student `s` prior to method invocation. In Alloy there is no implicit `self` object, and an explicit self parameter must be included in the operation signatures. This explicit parameter must then be used whenever `self` is implicitly used. For example, the OCL precondition `not passed->includes(c)` of method `enroll`, stating that a student can enroll only in courses not yet passed, can be specified in Alloy as `c not in s.passed.t`. There are many challenging issues to address when implementing an automatic translation from OCL to Alloy, such as the translation of OCL pre- and post-conditions. These have been addressed but not implemented in [1]. We will use the same approach, as in [1], to translate CDs annotated with OCL to Alloy but considering dynamic issues. In particular, we will generate a specification conforming the local state idiom; namely, to translate OCL pre- and post-conditions to predicates and to include an extra column `Time` at the end of each mutable field. Following this approach, an Alloy model equivalent to the one of Figure 4 can be generated from the UML model of Figure 1(b).

4 Specifying Protocol State Machines in Alloy

We present an approach to specify PSMs in Alloy. Firstly, we specify how CD enriched with OCL (CD+OCL) can be integrated in order to be used by a PSM. Then, we describe the transformation of a PSM to Alloy and we show how to perform V&V tasks using the Alloy Analyzer. The proposal is explained using a case study. Finally, we formalize the transformation by defining the rules in ATL.

4.1 Importing UML Class Diagram into Alloy

Two separate Alloy modules will be used: one to specify the CD+OCL, and another to specify the PSM. The latter imports the former, since the transformation from PSMs to Alloy requires the specification of classes, attributes, relations and operations, corresponding to the CD+OCL, in the local state idiom. This separation of concerns allows us to directly reuse part of the output of UML2Alloy tool, and, if the user makes changes to the Alloy model, it is possible to translate it back to a CD+OCL using Alloy2OCL, another tool previously developed for this particular effect [8].

4.2 PSM's States and Transitions

PSM simple states can be modelled directly in Alloy using singleton signatures. On the other hand, composite states and regions can be modeled using abstract signatures, to be extended by the signatures modeling its sub-states. At the top of the state hierarchy we will have the signature `State` containing all states. The pseudostate `Initial` is also modeled similarly to simple states. Following these rules, the states of our running example, in figure 1(b), can be specified as follows.

```
abstract sig State {}
abstract sig Studying extends State {}
abstract sig R1 extends Studying {}
abstract sig R2 extends Studying {}
one sig Lab extends R1 {}
one sig NoExam, Exam extends R2 {}
one sig Initial, Enrolling, Passed, Quitted, Failed extends State {}
```

The PSM itself is modeled using a singleton signature `PSM`, with a single mutable relation `state`, that, for each time instant and instance of the associated class returns the set of active states.

```
one sig PSM { state: State some -> Student -> Time }
```

Similarly to operations, transition between normal states will be modeled by a predicate that, for each instance of the class associated with the PSM, relates the pre- and post-state. The metamodel of PSMs (see Figure 2) establishes that a protocol transition can refer to zero or more operations. To simplify the presentation, we will limit this set to at most one operation per transition. The transition predicate invokes the referred operation, whose predicate is defined in the imported model. If no operation is referred, the transition predicate invokes a special `nop` predicate, with frame-conditions that constrain all mutable fields to remain unchanged. Each transition predicate also includes two constraints to model the dynamics of the machine: one checks if all the source states are active in the pre-state for the given instance, and the other changes the relation `state`, so that its target states are active in the post-state. For example, transition `t3` of figure 1(a) can be modeled as follows.

```
pred t3 [s : Student, t,t' : Time] {
  NoExam in PSM.state.t.s
  PSM.state.t' = PSM.state.t - (NoExam -> s) + (Exam -> s)
  approve[s,t,t'] }
```

The relational expression `NoExam -> s` denotes the cartesian product of `NoExam` and `s`. Since these are singletons, in this case it denotes just a tuple. As such, the second constraint ensures that relation `state` has the same pre- and post-state for all student instances, except for `s`, which changes its state from `NoExam` to `Exam`.

Some transitions are not translated as predicates. In particular, this is the case of incoming transitions of join pseudostates and outgoing transitions of fork pseudostates. Their source and target states will be handled by the respective outgoing and incoming transitions. For instance, consider the fork pseudostate whose incoming transition is `t1`, with two outgoing transitions leading to the `Studying` composite state, respectively to `Lab` and `NoExam`. These states will be activated by the predicate modeling `t1`, which is defined as follows.

```
pred t1 [s: Student, t,t' : Time] {
  Enrolling in PSM.state.t.s
  PSM.state.t' = PSM.state.t - (Enrolling -> s) + ((Lab + NoExam) -> s)
  some c : Course {  (no s.current.t) && enroll[s,c,t,t']  } }
```

Notice the usage of an existential quantifier, `some`, to introduce the parameters of operation `enroll`, and the inclusion of the Alloy translation of the specified OCL pre-condition before its invocation.

State invariants are enforced using a fact for each state that declares them. For composite states, the invariant must hold whenever any of its sub-states is active. For example, the state invariant of `Studying` is specified as follows.

```
fact Studying {
 all t:Time, s:Student | some (PSM.state.t.s & Studying)=> isEnrolled[s,t]}
```

4.3 Finite Execution Traces

To model finite execution traces, a total order will be imposed on the `Time` signature using the predefined Alloy library `ordering`. This library defines useful relations to manipulate the total order, namely `first` to denote the first atom, and `next`, a binary relation that, given an atom, returns the following atom in the order.

The `next` relation must be restricted to relate only `Time` atoms for which a transition predicate holds for one of the instances of the class associated with the PSM. Moreover, at the `first` time atom all instances must be at the `Initial` pseudostate. Both these constraints are defined in the special fact `Traces`.

```
fact Traces {
  all s : Student | PSM.state.first.s = Initial
  all t : Time, t' : t.next | some s : Student {
    t0[s,t,t'] or t1[s,t,t'] or t2[s,t,t'] or t3[s,t,t'] or t4[s,t,t'] or
    t5[s,t,t'] or t6[s,t,t'] or t7[s,t,t'] or t8[s,t,t'] or t9[s,t,t'] }}
```

Firing Priority. Our running example does not contain conflicting transitions. If two transitions `high` and `low` could potentially be in conflict (that is, the same operation is invoked in both), and `high` has higher priority then `low`, then fact `Traces` would invoke them using

```
high[s,t,t'] or  (not high[s,t,t'] and low[s,t,t']).
```

4.4 Verification and Validation of UML Diagrams

With both the PSM and the CD+OCL specified in Alloy, we can now check their consistency by asking for an execution trace. This can be done with the command `run`, that instructs the Alloy Analyzer to look for a valid instance of the model.

For instance, consider the command `run {} for 2 but 1 Student, 15 Time`. The keyword `for` can be used to define a scope bounding the number of atoms allowed for each signature. The keyword `but` establishes an exception for the boundary defined by `for`. In this case, the number of `Student` atoms is limited to 1 and the number of `Time` atoms is extended to 15.

In this particular example, the `run` command returns a valid trace and thus the PSM is consistent with the OCL annotated CD. However, this notion of consistency is very basic and does not suffice in order to validate the models. A more reliable notion is to check that every state of the PSM is reachable. For example, is it possible for a student to pass at least one course? Again, using a `run` command, we can ask the analyzer to return a trace where a student reaches state `Passed`.

```
run {some t : Time, s : Student | Passed in PSM.state.t.s
} for 2 but 1 Student, 15 Time
```

In this case the analyzer cannot find a valid execution trace, meaning that state `Passed` is unreachable in 15 steps. Obviously, this means that there is some problem with one of the models. Looking back at the PSM of Figure 1(a) we can see that there is a problem with the pre-condition of transition `t2`, that requires the (to be completed) lab assignment to already been completed before. After inserting the missing `not`, changing that pre-condition to `(not self.labs->includes(l)) and self.current.labs->includes(l)` the analyzer returns a valid execution trace. Figure 5 presents 6 consecutive states of this trace moving the student from state `Enrolling` to `Passed`: the student is first approved in the exam and then completes the two mandatory lab assignments. Since reachability is a desirable property, we will define a rule transformation to generate similar runs for each simple state of the PSM.

There are other examples of V&V tasks that can be performed using the Alloy Analyzer. For example, we can check that, when a student is in the `Passed` state, the `exams` relation contains the `current` course, using the following command.

```
check {all t :Time, s : Student { Passed in PSM.state.t.s =>
            s.current.t in s.exams.t  } } for 10 but 1 Student, 30 Time
```

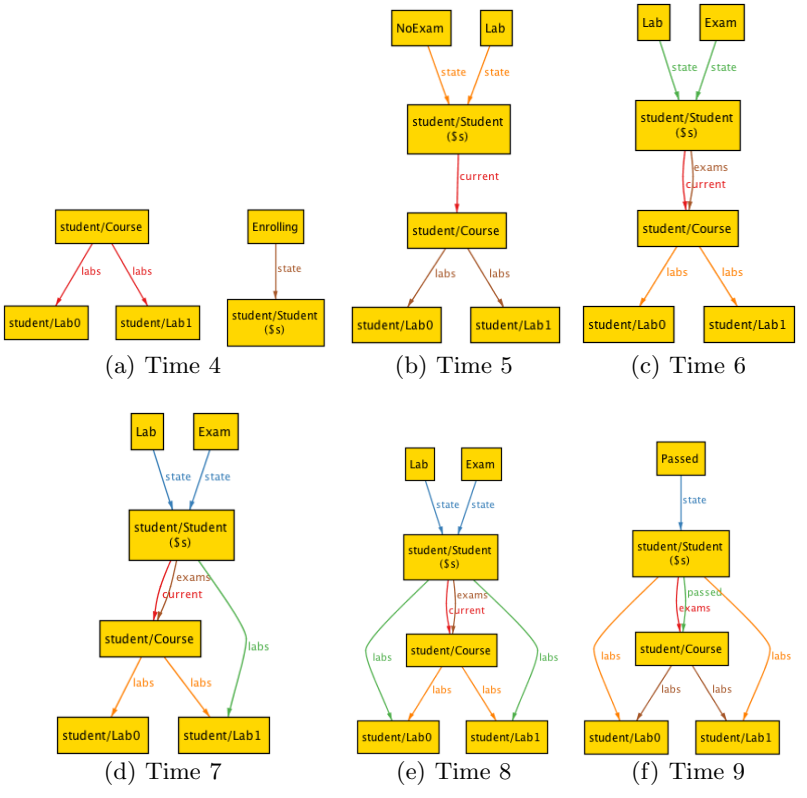


Fig. 5. Trace leading to a passed course

As previously mentioned, `check` verifies the assertion by searching for a counterexample. We specify a big scope in order to be confident that the assertion holds. In particular, we bound the number of atoms allowed for each signature to `10` but `30` for `Time`. Since no counterexamples are returned with such big scope, we can be more confident that this assertion holds.

4.5 Implementation

Our proposal was prototyped in a model transformation tool using the MDA approach: First, both the PSM and Alloy metamodels were specified, and then we defined a mapping from PSM elements to Alloy elements using the model transformation language ATL. Some of the ATL rules are presented in Figure 6.

Rule `Model2Module` maps a UML model of one PSM to an Alloy module, declaring the respective header and imports. Rule `PSM2Sig` creates the singleton signature `PSM` with the dynamic relation `state`. Rule `CompositeState2Sig` creates an abstract signature extending `State` for each PSM composite state. `SimpleState2Run` generate a run command for each simple state of the PSM. The ATL transformation is available for download at `http://sourceforge.net/p/psm2alloy`.

5 Related Work

Likewise other UML diagrams, the semantics of PSMs is quite ambiguous, and several attempts have been made to formalize it. For example, Bauer et al. [5] propose a model-theoretic semantics, based on labelled transition systems, for three different perspectives (namely, implementator's view, user's view, and interaction view) of the PSM. Here we will follow the user's view, that a PSM specifies the allowed call sequences on the classifiers operations.

The joint V&V of PSMs and other UML diagrams using traditional formal methods has also been proposed. For example, Lanoix et al. [12] use the B method to verify the interoperability and refinement of UML components, specified using CDs, sequence diagrams and PSMs. However, the focus is not on consistency and class methods are specified directly in B instead of the UML standard OCL. The consistency of an UML classes and the associated PSM has previously been addressed by Rash and Wehrheim [20], using a formalization to CSP. Again, classes are not specified with OCL, but using Object-Z. Lightweight formal methods have also been used for similar purposes before. In particular, Nimiya et al. [15] propose a method for verifying consistency of UML state machine diagrams and communication diagrams using Alloy, but it does not consider integration with CDs neither OCL. Ries [21] formalizes a subset of UML CDs and PSMs in Alloy, as part of a lightweight model-driven test selection process called SESAME, but it does not consider complex UML elements, such as composite states or fork and choice pseudostates, neither addresses the consistency of PSMs with CDs+OCL.

The relationship between CDs+OCL with Alloy has been extensively studied by Anastaskis et al. [2], resulting in a prototype model transformation tool named UML2Alloy that formalizes that relationship as a shallow embedding. Maoz et al [13] proposed a formalization of CDs using a deep embedding to Alloy, to support UML features not directly expressible in Alloy, such as multiple inheritance. However, both these formalizations yield Alloy specifications which are not well-suited to model dynamic behavior, namely by not making clear the distinction between pre- and post-states in method specification. Anastasakis [1] showed how UML2Alloy could be extended to solve that issue, but that extension was never incorporated into UML2Alloy. These formalizations did not consider PSMs, and thus left out some OCL features related to state machines, namely the OCL predefined operation *oclIsInState*, which evaluates whether an object is in a specific state.

UML has also been mapped to Alloy for model V&V of particular case-studies. We present three examples: the first one uses the Alloy Analyzer for formal security evaluation in a methodology called *Aspect-Oriented Risk-Driven Development* [9]; the second one describes a proposal for Alloy specification of Aspect-UML models, a UML Profile for extending UML with Aspect-oriented concepts [14]; the third one explains an approach to translate UML models, specified with OntoUML, for model validation using Alloy [6]. These examples, likewise [2] and [13], make evident Alloy potential for UML V&V, but they do not consider UML dynamic diagrams such as PSMs.

```
create OUT : MMAlloy from IN : MMUml;
rule Model2Module {
 from s  : MMUml!Model (
            MMUml!ProtocolStateMachine.allInstances()->size() =1 )
 to mId  : MMAlloy!ModuleId ( name  <- s.name    ),
    hd   : MMAlloy!Header   ( moduleId <- mId ),
    stId : MMAlloy!SigId    ( name  <- 'State'   ),
    st   : MMAlloy!Sig      ( abs   <- true, sigId <- stId ),
    m    : MMAlloy!Module   ( header<- hd,
             imports    <- MMAlloy!Import.allInstances(),
             paragraphs <- MMAlloy!Paragraph.allInstances() )
}
rule PSM2Sig {
 from s : MMUml!ProtocolStateMachine
 to sig : MMAlloy!SigId  ( name <- 'PSM' ),
    var : MMAlloy!VarId  ( name <- 'state' ),
   decl : MMAlloy!RelDecl(
            varIds <- var, mult <- #some, sigs <- getSigId('State')...),
    psm : MMAlloy!Sig (
            abs  <- false, mult <- #one, sigId <- sig, decls <- decl  )
}
rule CompositeState2Sig {
 from s  : MMUml!State  ( s.name <> '' and  s.isComposite() )
 to sigId: MMAlloy!SigId( name <- s.name        ),
   sigEx : MMAlloy!SigId( name <- s.getRegion() ),
   sig   : MMAlloy!Sig  ( abs <- true, sigId <- sigId, exts  <- sigEx )
}
rule SimpleState2Run {
 from s : MMUml!State ( s.name <> '' and  s.isSimple() )
 to
  v1  : MMAlloy!VarId ( name <- 't' ),
  v2  : MMAlloy!VarId ( name <- 's' ),
  dEx1: MMAlloy!DeclExpr (vId <- v1, sigId <- getSigId('Time')),
  dEx2: MMAlloy!DeclExpr (vId <- v2, sigId <- getSigCl() )),
  ex  : MMAlloy!BinOpExpr(op  <- #join, rEx <- ex2,lEx <- getSigId('PSM')),
  ex2 : MMAlloy!BinOpExpr(op  <- #join, rEx <- ex3,lEx <- getRel('state')),
  ex3 : MMAlloy!BinOpExpr(op  <- #join, rEx <- v1 , lEx <- v2 ),
  fm  : MMAlloy!CpExForm (op  <- #incl, lEx <- getSigId(s.name),rEx <- ex),
  qF  : MMAlloy!QuDcForm ( q  <- #some, decls<- Set{dEx1,dEx2},forms<- fm),
  tyT : MMAlloy!TypeScope( num <- '15',scopeable  <- getSigId('Time')),
  tyC : MMAlloy!TypeScope( num <- '1' ,scopeable  <- getSigIdClass()),
  sc  : MMAlloy!ButScope ( num <- '2', typeScopes <- Set{tyT,tyC}),
  run : MMAlloy!SimpleRun( form<- qF, scope <- sc ) }
```

Fig. 6. ATL rules to map a PSM to Alloy

6 Conclusions and Future Work

We have shown how both PSMs and CDs enriched with OCL can be formalized in Alloy, using the local state idiom to handle dynamics. This formalization enables us to perform automatic V&V of these UML diagrams using the Alloy Analyzer. In particular, it allows us to check they are consistent with each other, a fundamental property ignored by current UML tools. The proposed PSM formalization was prototyped using ATL. The proposed formalization of CDs+OCL could be implemented with a new version of UML2Alloy, to be (hopefully) released soon. The output (in Alloy) of this tool can be changed by the user (e.g., to correct ambiguities) and translated back into UML using the (previously developed) tool OCL2Alloy. This allows a smooth integration of Alloy in software development practices, namely allowing the use of the many available MDA tools on models which are verified and validated with Alloy.

The proposal could be scalable to other domains, such as safety-critical systems. So far, the formalization was only tested with small examples. We intent to validate it with larger case studies. Other ongoing work includes a (small) extension to Alloy to allow the specification of more complex behavioral properties in temporal logic (LTL). This will further simplify the V&V effort required by the user, by allowing him to reuse well-known temporal specification patterns [7]. In the future we also intend to use this formalization to automatically generate UML sequence diagrams, to be used in model based testing.

References

1. Anastasakis, K.: A Model Driven Approach for the Automated Analysis of UML Class Diagrams. Ph.D. thesis, University of Birmingham (2009)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2008)
3. de Andrade, F.R., Faria, J.P., Paiva, A.C.R.: Test generation from bounded algebraic specifications using alloy. In: ICSOFT (2), pp. 192–200 (2011)
4. ATLAS: ATLAS Transformation Language, LINA&INRIA (2009)
5. Bauer, S.S., Hennicker, R.: Views on Behaviour Protocols and Their Semantic Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 367–382. Springer, Heidelberg (2009)
6. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering 6(1-2), 55–63 (2010)
7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM (1999)
8. Garis, A., Cunha, A., Riesco, D.: Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 221–236. Springer, Heidelberg (2011)
9. Georg, G., Anastasakis, K., Bordbar, B., Houmb, S.H., Toahchoodee, I.R.M.: Verification and trade-off analysis of security properties in UML system models. IEEE Transactions on Software Engineering 36(3), 338–356 (2010)

10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
11. Kosiuczenko, P.: Specification of Invariability in OCL. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 676–691. Springer, Heidelberg (2006)
12. Lanoix, A., Souquières, J.: Trustworthy Assembly of Components using B Refinement. e-Informatica Software Engineering Journal (ISEJ) 2(1) (2008)
13. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011)
14. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling, pp. 41–48. ACM (2007)
15. Nimiya, A., Yokigawa, T., Miyazaki, H., Amasaki, S., Sato, Y., Hayase, M.: Model checking consistency of UML diagrams using Alloy. World Academy of Science, Engineering and Technology 71(99), 547–550 (2010)
16. OMG: UML Superstructure, Version 2.4.1 (2011)
17. OMG: Object Constraint Language, Version 2.3.1 (2012)
18. Paiva, A.C.R., Faria, J.C.P., Vidal, R.F.A.M.: Towards the integration of visual and formal models for GUI testing. Electronic Notes in Theoretical Computer Science 190(2), 99–111 (2007)
19. Porres, I., Rauf, I.: Generating class contracts from UML protocol statemachines. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa 2009. pp. 8:1–8:10. ACM (2009)
20. Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 229–243. Springer, Heidelberg (2003)
21. Ries, B.: SESAME: A Model-Driven Process for the Test Selection of Small-Size Safety- Related Embebbed Software. Ph.D. thesis, Université du Luxembourg (2009)
22. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML verification environment. In: Proceedings of the Software Engineering and Formal Methods, SEFM 2004. pp. 174–183 (2004)
23. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15, 92–122 (2006)
24. Taghdiri, M., Jackson, D.: A Lightweight Formal Analysis of a Multicast Key Management Scheme. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 240–256. Springer, Heidelberg (2003)

Patterns for a Log-Based Strengthening of Declarative Compliance Models

Dennis M.M. Schunselaar*, Fabrizio M. Maggi**, and Natalia Sidorova

Eindhoven University of Technology, The Netherlands
{d.m.m.schunselaar,f.m.maggi,n.sidorova}@tue.nl

Abstract. LTL-based declarative process models are very effective when modelling loosely structured processes or working in environments with a lot of variability. A process model is represented by a set of constraints that must be satisfied during the process execution. An important application of such models is compliance checking: a process model defines then the boundaries in which a system/organisation may work, and the actual behaviour of the system, recorded in an event log, can be checked on its compliance to the given model.

A compliance model is often a general one, e.g., applicable for a whole branch of industry, and some constraints used there may be irrelevant for a company in question: for example, a constraint related to property assessment regulations will be irrelevant for a rental agency that does not execute property assessment at all. In this paper, we take the compliance model and the information about past executions of the process instances registered in an event log and, by using a set of patterns, we check which constraints of the compliance model are irrelevant (vacuously satisfied) with respect to the event log. Our compliance patterns are inspired by vacuity detection techniques working on a single trace. However, here we take all the knowledge available in the log into consideration.

Keywords: Linear Temporal Logic, Declare, Vacuity detection, Compliance checking, Event log.

1 Introduction

While imperative process modelling languages such as BPMN, UML ADs, EPCs and BPEL are very useful when it is necessary to provide strong support to the process participants during the process execution, they are less appropriate for environments characterised by high flexibility and variability. In such cases, declarative process models are more effective than the imperative ones [1,14]. Instead of explicitly specifying all the possible sequences of activities in a process,

* This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

** This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 327–342, 2012.

declarative models implicitly specify the allowed behaviour of the process with constraints, i.e., rules that must be followed during execution. In comparison to imperative approaches, which produce "closed" models (what is not explicitly specified is forbidden), declarative languages are "open': everything that is not forbidden is allowed. In this way, models offer flexibility and still remain compact.

Recent works have showed that declarative languages based on LTL (Linear Temporal Logic) [16] can be fruitfully applied in the context of process discovery [7,11] and compliance checking [5,10,12]. In [13], the authors introduced a declarative process modelling language called *Declare* characterised by a user-friendly graphical representation and a formal semantics grounded in LTL. A *Declare* model is a set of *Declare* constraints, which are defined as instantiations of *Declare* templates. Templates are abstract entities that define parameterised classes of properties. Fig. 1 shows the representation of the *response* template $\Box(x \Rightarrow \Diamond y)$ in *Declare* and its possible instantiation in a process for renting apartments, where parameters x and y take the values *Plan final inspection* and *Execute final inspection*. This constraint means that every action *Plan final inspection* must eventually be followed by action *Execute final inspection*.

Since *Declare* models are focused on ruling out forbidden behaviour, *Declare* is very suitable for defining *compliance models* that are used for checking that the behaviour of a system complies certain regulations. The compliance model defines the rules related to a single instance of a process, and the expectation is that all the instances follow the model. For example, the constraint *after planning a final inspection, the inspection must be eventually executed* will be satisfied for a process instance trace if the activity "plan final inspection" is followed by "execute final inspection", or if "plan final inspection" does not happen at all. Note that this constraint is only informative for a company if "plan final inspection" can happen in some process instance; if it never happens, the constraint is still satisfied but in an uninteresting way, which is called *vacuous satisfaction*.

Vacuous satisfaction of a constraint might signal that the constraint from some reference compliance model is (1) irrelevant for a particular company (e.g., the company does not do final inspections at all), or (2) it might indicate some underspecification in the compliance model (e.g., the constraint saying that every process execution should contain the planning of the final inspection is missing). In case (1), the model is unnecessarily difficult for (new) company employees and they get inclined to disregard it as irrelevant. The company needs this constraint to be strengthened to the constraint saying that "plan final inspection" cannot occur. In case (2), the danger is even bigger, since the model does not capture the regulations correctly.

In this paper, we take the information about the executions of process instances captured in the event log (assuming that it is long enough [4] and thus captures enough information about the system behaviour), we check which constraints are vacuously satisfied on the log and we replace them with stronger constraints using our compliance patterns. We start from the existing results in the field of vacuity detection for single traces [2,6] and we extend the method

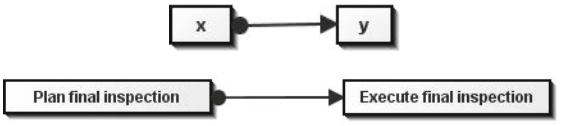


Fig. 1. Response template in *Declare* and its possible instantiation

of [6] in order to take into account the context of compliance checking where constraints are evaluated not just on single traces but on sets of traces coming from an event log.

We have evaluated our approach on an event log from a Dutch rental agency. Starting from an input compliance model defined by a domain expert we applied our approach and diagnosed the options for strengthening the compliance model so that the obtained model shows constraints relevant with respect to the log.

The remainder of the paper is structured as follows: we discuss related work in Sect. 2. In Sect. 3, we provide an informal introduction to the *Declare* language based on the *Declare* model for cancellation of a rental contract of an apartment rental company that we use as a running example. In Sect. 4, we propose compliance patterns for *Declare* constraints and in Sect. 5 we discuss our methodology for strengthening compliance models. In Sect. 6, we show an application of our approach to a real-life example. Section 7 concludes the paper.

2 Basic Concepts and Related Work

In Tab. 1, we briefly introduce the standard LTL operators and their (informal) semantics [16], which is used in *Declare* constraints.

In this paper, we start from the notions of *vacuity detection* and *interesting witness* first introduced in [2] for CTL* formulas. Since CTL* is a superset of LTL (in which we are interested in the context of *Declare*), we can apply these notions in our work. According to [2], a path π is an *interesting witness* for a formula φ if π satisfies φ non-vacuously, which means that every subformula ψ of φ affects the truth value of φ in π. In [2], the authors present an approach for vacuity detection for w-ACTL, a subset of Action Computational Tree Logic, which is, in turn, a subset of CTL. In [17], the authors present an approach for

Table 1. The LTL operators and their semantics

Operator	Semantics
$\bigcirc\varphi$	*φ holds in the next position of a path.*
$\Box\varphi$	*φ holds always in the subsequent positions of a path.*
$\Diamond\varphi$	*φ holds eventually (somewhere) in the subsequent positions of a path.*
$\varphi\mathcal{U}\psi$	*φ holds in a path at least until ψ holds. ψ must hold in the current or in a future position.*

vacuity detection in CTL formulas. They do not provide, however, an operative algorithm to be applied to LTL specifications.

In [6], the authors introduce an approach for vacuity detection in temporal model checking for LTL; they provide a method for extending an LTL formula φ to a new formula $witness(\varphi)$ that, when satisfied, ensures that the original formula φ is non-vacuously true. In particular, $witness(\varphi)$ is generated by considering that a path π satisfies φ non-vacuously (and then is an interesting witness for φ), if π satisfies φ and π satisfies a set of additional conditions that guarantee that every subformula of φ does really affect the truth value of φ in π. These conditions correspond to the formulas $\neg\varphi[\psi \leftarrow \bot]$ where, for all the subformulas ψ of φ, $\varphi[\psi \leftarrow \bot]$ is obtained from φ by replacing ψ by false or true, depending on whether ψ is in the scope of an even or an odd number of negations. Then, $witness(\varphi)$ is the conjunction of φ and all the formulas $\neg\varphi[\psi \leftarrow \bot]$ with ψ being a subformula of φ:

$$witness(\varphi) = \varphi \wedge \bigwedge \neg\varphi[\psi \leftarrow \bot]. \tag{1}$$

This approach was applied to *Declare* in [11] for vacuity detection in the context of process discovery. However, the algorithm introduced in [6] can generate different results for equivalent LTL formulas. Consider, for instance, the following equivalent formulas (corresponding to a *Declare alternate response* constraint):

$$\varphi = \Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc((\neg a \mathcal{U} b) \vee \Box(\neg b))) \text{ , and}$$

$$\varphi' = \Box(a \Rightarrow \bigcirc(\neg a \mathcal{U} b)).$$

When we apply (1) to φ and φ', we obtain that $witness(\varphi) \neq witness(\varphi')$:

$$witness(\varphi) = false,$$

$$witness(\varphi') = \varphi' \wedge \Diamond(\neg \bigcirc (\neg a \mathcal{U} b)) \wedge \Diamond(a) \wedge \Diamond(a \wedge \neg \bigcirc (b)).$$

In compliance models, LTL-based declarative languages like *Declare* are used to describe requirements to the process behaviour. In this case, each LTL rule describes a specific constraint with clear semantics. Therefore, we need a *univocal* (i.e., not sensitive to syntax) and intuitive way to diagnose vacuously compliant behaviour in an LTL-based process model.

Another issue in the approach proposed by [6] is that for two LTL formulas f and g, the composite formula $\varphi = f \vee g$ is *never non-vacuously true*. This is definitely counterintuitive, because one would expect that φ is non-vacuously true if f is non-vacuously true or g is non-vacuously true.

Furthermore, the notion of vacuous satisfaction, as introduced in [2,6], is designed for formulas that hold *on a given trace in an uninteresting way*. However, when applying (1) in the context of a log (a set of traces), we obtain, in some cases, conditions for vacuity detection that are too strong and difficult to satisfy.

For instance, if we apply (1) to $\varphi = \Box(\textit{Agree on self made changes?} \Rightarrow \Diamond(\textit{Plan final inspection} \vee \textit{Adjust floor plan}))$, we obtain that $witness(\varphi)$ is:

$$\varphi \wedge \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Plan final inspection})) \wedge \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Adjust floor plan})).$$

This formula is too strong in the context of a log, since we will not have in every trace *Agree on self made changes?* followed by both *Plan final inspection* and *Adjust floor plan*. In our approach, we will "weaken" this condition by requiring that each term of the conjunction must be valid, separately, on different traces. In addition, the original formula must be also always valid. This yields the following two formulas:

$$\varphi \wedge \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Plan final inspection})), \text{ and}$$
$$\varphi \wedge \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Adjust floor plan}))$$

each of which is expected to hold independently on some trace of the log to justify that the original formula is non-vacuously satisfied in the log.

3 Declare

Declare is characterised by a user-friendly graphical front-end and is based on a formal LTL back-end. These characteristics are crucial for two reasons. First of all, *Declare* is understandable for end-users and suitable to be used by stakeholders with different backgrounds. For instance, *Declare* has been already effectively applied in a project for maritime safety and security [9] where several project members did not have any formal background. Secondly, *Declare* has a formal semantics and *Declare* models are verifiable. This characteristic is important for the implementation of tools to check the compliance of process behaviour to *Declare* models (see, e.g., [10]).

A *Declare* model consists of a set of constraints which, in turn, are based on templates. Templates are parameterised classes of properties (a superset of the ones defined by Dwyer et al. in [3]) equipped with a graphical representation and a semantics specified through LTL formulas. Each *Declare* constraint inherits the graphical representation and semantics from its template. When a template is instantiated in a constraint, a template parameter is replaced by one or several activities. When two or more activities are used for one parameter, we say that this parameter branches and it is then substituted by a disjunction of branched activities in the LTL formula.

Declare constraints can be subdivided into four groups: *existence* (i.e., *existence*, *absence*, *exactly* and *init*), *relation* (i.e., *responded existence*, *co-existence*, *response*, *precedence*, *succession*, *alternate response*, *alternate precedence*, *alternate succession*, *chain response*, *chain precedence* and *chain succession*), *negative relation* (i.e., *not co-existence*, *not succession* and *not chain succession*), and *choice* (i.e., *choice* and *exclusive choice*). The LTL semantics for existence and relation constraints are listed in Tab. 2, Tab. 3 and Tab. 4 (first line for each constraint). For the full overview of the language we refer the reader to [13,15].

Fig. 2. An example of a *Declare* model

Fig. 2 shows a *Declare* model that describes a process for cancellation of a rental contract at a rental agency, which we use to explain the main characteristics of the language. The process in Fig. 2 involves five activities, depicted as rectangles (e.g., *Plan final inspection*), and three constraints, showed as connectors between the activities (e.g., *not succession*). In our example, prior to agreeing to any changes made by the tenant, the company must create a rental cancellation form. This form will be used in the activity *Agree on self made changes?* to specify whether the company agrees or disagrees with the self-made changes. This is indicated by the *precedence* constraint. After agreeing or disagreeing with the self-made changes, the company either plans a final inspection (to determine whether the tenant has reverted or mended her self-made changes), or adjusts the floor plan to reflect the current situation after the self-made changes. If they partially agree on the changes made by the tenant, it is possible to adjust the floor plan *and* plan a final inspection. All this is captured in the branched *response* constraint. Finally, the company cannot plan a final inspection after having created (and sent) a confirmation letter (stating that no problem was encountered), as indicated by the *not succession* constraint.

The response constraint in Fig. 2 is an example of a branched *response* constraint $\Box(x \Rightarrow \Diamond y)$, where parameter x is replaced by *Agree on self made changes?* and parameter y is branched on *Plan final inspection* and *Adjust floor plan*. This means that if *Agree on self made changes?* occurs in a trace, it must be eventually followed by *Plan final inspection* or *Adjust floor plan*, captured in LTL as $\Box(\mathit{Agree\ on\ self\ made\ changes} \Rightarrow \Diamond(\mathit{Plan\ final\ inspection} \vee \mathit{Adjust\ floor\ plan}))$.

The other two constraints in our renting agency model, are captured in LTL as $\Box(\mathit{Create\ confirmation\ letter} \Rightarrow (\neg\Diamond \mathit{Plan\ final\ inspection}))$ (*not succession*), and $(\neg \mathit{Agree\ on\ self\ made\ changes?}\ \mathcal{U}\ \mathit{Create\ rental\ cancellation\ form}) \vee \Box(\neg \mathit{Agree\ on\ self\ made\ changes?})$ (*precedence*).

The semantics of the whole model is determined by the conjunction of these formulas. Note that, when operating on business processes, we reason on finite traces. Therefore, we assume that the semantics of the *Declare* constraints is expressed in FLTL [8], a variant of LTL for finite traces.

4 Approach

In the remainder of the paper, we write $\mathcal{A}$ for the disjunction over the activities in $A = \{a_1, \cdots, a_n\}$, i.e., $\mathcal{A} = \bigvee_{a \in A} a$. Similarly, we write $\mathcal{B}$ for the disjunction $\bigvee_{b \in B} b$ over the activities in $B = \{b_1, \cdots, b_m\}$. We write W for an event log, $t \in W$ for a trace in W and we assume the activities to be atomic.

Similarly to the notion for vacuity detection captured by (1), we define a vacuity detection condition as follows:

Definition 1. *Given a (branched) Declare constraint φ, a* vacuity detection condition *of φ is a formula $\neg\varphi[\psi \leftarrow \bot]$ with ψ being a subformula of φ.*

In the context of compliance checking, we do not reason in terms of single traces but in terms of event logs that are composed of multiple traces. Therefore, as explained in Sect. 2, when applying (1) to a branched *Declare* constraint, instead of verifying the conjunction of all the vacuity detection conditions on every single trace, we adopt a more "permissive" approach. In particular, we require that for each vacuity detection condition, there exists a trace on which the condition holds.

According to [6], the algorithm described by (1) can be applied in a user-guided mode by limiting the evaluation of $witness(\varphi)$ only to a subset of vacuity detection conditions. We choose these subsets differently for different *Declare* constraints by considering the vacuity detection conditions that give significant results from the point of view of each specific constraint. We have to use the user-guided mode because of the problems mentioned in Sect. 2.

As final output of our approach we want to obtain from a given compliance model and a log a more restrictive compliance model, where strengthening of the constraints is defined by the results of the vacuity check. The vacuity check can be done by applying a set of *compliance patterns*. First, through a compliance pattern, we check whether φ is satisfied everywhere in the log. Second, we check whether, for each vacuity detection condition $\neg\varphi[\psi \leftarrow \bot]$, there is a trace of the considered log where the condition is satisfied. Third, we check, for each constraint of the original model, whether a stronger constraint holds non-vacuously on every trace of the log. For this purpose, in Fig. 3, we define a hierarchy of *Declare* constraints where an arrow from a node x to a node y means that x implies y (x is stronger than y). The names of the constraints suggest their meaning and their semantics is defined later in this section. Note that RE^{-1} is used to denote that the sets of activities has been swapped. Therefore, from the hierarchy, *responded existence*(B, A) is weaker than *precedence*(A, B).

Based on these observations, we can define the compliance pattern of a *Declare* constraint:

Definition 2. *Given a log W and a (branched) Declare constraint φ, the* compliance pattern *of φ in W is a set composed of three conditions:*

1. *φ holds on every trace of W;*
2. *for each element of a selection of vacuity detection conditions of φ, there is a trace in W on which this element holds (user-guided application of (1));*
3. *no stronger constraint holds non-vacuously in W.*

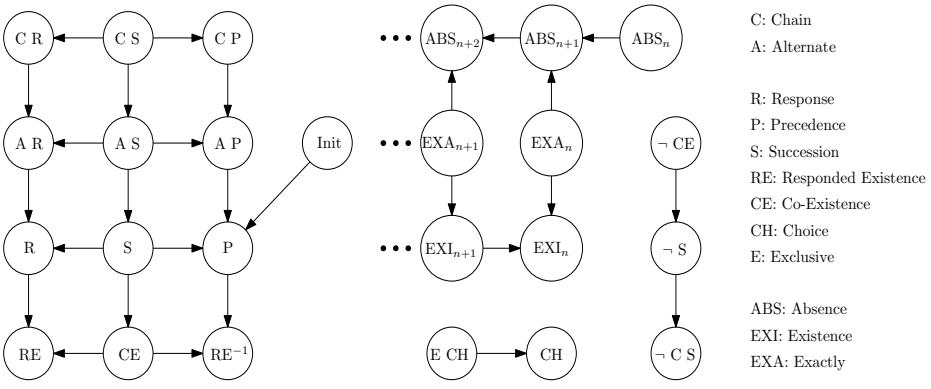


Fig. 3. The hierarchy of the *Declare* constraints

Furthermore, we define the notion of strong compliance as follows:

Definition 3. *Given a log W and a Declare constraint φ, W is* strongly compliant *to φ if all the conditions of the corresponding compliance pattern of φ are satisfied in conjunction on W.*

In the remaining subsections, we describe the compliance patterns of the *existence* and *relation* constraints. For the *negative relation* and *choice* constraints the application of the compliance patterns can be reduced to the evaluation of items 1 and 3 of Def. 2.

4.1 Existence Constraints

The compliance patterns for the existence constraints are listed in Tab. 2. For each constraint, the first line of the pattern shows the original LTL semantics that must hold for every trace in the log. For the *init* constraints, the additional condition is obtained by applying the approach for vacuity detection (11). When applying (11) on the $existence(nr, A)$, we replace $a \in A$ by false. Then, we obtain $\neg existence(nr, A[a \leftarrow false]) \equiv absence(nr, A \setminus \{a\})$.

Moreover, we want to ensure that in all these cases a stronger constraint does not hold (these conditions are not shown in the table for the sake of readability: they can be derived from the hierarchy in Fig. 3).

4.2 Relation Constraints

Our compliance patterns for the relation constraints are listed in Tab. 3 and Tab. 4. The conditions

$$\forall a \in A\ \exists t \in W : t \models \Diamond a, \text{ and } \forall b \in B\ \exists t \in W : t \models \Diamond b$$

must always be satisfied and we omit them in the tables. We also omit the conditions defining for each constraint that no stronger constraint holds non-vacuously in the log: they can be directly derived from the hierarchy in Fig. 3.

Table 2. Compliance patterns for existence constraints

Constraint	Pattern
$existence(1,\ A)$	$\forall t \in W : t \models \Diamond(\mathcal{A})$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$
	$\forall a \in A : \exists t \in W : t \models absence(1, A \setminus \{a\})$
$existence(nr,\ A)$	$\forall t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(existence(nr-1, A)))$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$
	$\forall a \in A : \exists t \in W : t \models absence(nr, A \setminus \{a\})$
$absence(nr,\ A)$	$\forall t \in W : t \models \neg existence(nr, A)$
$exactly(nr,\ A)$	$\forall t \in W : t \models existence(nr, A) \wedge$ $absence(nr+1, A)$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$
$init(A)$	$\forall t \in W : t \models \mathcal{A}$
	$\forall a \in A : \exists t \in W : t \models a$

Table 3. Compliance patterns for relation constraints without order

Constraint	Pattern
$responded\ existence(A, B)$	$\forall t \in W : t \models \Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B})$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A}) \wedge \Diamond(b)$
$co\text{-}existence(A, B)$	$\forall t \in W : t \models responded\ existence(A, B) \wedge$ $responded\ existence(B, A)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A}) \wedge \Diamond(b)$
	$\forall a \in A : \exists t \in W : t \models \Diamond(\mathcal{B}) \wedge \Diamond(a)$

For each constraint, the first line of the pattern shows the original LTL semantics that must hold on every trace of the log. The additional conditions are obtained by applying (1) to the original semantics. Due to space restrictions we will only elaborate on the deduction of some compliance patterns.

Responded Existence. Applying (1) to $responded\ existence(A, B)$, we replace $b \in B$ by false in the LTL formula $\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B})$. We obtain $\neg(\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B}[b \leftarrow false]))$. This formula is equivalent to $\Diamond(\mathcal{A}) \wedge \neg\Diamond(\mathcal{B}[b \leftarrow false])$. Combining this formula with $responded\ existence(A, B)$ yields $\Diamond(\mathcal{A}) \wedge \Diamond(b)$. The condition of the compliance pattern of $responded\ existence(A, B)$ is the combination of the conditions we obtain by replacing each $b \in B$ by false in the original formula.

Response. When we replace $b \in B$ in the LTL formula of $response(A, B)$ by false, we obtain $\neg\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}[b \leftarrow false]))$. This is equivalent to $\Diamond(\mathcal{A} \wedge \neg\Diamond(\mathcal{B}[b \leftarrow false]))$. Considering that the original formula must be true, we can conclude that every $a \in A$ is not followed by any $b' \in B \setminus \{b\}$ is equivalent to every $a \in A$ is followed by b. This implies that $\Diamond(\mathcal{A} \wedge \Diamond(b))$. When we replace every $b \in B$ by false in the original formula, we have the condition of the pattern for $response(A, B)$.

If we apply this pattern, for example, to *response* ({*Agree on self made changes?*}, {*Plan final inspection*, *Adjust floor plan*}), we obtain the following set of conditions that need to hold in the log:

Table 4. Compliance patterns for relation constraints with order

Constraint	Pattern
$response(A, B)$	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}))$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
$precedence(A, B)$	$\forall t \in W : t \models \neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
	$\exists t \in W : t \models \neg init(A)$
$succession(A, B)$	$\forall t \in W : t \models response(A, B) \wedge precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
	$\exists t \in W : t \models \neg init(A)$
$alternate\ response(A, B)$	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B})) \wedge \Box(\mathcal{A} \Rightarrow \bigcirc(\neg\mathcal{A}\mathcal{U}\mathcal{B}))$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
$alternate\ precedence(A, B)$	$\forall t \in W : t \models (\neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})) \wedge$ $\Box(\mathcal{B} \Rightarrow \bigcirc(\neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
$alternate\ succession(A, B)$	$\forall t \in W : t \models alt.\ response(A, B) \wedge alt.\ precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
$chain\ response(A, B)$	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \bigcirc\mathcal{B})$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(b))$
$chain\ precedence(A, B)$	$\forall t \in W : t \models \Box(\bigcirc\mathcal{B} \Rightarrow \mathcal{A})$
	$\forall a \in A : \exists t \in W : t \models \Diamond(\bigcirc(\mathcal{B}) \wedge a)$
$chain\ succession(A, B)$	$\forall t \in W : t \models ch.\ response(A, B) \wedge ch.\ precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(b))$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a \wedge \bigcirc(\mathcal{B}))$

$\forall t \in W : t \models \Box(Agree\ on\ self\ made\ changes? \Rightarrow$
$\Diamond(Plan\ final\ inspection \vee Adjust\ floor\ plan))$;
$\exists t \in W : t \models \Diamond(Agree\ on\ self\ made\ changes? \wedge \Diamond(Plan\ final\ inspection))$;
$\exists t \in W : t \models \Diamond(Agree\ on\ self\ made\ changes? \wedge \Diamond(Adjust\ floor\ plan))$.
Also, every constraint stronger than $response(\{Agree\ on\ self\ made\ changes?\}, \{Plan\ final\ inspection, Adjust\ floor\ plan\})$ must not hold non-vacuously in the log.

Precedence. If we apply (1) to the LTL formula of $precedence(A, B)$ and replace $a \in A$ by false, we obtain the condition $\neg((\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow false])) \vee \Box(\neg\mathcal{B}))$ that is equivalent to $\neg(\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow false])) \wedge \neg\Box(\neg\mathcal{B}))$. Similarly to the $response(A, B)$, given that the original LTL formula holds, we have $(\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$. When we replace every $b \in B$ in the original formula by true, we obtain the condition $\neg(false\mathcal{U}\mathcal{A}) \vee \Box(false)$. This is equivalent to $\neg\mathcal{A}$, which means that there exists a trace where no $a \in A$ occurs at the first position.

Chain Response. Applying (1) to $chain\ response(A, B)$, we replace in $\Box(\mathcal{A} \Rightarrow \bigcirc(\mathcal{B}))$ each $b \in B$ by false. We have $\neg\Box(\mathcal{A} \Rightarrow \bigcirc(\mathcal{B}[b \leftarrow false]))$ that is equivalent to $\Diamond(\mathcal{A} \wedge \neg \bigcirc (\mathcal{B}[b \leftarrow false]))$. Combining this formula with the original formula yields the condition $\Diamond(\mathcal{A} \wedge \bigcirc(b))$.

Take, for example, the constraint *chain response*({*Plan final inspection*}, {*Execute final inspection*, *Cancel final inspection*}). Applying this compliance pattern, we have:

$\forall t \in W : t \models \Box(\textit{Plan final inspection} \Rightarrow$
$\bigcirc(\textit{Execute final inspection} \vee \textit{Cancel final inspection}))$;
$\exists t \in W : t \models \Diamond(\textit{Plan final inspection} \wedge \bigcirc(\textit{Execute final inspection}))$;
$\exists t \in W : t \models \Diamond(\textit{Plan final inspection} \wedge \bigcirc(\textit{Cancel final inspection}))$.

Moreover, *chain succession*({*Plan final inspection*}, {*Execute final inspection*, *Cancel final inspection*}) and *absence*(1, {*Plan final inspection*}) must not hold non-vacuously in the log.

5 Methodology

We present now a methodology for the application of the patterns from Sect. 4 in order to transform an existing compliance model into a strongly compliant one. We assume that the constraints of the existing model do hold on the log. Moreover, we check, for each activity in the model, whether it is present in the log. If not, we explicitly introduce an absence constraint on it.

Algorithm 1 lists the steps we execute to obtain a strongly compliant model M_{output} starting from a given compliant model M_{input}. We take a top-down approach, i.e., we start with the strongest constraints being candidates for strengthening and, according to the hierarchy defined in Fig. 3, we weaken them until we find a set of non-vacuously satisfied constraints (possibly the original constraint). When strengthening a constraint, we immediately remove branching on activities that do not occur in the log. For instance, if our model contains the constraint $\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}))$ and a $b \in B$ does not occur in the log, we strengthen the constraint to $\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}'))$ (where $B' = B \setminus \{b\}$). It cannot be the case that no $b \in B$ occurs in the log since this would mean that the constraint does not hold in the log. If b does not occur in the log, we add $\textit{absence}(1, b)$ to the output model.

The algorithm relies on the following notion of a composed constraint:

Definition 4. *A* composed constraint φ *is a constraint that can be obtained by the conjunction of some other constraints, which we call components of* φ.

In Algorithm 1, we write C for the set of the components of constraint c; $C = \{c\}$ if c is not composed. For instance, for $c = \textit{chain succession}(A, B)$ we have $C = \{\textit{chain response}(A, B), \textit{chain precedence}(A, B)\}$, for $c = \textit{co-existence}(A, B)$ we have $C = \{\textit{responded existence}(A, B), \textit{responded existence}(B, A)\}$, and for $\varphi = \textit{alternate response}(A, B)$, $C = \{\textit{alternate response}(A, B)\}$.

For each constraint in the model, we first check whether it is the case that the constraint is of the type $\textit{precedence}(A, B)$ and $\textit{init}(A)$ holds non-vacuously. If so, we add $\textit{init}(A)$ to the output model. If the constraint is not of the type $\textit{precedence}(A, B)$ or $\textit{init}(A)$ does not hold non-vacuously, we check whether (a)

Algorithm 1. Transforming a compliant model to strongly compliant

Input: M_{input} a model, W a log
Output: M_{output} a strongly compliant model
(1) $M'_{input} \leftarrow$ an empty model
(2) **foreach** Constraint c in M_{input}
(3) substitute c by the strongest constraint w.r.t. the hierarchy and add it to M'_{input}
(4) **while** M'_{input} is not empty
(5) $M_{temp} \leftarrow$ an empty model
(6) **foreach** constraint c in M'_{input}
(7) **if** c is *precedence(A, B)* and *init(A)* holds non-vacuously on W **then**
(8) add *init(A)* to M_{output}
(9) **else**
(10) **if** *precedence(A, B)* $\in C$ and *init(A)* holds non-vacuously on W **then**
(11) $c_p \leftarrow$ *precedence(A, B)*
(12) add *init(A)* to M_{output}
(13) add all $c' \in C \setminus \{c_p\}$ which hold non-vacuously on W to M_{output} and add the remaining constraints in $C \setminus \{c_p\}$ to M_{temp}
(14) **else if** each $c' \in C$ holds non-vacuously on W **then**
(15) add c to M_{output}
(16) **else if** some $c' \in C$ hold non-vacuously **then**
(17) add all $c' \in C$ which hold non-vacuously on W to M_{output} and add the remaining constraints in C to M_{temp}
(18) **else if** no $c' \in C$ holds non-vacuously on W **then**
(19) substitute c by its immediate weaker notion and add it to M_{temp}
(20) replace M'_{input} by M_{temp}
(21) **return** M_{output}

precedence(A, B) is in C and *init*(A) holds non-vacuously, or (b) all constraints in C hold non-vacuously, or (c) a subset of the constraints in C holds non-vacuously, or (d) all constraints in C do not hold non-vacuously.

In the first case, we add the *init* and all non-vacuously satisfied constraints from C to the output model. The remaining constraints of C are added to the temporary model M_{temp} to be processed in the next iteration. In the second case, we add the constraint to the output model since in this case the constraint holds non-vacuously. In the third case, we add the subset of non-vacuously satisfied constraints in C to the output model and we add the remaining constraints in C to the temporary model to be processed in the next iteration. Consider, for instance, an *alternate succession* constraint where only the component *alternate response* is non-vacuously satisfied. In this case, we add the *alternate response* component to M_{output} and we keep the *alternate precedence* for future iterations.

In the fourth case, we add one of the weaker constraints (following the hierarchy defined in Fig. 3) to the temporary model.

When we check whether a constraint holds non-vacuously, we also remove vacuously satisfied branches of the constraint.

6 Case Study

We now present a small case study provided by a Dutch apartment rental agency in the form of an event log, recording process executions of a process for the cancellation of the rental contract by a tenant, and a compliance model defined by their domain expert (showed in Fig. 4). When the tenant gives a notice, the rental agency has to perform inspections to determine that the apartment is in a proper state. Based on these inspections, further actions might be needed.

Given an input model and a log, the question we want to pose is: *Does this compliance model correctly reflect the behaviour of the process represented in the log, assuming that the behaviour complies the model?* Note that we only want to facilitate the answering of this question for the domain expert. The final answer is up to the user, who can decide in which parts the strongly compliant model that our approach generates provides her with relevant information.

Starting from the model in Fig. 4, we apply the methodology from Sect. 5. First, we verify whether each activity in the model occurs at least once in the log. The activity *Create rental cancellation* never occurs in the log, so we add the constraint *absence*$(1, 7)$ to the output model (labelled with "0" in Fig. 5). Moreover, we replace all constraints by the strongest constraints with respect to the hierarchy introduced in Fig. 3. In particular, we replace the *precedence* constraint and all the *response* constraints by *chain succession* constraints. Moreover, we replace *not succession* by *not co-existence*.

After that, for all constraints, we check which of them hold non-vacuously on the log. To do this we use the LTL Checker plug-in of ProM[1]. The LTL Checker allows us to verify the validity of an LTL formula on a log. We use it to verify the validity of the conditions of a compliance pattern.

The *not co-existence* constraint holds on the log. This is enough to add this constraint to our output model. Indeed, this constraint is always non-vacuously true and there is no constraint stronger than *not co-existence*. All *chain succession* constraints added after the previous step do not hold: both the *chain response* component and the *chain precedence* component of each *chain succession* constraint do not hold. Therefore, we replace the *chain succession* constraints by *alternate succession* constraints. None of the *alternate succession* constraints hold, i.e., both the *alternate response* component and the *alternate precedence* component of each *alternate succession* constraint do not hold.

We replace then all *alternate succession* constraints by *succession* constraints. The *succession* constraints are also composed constraints. If we first consider *succession*$(3, \{4, 5, 6\})$, we need to have that *response*$(3, \{4, 5, 6\})$ and *precedence*$(3, \{4, 5, 6\})$ must hold non-vacuously. In the log, *response*$(3, \{4, 5, 6\})$

[1] `www.processmining.org`

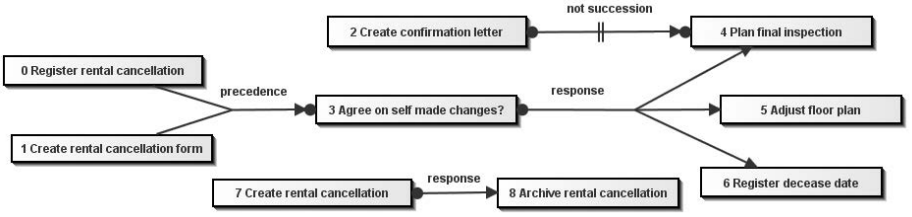


Fig. 4. Input model

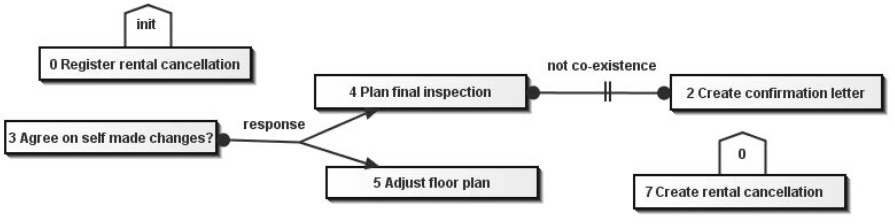


Fig. 5. Output model

Table 5. The compliance of the different conditions on the log

Constraint	Compliance pattern conditions	Valid
precedence({0, 1}, 2)	$\forall t \in W : t \models \neg 2\mathcal{U}(0 \vee 1) \vee \Box(\neg 2)$	✓
	$\exists t \in W : t \models \neg 2\mathcal{U}0 \wedge \Diamond(2)$	✓
	$\exists t \in W : t \models \neg 2\mathcal{U}1 \wedge \Diamond(2)$	✓
	$\exists t \in W : t \models \neg \mathit{init}(\{0, 1\})$	✗
	a stronger constraint should not hold non-vacuously	✓
not succession(2, 4)	$\forall t \in W : t \models \Box(2 \Rightarrow \neg\Diamond(4))$	✓
	$\neg\forall t \in W : t \models \neg(\Diamond 2 \wedge \Diamond 4)$	✗
response(3, {4, 5, 6})	$\forall t \in W : t \models \Box(3 \Rightarrow \Diamond(4 \vee 5 \vee 6))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(4))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(5))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(6))$	✗
	a stronger constraint should not hold non-vacuously	✓
response(7, 8)	$\forall t \in W : t \models \Box(7 \Rightarrow \Diamond(8))$	✓
	$\exists t \in W : t \models \Diamond(7 \wedge \Diamond(8))$	✗
	a stronger constraint should not hold non-vacuously	✗

holds non-vacuously if we remove the branch on activity 6, so we add *response*(3, {4, 5}) to the output model. Moreover, *precedence*(3, {4, 5, 6}) does not hold, so we add this constraint to the temporary model to verify it in the next iteration. We also split *succession*({0, 1}, 3) into *response*({0, 1}, 3) and *precedence*({0, 1}, 3). We have that the *init*(0) holds non-vacuously, so we add *init*(0) to the

output model. Moreover, *response*({0, 1}, 3) does not hold and we add it to the temporary model.

We have now two constraints we want to verify: *precedence*(3, {4, 5, 6}) and *response*({0, 1}, 3). Both do not hold in our log. *precedence*(3, {4, 5, 6}) is weakened to a *responded existence*({4, 5, 6}, 3). *response*({0, 1}, 3) is weakened to a *responded existence*({0, 1}, 3) and both are checked. Both *responed existence* constraints do not and are removed from the model. The strongly compliant model we obtain is depicted in Fig. 5. Here, all constraints hold non-vacuously.

All the constraints from the input model and their compliance patterns are listed in Tab. 5. For each pattern we have indicated whether every single condition is valid on the log or not. The table shows that for each constraint in the original model a part of the pattern is not valid on the log. Based on the results in Tab. 5, each constraint of the original model must be modified to obtain the strongly compliant model depicted in Fig. 5. The advantages of the output model with respect to the input model are: (1) precision, the output model describes reality better than the input model, and (2) understandability, one only has to understand the relevant parts.

7 Conclusion

In this paper, we describe compliance patterns for strengthening constraints in compliance models specified in *Declare* in order to show which part of the behaviour is actually covered by the process executions recorded in the event log of the system, and which (parts of) constraints are vacuously satisfied. This approach can be used for configuring reference models towards the needs of a company. We have shown in the case study how we make use of our constraints hierarchy to achieve the best results.

Our approach can easily be extended for the use in situations when some log traces violate a compliance model in order to produce a weakened compliance model showing what part of the compliance regulations does hold in the company practice.

For the future work, we plan to introduce quantitative measurements for vacuity, which are interesting in the context of large logs. In this case, a strengthened model can show in which way *most* of the process executions satisfy the compliance model, and which part of the behaviour is rather exceptional for the system in question. The quantitative approach can also be useful for logs with noise.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, M.H.: Declarative workflows: Balancing between flexibility and support. Computer Science - Research and Development 23, 99–113 (2009)
2. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient Detection of Vacuity in Temporal Model Checking. Formal Methods in System Design 18, 141–163 (2001)

3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM (1999)
4. van Hee, K.M., Liu, Z., Sidorova, N.: Is my event log complete? - A probabilistic approach to process mining. In: Proceedings of RCIS 2011, pp. 1–7. IEEE (2011)
5. Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On Enabling Data-Aware Compliance Checking of Business Process Models. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 332–346. Springer, Heidelberg (2010)
6. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. International Journal on Software Tools for Technology Transfer 4(2), 224–233 (2003)
7. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing Declarative Logic-Based Models from Labeled Traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007)
8. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The Glory of the Past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
9. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: Analyzing Vessel Behavior using Process Mining in the Poseidon book edited by Springer (to appear)
10. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
11. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Proceedings of CIDM 2011, pp. 192–199. IEEE (2011)
12. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring Business Constraints with the Event Calculus. Technical Report DEIS-LIA-002-11, University of Bologna, Italy (2011)
13. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
14. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-Based Workflow Models: Change Made Easy. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 77–94. Springer, Heidelberg (2007)
15. Pesic, M., Schonenberg, M.H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: Proceedings of EDOC 2007, pp. 287–300. IEEE Computer Society (2007)
16. Pnueli, A.: The Temporal Logic of Programs. In: Proceedings of FOCS 1977, pp. 46–57. IEEE Computer Society (1977)
17. Purandare, M., Somenzi, F.: Vacuum Cleaning CTL Formulae. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 485–499. Springer, Heidelberg (2002)

A Formal Interactive Verification Environment for the Plan Execution Interchange Language

Camilo Rocha[1], Héctor Cadavid[2], César Muñoz[3], and Radu Siminiceanu[4]

[1] University of Illinois at Urbana-Champaign, Urbana IL, USA
[2] Escuela Colombiana de Ingeniería, Bogotá, Colombia
[3] NASA Langley Research Center, Hampton VA, USA
[4] National Institute of Aerospace, Hampton VA, USA

Abstract. The Plan Execution Interchange Language (PLEXIL) is an open source synchronous language developed by NASA for commanding and monitoring autonomous systems. This paper reports the development of the PLEXIL's Formal Interactive Verification Environment (PLEXIL5), a graphical interface to the formal executable semantics of PLEXIL. Among its main features, PLEXIL5 provides model checking of plans with support for formula editing and visualization of counterexamples, interactive simulation of plans at different granularity levels, and random initialization of external environment variables. The formal verification capabilities of PLEXIL5 are illustrated by means of a human-automation interaction model.

1 Introduction

Plan execution is a centerpiece of systems involving intelligent software agents such as robotics, unmanned vehicles, and habitats. The *Plan Execution Interchange Language* PLEXIL [8] is a synchronous language developed by NASA to support autonomous spacecraft operations. Programs in PLEXIL, called *plans*, specify actions to be executed by an autonomous system as part of normal spacecraft operations or as reactions to changes in the environment. The computer system on board the spacecraft that executes plans is called the *executive* and is a safety-critical component of the space mission. The PLEXIL Executive [18] is an open source executive developed by NASA (`http://plexil.sourceforge.net`). PLEXIL has been used on mid-size applications such as robotic rovers, a prototype of a Mars drill, and to demonstrate automation capabilities for potential future use on the International Space Station. A summary of PLEXIL's syntax and semantics is presented in Section 2.

Spacecraft operations require flexible, efficient, and reliable plan execution. Given its critical nature, PLEXIL's operational semantics has been formally specified in the Prototype Verification System (PVS) [5]. Moreover, key metatheoretical properties of the language, such as determinism and compositionality, have been mechanically verified in PVS [6]. Based on this formalization, a formal executable semantics of PLEXIL has been specified in the rewriting logic engine Maude [7]. The executable semantics of PLEXIL serves as an efficient formal

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 343–357, 2012.

interpreter of the language and, as illustrated by this paper, is at the core of the PLEXIL Formal Interactive Verification Environment (PLEXIL5).

PLEXIL5 is an interactive environment for verifying and testing PLEXIL plans and for studying new features and possible variants of the language. A proof of concept of such an environment was originally presented in [11], but that tool was mainly concerned with the semantic validation of the language. This paper reports significant progress on the evolution of this proof of concept into an environment for the validation and *formal verification* of PLEXIL plans. To emphasize these new capabilities, the word "Visual" in the original acronym became "Verification" in the new system. PLEXIL5 consists of a graphical environment developed in Java that interfaces with the rewriting logic semantics in Maude. Users are not required to have any knowledge of the Maude system to take advantage of PLEXIL5's formal analysis capabilities. An overview of some architecture and design features, software metrics, and aspects of user interaction are presented in Section 3.

The formal analysis capabilities in PLEXIL5 are based on the rewriting logic semantics of the language and the formal analysis tools available in the Maude system, such as the rewriting engine, the model checker, and the strategy language [4]. The environment supports the verification of temporal properties on PLEXIL plans. These properties can be provided by the user or automatically generated from plan annotations such as preconditions, invariants, and postconditions. PLEXIL5 provides a mechanisms for modeling the interaction of plans with the external environment. Technical details on the formal analysis capabilities, i.e., simulation, model checking, and semantic validation, offered by PLEXIL5 are given in Section 4. As a case study, Section 5 presents a formalization of a simple cruise control system in PLEXIL and illustrates how PLEXIL5 can aid in discovering and correcting errors in plans. The case study presented in this paper and more information about PLEXIL5 is available from `http://shemesh.larc.nasa.gov/people/cam/PLEXIL`.

2 PLEXIL Overview

This section presents an overview of PLEXIL, a synchronous language for automation developed by NASA. The reader is referred to [8] for a detailed description of the language.

A PLEXIL program, called a *plan*, is a tree of *nodes* representing a hierarchical decomposition of tasks. Interior nodes, called *list nodes*, provide control structure and naming scope for local variables. The primitive actions of a plan are specified in the leaf nodes. Leaf nodes can be *assignment nodes*, which assign values to local variables, *command nodes*, which call external commands, or *empty nodes*, which do nothing. PLEXIL plans interact with a functional layer that provides the interface with the external environment. This functional layer executes the external commands and communicates the status and result of their execution to the plan through *external variables*.

Nodes have an *execution state*, which can be *inactive*, *waiting*, *executing*, *iterationend*, *failing*, *finishing*, or *finished*, and an *execution outcome*, which can

be *unknown*, *skipped*, *success*, or *failure*. They can declare local variables that are accessible to the node in which they are declared and all its descendants. In contrast to local variables, the execution state and outcome of a node are visible to all nodes in the plan. Assignment nodes also have a *priority* that is used to solve race conditions. The *internal state* of a node consists of the current values of its execution state, execution outcome, and local variables.

Each node is equipped with a set of *gate conditions* and *check conditions* that govern the execution of a plan. Gate conditions provide control flow mechanisms that react to external events. In particular, the *start condition* specifies when a node starts its execution, the *end condition* specifies when a node ends its execution, the *repeat condition* specifies when a node can repeat its execution, and the *skip condition* specifies when the execution of a node can be skipped. Check conditions are used to signal abnormal execution states of a node and they are *pre-condition*, *post-condition*, and *invariant*. The language includes Boolean, integer and floating-point arithmetic, and string expressions. It also includes *lookup expressions* that read the value of external variables provided to the plan through the executive. Expressions appear in conditions, assignments, and arguments of commands. Each one of the basic types is extended by a special value *unknown* that can occur in the case, for instance, when a lookup fails.

The execution of a plan in PLEXIL is driven by external events that trigger changes in the gate conditions. All nodes affected by a change in a gate condition synchronously respond to the event by modifying their internal state. These internal modifications may trigger more changes in gate conditions that in turn are synchronously processed until quiescence is reached for all nodes involved. External events are considered in the order in which they are received. An external event and all its cascading effects are processed before the next event is considered. This behavior is known as run-to-completion semantics.

Henceforth, the notation (Γ, π) is used to represent the execution state of a plan, where Γ is a set of external variables and their current values, and π is a set of nodes and their internal states. Formally, the semantics of PLEXIL is defined on states (Γ, π) by a compositional layer of five reduction relations [8]. The *atomic relation* describes the execution of an individual node in terms of state transitions triggered by changes in the environment. The *micro relation* describes the *synchronous* reduction of the atomic relation with respect to the *maximal redexes strategy*, i.e., the synchronous application of the atomic relation to the maximal set of nodes of a plan. The remaining three relations are the *quiescence relation*, the *macro relation*, and the *execution relation* that, respectively, describe the reduction of the micro relation until normalization, the interaction of a plan with the external environment upon one external event, and the n-iteration of the macro relation corresponding to n time steps.

Consider the PLEXIL plan in Figure 1. The plan consists of a root node *Exchange* of type list, and leaf nodes *SetX* and *SetY* of type *assignment*. The node *Exchange* declares two local variables x and y. The values of these variables are exchanged by the synchronous execution of the node assignments *SetX* and *SetY*. The node *Exchange* also declares a start condition and an invariant condition.

The start condition states that the node can start executing whenever the value of an external variable T is greater than 10. The invariant condition states that at any state of execution the values of x and y add up to 3.

```
Exchange: {
    Integer x = 1;
    Integer y = 2;
    StartCondition: Lookup(T) > 10;
    Invariant: x+y == 3;
    NodeList:
        SetX: { Assignment: x = y; }
        SetY: { Assignment: y = x; }
}
```

Fig. 1. A PLEXIL plan that reads the value of an external variable T and synchronously exchanges the values of internal variables x and y

3 PLEXIL5

PLEXIL5 is a graphical environment for the formal simulation and verification of PLEXIL plans, and the validation of the intended semantics of the language against its rewriting logic semantics. This section presents an overview of its architecture and design, including some software metrics, and aspects regarding user interaction in the environment.

3.1 Architecture and Design

Figure 2 depicts PLEXIL5's key components and their interaction. The graphical user interface has been developed in Java using the *model-view-controller* pattern and, for some views on execution states, uses third-party open-source libraries such as JGraph and JGoodies. The object oriented *model* represents the hierarchical structure of plans, their execution behavior, and the external environment. The *view* consists of several classes that present the user with views of the tree-like-structure of plans. The *controller* consists of a custom *controller-facade* class and *listener* classes using and extending the Java framework.

PLEXIL5 supports a number of input formats defining plans. For this purpose, the tool links a series of parsers and translators that internally (i) generate the format supported by the rewriting logic semantics of the language implemented in Maude and (ii) construct an object oriented plan model from Maude's output. The parsers are all generated from XML Schemas and BNF-like specifications by external tools, such as ANTLR, JAXB, and JavaCC. Some of the XML schemas have been borrowed and adapted from PLEXIL's software distribution. Java and Maude communicate as processes at the operating system's level with help of the Java/Maude Integration API, developed as part of the PLEXIL5 framework.

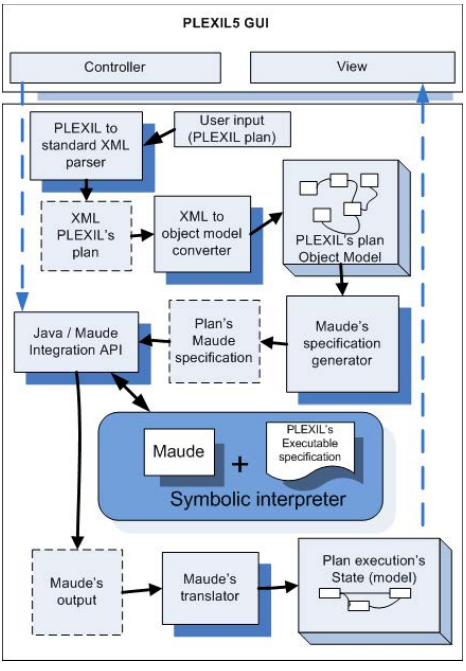


Fig. 2. PLEXIL5 logical components and their interaction

The implementation of PLEXIL5 consists of 270 Java classes and 38 Maude modules, among other resources. The Java classes comprise 85K lines of code, of which 24K are automatically generated by the external tools. The Maude modules are 2K lines of code.

3.2 User Interaction

Once PLEXIL5 is launched for the first time, the user is required to select the folder containing PLEXIL's rewriting logic semantics. This selection is kept for future sessions and can be modified at any time through the graphical interface.

A plan is read from a file containing one of the several supported PLEXIL notations, then transformed into an object model, and ultimately presented to the user with a visual representation of the initial state of the plan. The visual representation of plans implemented in the prototype described in [11] was based on trees. That representation is only practical for plans with a small numbers of nodes. In the current version, plans are displayed by default as tables and the hierarchical structure of plans is given by tabular indentations. The original tree representation of plans is still supported.

A plan can be edited by the user with the help of the graphical user interface. The plan can be accompanied by a *script* file, in XML format, describing the values of external variables at different macro steps. External variables can be initialized to random Boolean, integer, and floating-point values, and can be

specified using an enumeration or a range. The following XML script specifies the values for the external variable T of integer type, for the plan *Exchange* in Figure 1. In the first macro step the variable T is assigned the value 2, at the second macro step it is assigned a random non-negative value, and in the third macro step it is assigned a value randomly chosen from 2 or 7.

```
<Script>
  <Step>
    <State name="T" type="int"><Value>2</Value></State>
  </Step>
  <Step>
    <State name="T" type="int"><RandomValue min="0"/></State>
  </Step>
  <Step>
    <State name="T" type="int">
      <RandomValue><Enum value="2"/><Enum value="7"/></RandomValue>
    </State>
  </Step>
</Script>
```

The translation process of a plan and its script only takes place the first time the plan is loaded and every time a plan is edited.

Plans can be executed at the level of the micro, quiescence, macro, and execution semantic relations, with undo-redo support. The tool can automatically generate formulas for checking invariant, pre, and post conditions, and the user can also define formulas from atomic predicates parameterized by the active plan. A Maude specification in the syntax of the rewriting logic semantics is generated from the object model every time the user requests to perform an action on the current state of execution. This Maude specification and the user's command are delegated to Maude via the Java/Maude integration API. The resulting output is then used to generate a new instance of the object model that is graphically presented to the user.

4 Formal Analysis in PLEXIL5

The formal analysis capabilities offered by PLEXIL5 are based on PLEXIL's rewriting logic semantics written in Maude. This section provides technical details on how these capabilities, i.e., simulation and debugging, model checking, and semantic validation, are implemented in PLEXIL5 via Maude's verification tools. This section uses standard notation and terminology of rewriting logic; the user is referred to [4] for more details.

Rewriting logic [10] is a general semantic framework that unifies a wide range of models of concurrency. Rewriting logic specifications can be executed in Maude, a high-performance rewriting logic implementation, and thus take advantage of all the formal analysis tools available in Maude. A rewriting logic specification is a tuple $\mathcal{R} = (\Sigma, E, R)$ where (Σ, E) is an order-sorted equational theory with signature Σ and equations E, and a set of rewrite rules R. The

equational theory (Σ, E) induces the congruence relation $=_E$ on the set T_Σ of Σ-ground terms defined for any $t, u \in T_\Sigma$ by $t =_E u$ if and only if $(\Sigma, E) \vdash t = u$. The expression $\mathcal{T}_{\Sigma/E}$ denotes the initial algebra of (Σ, E). Similarly, a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\longrightarrow_{\mathcal{R}}$ on the set $T_{\Sigma/E}$ of E-equivalence classes of ground Σ-terms defined by any $t, u \in T_\Sigma$ by $[t]_E \longrightarrow_{\mathcal{R}} [u]_E$ if and only if $t \longrightarrow u$ can be deduced from $\mathcal{R}$ by the deduction rules in [3]. The tuple $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \longrightarrow_{\mathcal{R}})$ is called the *initial reachability model* of $\mathcal{R}$. Intuitively, $\mathcal{T}_{\mathcal{R}}$ represents the concurrent system whose states are the set of E-equivalence classes of ground Σ-terms and whose concurrent transitions are specified by R.

4.1 Simulation and Debugging

The rewriting logic semantics of a synchronous language such as PLEXIL poses interesting practical challenges because Maude implements the maximal concurrency of rewrite rules by interleaving, i.e., asynchronous concurrency. To overcome this situation, the rewriting logic semantics $\mathcal{P} = (\Sigma_{\mathcal{P}}, E_{\mathcal{P}}, R_{\mathcal{P}})$ of PLEXIL implements a serialization procedure [13] that completely and correctly simulates PLEXIL's synchronous semantics. Since PLEXIL is deterministic, the serialization procedure implemented by $\mathcal{P}$ can be equationally defined in $E_{\mathcal{P}}$, thus avoiding the interleaving semantics associated with rewrite rules in Maude.

A PLEXIL node in $\mathcal{P}$ is a term object denoted $\langle O : C \mid a_1 : v_1, \ldots, a_m : v_m \rangle$, where O is the object's identifier corresponding to the node's qualified name, C is the object's class corresponding to the node's type, e.g., assignment, list, local variable, etc., and where v_1 to v_m are the current values of the attributes a_1 to a_m corresponding to the node's internal state of execution. An execution state of a plan has sort *PlxState* and the form $(\overline{\Gamma}, \overline{\pi})$, where $\overline{\Gamma}$ has the structure of a multiset of pairs representing the set Γ of external variables and their values, and $\overline{\pi}$ is a term that has the structure of a multiset of objects representing the set of nodes π. Multiset union is denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is multiset rewriting supported in Maude.

Given a PLEXIL plan p, PLEXIL5 internally generates the rewrite theory $\mathcal{P}(p)$ that extends $\mathcal{P}$ with the constructs of p. The rewrite theory $\mathcal{P}(p)$ is a formal model of p in rewriting logic and it induces the rewrite relation $\longrightarrow_{\mathcal{P}(p),m}$ that uses the equationally defined serialization procedure to soundly and completely simulate PLEXIL's synchronous micro relation for p.

Maude's strategy language [9] is used to simulate the quiescence, macro, and execution semantic relations from $\longrightarrow_{\mathcal{P}(p),m}$. By definition, the quiescence relation $\longrightarrow_{\mathcal{P}(p),q}$ is the normalized relation obtained from $\longrightarrow_{\mathcal{P}(p),m}$, namely, $\longrightarrow_{\mathcal{P}(p),q} = \longrightarrow_{\mathcal{P}(p),m}^{\downarrow}$. Because $\longrightarrow_{\mathcal{P}(p),m}$ is deterministic, the quiescence relation $\longrightarrow_{\mathcal{P}(p),q}$ is also deterministic. For the purpose of simulating the macro and execution relations, PLEXIL5 allows for the definition of a sequence $\mathbf{\Gamma} = \Gamma_0, \Gamma_1, \ldots, \Gamma_n$ of collections of external variables indicating their value at each time step $0, 1, \ldots, n$ ($\mathbf{\Gamma}$ can be empty when the plan does not depend on external variables). For a sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$, the macro relation $\longrightarrow_{\mathcal{P}(p),M}$ is defined by $(\overline{\Gamma_i}, \overline{\pi}) \longrightarrow_{\mathcal{P}(p),M} (\overline{\Gamma'}, \overline{\pi'})$ if and only if $\Gamma' = \Gamma_{i+1}$ and $(\overline{\Gamma_i}, \overline{\pi}) \longrightarrow_{\mathcal{P}(p),q}$

$(\overline{\Gamma_i}, \overline{\pi'})$, for $0 \leq i < n$. The execution relation $\longrightarrow_{\mathcal{P}(p),E}$ normalizes a given state with the macro relation and then normalizes the resulting state further with the quiescence relation in the last time step. It is formally defined by $\longrightarrow_{\mathcal{P}(p),E} = \longrightarrow^{\downarrow}_{\mathcal{P}(p),M} \circ \longrightarrow_{\mathcal{P}(p),q}$.

4.2 Model Checking

In general, a Kripke structure can be associated with the initial reachability model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ by making explicit the intended sort *State* of states in the signature Σ and the relevant set Φ of atomic predicates on states. The set of atomic propositions Φ is defined by an equational theory $\mathcal{E}_\Phi = (\Sigma_\Phi, E \uplus E_\Phi)$. Signature Σ_Φ contains Σ and a sort *Bool* with constant symbols $\bot$ and $\top$ of sort *Bool*, predicate symbols $\phi : State \rightarrow Bool$ for each $\phi \in \Phi$, and optionally some auxiliary function symbols. Equations in E_Φ define the predicate symbols in Σ_Φ and auxiliary function symbols, if any, including the Boolean operations on the sort *Bool*. For $\phi \in \Phi$ and a ground term of sort *State* $t \in T_{\Sigma,State}$, the *semantics* of ϕ in $\mathcal{T}_{\mathcal{R}}$ is defined by $\mathcal{E}_\Phi$ as follows: $\phi(t)$ *holds* in $\mathcal{T}_{\mathcal{R}}$ if and only if $\mathcal{E}_\Phi \vdash \phi(t) = \top$. This defines the Kripke structure $\mathcal{K}^\Phi_{\mathcal{R}} = (T_{\Sigma/E,State}, \longrightarrow_{\mathcal{R}}, L_\Phi)$ with labeling function L_Φ defined for any $t \in T_{\Sigma,State}$ by $\phi \in L_\Phi(t)$, written $\mathcal{K}^\Phi_{\mathcal{R}}, t \models \phi$, if and only if $\phi(t)$ holds in $\mathcal{T}_{\mathcal{R}}$. All formulas of the Linear Temporal Logic (LTL) can be interpreted in $\mathcal{K}^\Phi_{\mathcal{R}}$ in the standard way.

PLEXIL5 supports LTL model checking of plans at the level of the micro relation on the sort *PlxState*. The set of atomic propositions is parameterized by the set of qualified names of nodes and variables (internal and external) in the plan to be model checked. The BNF-like notation in Figure 3 defines the syntax of the atomic propositions $\Phi_\mathcal{N}$ and formulas $LTL_\mathcal{N}$ for model checking a plan p with set of qualified names $\mathcal{N}$. The collection of PLEXIL Boolean expressions parameterized by $\mathcal{N}$ is denoted with $BExpr_\mathcal{N}$. They include comparison operators for Boolean and arithmetic expressions, evaluation of local variables, and lookups. Atomic propositions $\Phi_\mathcal{N}$ include the constants *true* and *false*, predicates for testing the status, outcome, and gate and checking conditions of a node. They also

$$
\begin{aligned}
Status_\mathcal{N} &::= inactive \mid waiting \mid executing \mid finishing \mid iterended \mid failing \mid finished \\
Failure_\mathcal{N} &::= parent \mid invariant \mid pre \mid post \\
Outcome_\mathcal{N} &::= unknown \mid skipped \mid success \mid fail(\mu) \\
Cond_\mathcal{N} &::= start \mid end \mid repeat \mid pre \mid post \mid invariant \\
\Phi_\mathcal{N} &::= true \mid false \mid status(\lambda, \sigma) \mid outcome(\lambda, \omega) \mid \psi(\lambda, \delta) \mid eval(\delta) \\
LTL_\mathcal{N} &::= \alpha \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \varphi \Rightarrow \varphi' \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi' \mid \varphi\mathbf{W}\varphi' \mid \varphi\mathbf{R}\varphi'
\end{aligned}
$$

with variables

$$
\begin{array}{cccc}
\mu : Failure_\mathcal{N} & \lambda : \mathcal{N} & \sigma : Status_\mathcal{N} & \omega : Outcome_\mathcal{N} \\
\psi : Cond_\mathcal{N} & \delta : BExpr_\mathcal{N} & \alpha : \Phi_\mathcal{N} & \varphi, \varphi' : LTL_\mathcal{N}
\end{array}
$$

Fig. 3. Parameterized atomic predicates $\Phi_\mathcal{N}$ and LTL formulas $LTL_\mathcal{N}$ in PLEXIL5

include the atomic proposition *eval* for testing PLEXIL's Boolean expressions. Formulas in $LTL_{\mathcal{N}}$ include the usual Boolean connectives, and the temporal connectives 'always' (**G**), 'eventually' (**F**), 'next' (**X**), 'until' (**U**), 'weak until' (**W**), and 'release' (**R**), all interpreted in the standard way.

Given a plan p, an initial state (Γ, π), and a $LTL_{\mathcal{N}}$ formula φ over the names $\mathcal{N}$ in p, PLEXIL5 uses Maude's LTL model checker to check $\mathcal{K}^{\Phi_{\mathcal{N}}}_{\mathcal{P}(p)}, (\overline{\Gamma}, \overline{\pi}) \models \varphi$, where $\mathcal{K}^{\Phi_{\mathcal{N}}}_{\mathcal{P}(p)}$ is the Kripke structure associated with $\mathcal{T}_{\mathcal{P}(p)}$, with the set of states $T_{\Sigma(p)/E(p),PlxState}$, transition relation $\longrightarrow_{\mathcal{P}(p),m}$, and labeling function $L_{\Phi_{\mathcal{N}}}$.

PLEXIL5 provides interactive means for producing and visually inspecting counterexamples. The model checking window is equipped with three predefined checks that can be performed on any PLEXIL program: "check invariants", "check pre-conditions", and "check post-conditions". Pushing one of the three buttons generates the corresponding LTL formulas. Additionally, an input field in provided to enter custom, application specific LTL formulas specified using the above syntax. For example, the formula

$$\mathbf{G}\ \mathit{invariant}(\mathit{Exchange}, \mathit{true}) \ \wedge\ \mathbf{F}\ \mathit{status}(\mathit{Exchange}, \mathit{finished})$$

for the plan *Exchange* in Figure 1, tests the invariant of node *Exchange* and that it will eventually transition to state *finished*.

Counterexamples are displayed in a tree table with collapsible nodes, conforming to the PLEXIL program tree structure, and can be interactively navigated step-by-step for debugging and validation purposes.

4.3 Semantic Validation

The rewriting logic semantics $\mathcal{P}$ is being used to study variations and extensions of PLEXIL. This section provides examples of such variants and extensions that have been studied in PLEXIL5.

PLEXIL's macro relation is especially important because it is the semantic relation defining the interaction of a plan with the external environment. On the one hand, it is reasonable to have access to the external state as often as possible so that lookups in each atomic reduction can use the latest information available. On the other hand, it can be computationally expensive to implement such a policy because sensors or similar artifacts can significantly delay the execution of a plan. Another dimension of the problem arises when a guard of an internal loop depends on external variables: should the loop run-to-completion regardless of the possible updates to the value of the variable in its guard, or should it stop at each iteration so that the value of the external variable can be updated? The rewriting logic semantics $\mathcal{P}$ has been modified to accommodate alternative specifications of PLEXIL's semantics with different definitions of the macro relation. These semantic variants of PLEXIL have been studied and exercised using PLEXIL5. Thanks to its modular design, PLEXIL5 can integrate the alternative semantics with a click of a button: the user has the freedom to choose the formal semantics of preference.

Another concrete example that illustrates the use of PLEXIL5 by the designers of the language is the addition of a gate condition called *exit condition*. The exit condition provides a mechanism for a clean interruption of execution. In order to support this feature in PLEXIL5, the specification of the PLEXIL's atomic relation in Maude was modified to include the intended semantics. Given the modular definition of the formal semantics none of the other rewriting relations were modified.

5 A Case Study

A cruise control system adapted from [2] is presented to showcase the model checking capabilities implemented in PLEXIL5. Originally, the model was designed for the Enhanced Operator Function Model (EOFM) formalism, which is intended for the study of human behavior in a human-computer interaction framework. However, PLEXIL shares many characteristics with EOFM, including the hierarchical structure of tasks decomposed into sub-tasks and the execution governed by conditions (*pre*, *post*, *repeat*, *invariant*).

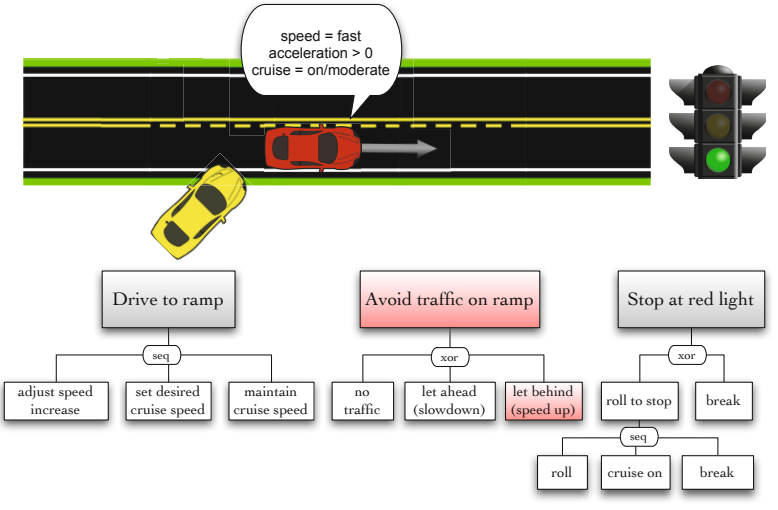


Fig. 4. Cruise control model with task hierarchy

5.1 Model Description

The model consists of three main components: car, driver, and stoplight, which execute synchronously. The operator drives the car on a street, approaching the stoplight. Other cars may merge into the lane from a side ramp, roughly midway through. The car has three controls represented in the model: the gas and break pedals to manage speed and acceleration, and a cruise button to switch the cruise mode on/off and set the cruise speed. The human operator's plan is to safely

operate the controls of the car to achieve three sub-goals: (i) drive at a desired cruise speed (ii) avoid the possible merging traffic from the ramp, and (iii) obey the traffic light at the intersection, i.e., stop the car in time if the light turns red. All three properties can be represented in PLEXIL. Here we focus on the third, which is a safety property.

The model parameters are: the geometry of the intersection, i.e., the length of each street segment; the location of the ramp along the street, in distance units; the stoplight cycle length, in time units, for each color; and the speed range, in distance per time units.

Model variables. The model variables and their range are selected according to an abstraction scheme that discretizes the values to allow finite state model checking, yet leaves sufficient information to make the study relevant.

- $distance \in [0 \ldots 55]$, the distance of the car to the intersection;
- $time \in [0 \ldots 28]$;
- $speed \in \{stopped = 0, slow = 1, moderate = 2, fast = 3\}$;
- $acceleration \in \{-1, 0, 1\}$;
- $cruise\_enabled \in \{true, false\}$;
- $cruise\_speed \in \{0, 1, 2, 3\}$;

Transitions. The car advances according to its speed until it reaches the intersection, formally, update $distance := distance - speed{*}timestep$ while the condition $speed > 0 \wedge distance > 0$ holds. The discretized speed can change by at most one unit at a time, hence the possible values for acceleration are only $\{-1, 0, 1\}$. The stoplight counts down the time units to the end of the green-yellow-red cycle by assigning $stoplight := stoplight - timestep$. The light is red in the time interval $[0 \ldots 8]$, yellow in $[9 \ldots 12]$, and green in $[13 \ldots 28]$.

The complexity resides in capturing the decision making of the driver. In the first segment, the driver wants to set the cruise control to a desired speed (e.g., *moderate*). The driver has the choice to accelerate from slow or decelerate from *fast*, then enable the cruise control which will maintain the desired speed. On the second segment, the driver needs to react to merging traffic from the ramp. If any car is on the ramp, the driver may choose to let the other car in front by slowing down, or behind by speeding up. On the last segment, the driver has to react to the stoplight turning red. The driver may choose to maintain the speed and then break before reaching the stoplight, or roll to a stop by releasing the gas pedal.

Comparison with the EOFM model.

- The original abstraction has been refined in PLEXIL to allow more distance and time divisions, making it more realistic; in the EOFM model the distance is heavily discretized (abstract locations 0 to 7) and not coordinated with the time to travel each segment.
- Non-determinism is introduced by lookups of environment variables. The script plays out a sequence of random choices for three Boolean environment variables: *MergingTraffic*, *LetBehind*, *RollStop*.

- Some of the concepts are essentially cognitive in nature, as they depend on the subjective (sometimes erroneous) perceptions and assessments of the situation by the human operator, hence they cannot be as naturally captured in the formal model. However, both normative and erroneous behaviors are captured in the PLEXIL model, and it is the job of the model checker to discover violations.
- The synchronous behavior is natural in PLEXIL, no further instrumentation is necessary, while in EOFM synchrony has to be expressly specified, using appropriate decomposition operators.

5.2 Verification

The property of interest can be expressed either as a global invariant in the PLEXIL model itself and checked with the generic "check invariants" button, or entered in the LTL Model Checking dialog window. The safety property is specified in the top level task node *Main* as the invariant condition:

$$not(stoplight <= red\ and\ distance == 0\ and\ speed > 0),$$

stating that it is not the case that the vehicle is moving at the intersection when the light is red.

The PLEXIL5 simulator shows that the execution of the plan ends with the outcome *invariantFail* for the root node (and *parentFail* for the successor nodes) when the environment variables *MergingTraffic*, *LetBehind*, and *RollStop* are all *true*. The result of model checking the safety property is an execution trace where the formula is violated. The counter example can be described as follows:

1. the car enters at *low* speed at $distance = 55$ and $time = 28$;
2. the driver accelerates to the desired *moderate* speed and sets the cruise on at $time = 20$ and $distance = 42$;
3. at the ramp, with $distance = 33$, the driver decides to let the merging car behind by accelerating to *fast* at $time = 12$ and $distance = 25$;
4. the stoplight light turns yellow, the driver chooses to roll to a stop (assessing there is sufficient distance to the intersection to do so, by releasing the gas pedal);
5. with the acceleration negative, the driver does not disengage the cruise mode, the cruise control kicks in and maintains the cruise *speed* to *moderate* for one execution cycle at $time = 6$ and $distance = 10$;
6. the effect of the automation is that the (now necessary) breaking is too late to decrease the speed from *moderate* to *low* at $time = 2$ and $distance = 2$, and then *stopped* in two execution cycles; and
7. when time expires, the car is moving in the intersection on the red light.

The PLEXIL5 model checking environment provides the means for detecting the aforementioned error using the predefined "check invariant" test. To correct the problem, the node corresponding to the "roll to stop" action has to be rectified, in order to include a check on the status of the cruise control. The driver either has

to make sure it is disabled before initiating the "roll to stop" option or manually disable it. In PLEXIL, this can be instrumented via a *start condition* or, by duality, with the corresponding negated *skip condition*. No other combination of environment lookup variables leads to violations in this model.

The full model of the cruise control system consists of 252 lines of PLEXIL code. The generated Maude file is 929 lines long.

6 Related Work and Conclusion

An executable semantics of PLEXIL has been developed by P. J. Strauss in the Haskell language [16] with the aim of analyzing features of the language regarding the plan interaction with the environment. As a result, new data types representing the external world have been proposed for more dynamic runtime behavior of PLEXIL plans. More recently, D. Balasubramanian et al. have proposed Polyglot, a framework for modeling and analyzing multiple Statechart formalisms, and have initiated research towards the formal analysis of a Statechart-based semantics of PLEXIL [1]. In rewriting logic literature, similar approaches to the one used in PLEXIL5 have been proposed for other languages and protocol analysis. In particular, A. Verdejo and N. Martí-Oliet [17] have explored the idea of having easy-tool-building techniques from operational semantics specified in Maude. S. Santiago et al. [15] have developed a graphical user interface that animates the Maude-NPA verification process, displaying the complete search tree and allowing users to display graphical representations of final and intermediate nodes of the search tree. Maude-NPA is a crypto protocol analysis tool developed in Maude that takes into account algebraic properties of crypto-systems.

This paper reported significant progress on the evolution of PLEXIL5, an environment for the verification and validation of NASA's synchronous language PLEXIL. The environment uses the formal semantics of the language written in Maude to formally analyze PLEXIL plans. Maude is a rewriting logic formalism that provides advanced verification tools such as a fast rewriting engine and a LTL model checker. In PLEXIL5, the user is presented with the option to execute any combination of the micro, quiescence, macro, and execution reduction relations. In this way, the user has the freedom to determine the level of detail for simulating and debugging plans. The verification tools are available in PLEXIL5 through a graphical interface that does not require knowledge of rewriting logic or the Maude system.

The formal environment has been used by the developers of the language to investigate semantic variations and extensions of PLEXIL. These include a new semantics for the execution of loops and a new feature in the language to handle exit conditions. Furthermore, several minor issues in the original intended semantics of PLEXIL have been identified and corrected. PLEXIL5 has become a formal benchmark for executives and will be part of PLEXIL's distribution in an upcoming release.

An important subset of PLEXIL's core language is currently supported by PLEXIL5. The main features of the language that are not supported by the formal semantics are array variables (arrays are not directly supported in Maude),

PLEXIL's resource model, which enables the specification of resource requirements for commands, and Update nodes, which provide an importing mechanism to the language. Regarding research on PLEXIL5's rewriting logic semantics, future work will explore the possibility of having an operational semantics of the language using the framework presented in [12], so that the dependency between the rewrite rules specifying the atomic relation and the serialization procedure can be eliminated. Another interesting alternative is to study the extension of the K framework [14] with priorities for state transitions, so that it can accommodate the specification of the atomic relation. Regarding the verification and validation capabilities in PLEXIL5, future work will add support for symbolic execution and concolic testing of PLEXIL plans, and will study scalability issues with mid-size and large plans.

Acknowledgments. The authors would like to thank Michael Dalal and the Planning and Scheduling group at NASA Ames for fruitful discussions on PLEXIL and suggestions for the PLEXIL5 tool. They are also grateful to the anonymous referees for comments that helped to improve the paper. This work is supported by NASA's Autonomous Systems and Avionics Project, Software Verification Algorithms. The first author has been partially supported by NSF grant CCF 09-05584. The first, second, and fourth authors have been partially supported by the National Aeronautics and Space Administration at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A awarded to the National Institute of Aerospace.

References

1. Balasubramanian, D., Păsăreanu, C., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple Statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA, pp. 45–55. ACM (2011)
2. Bolton, M.L., Bass, E.J., Siminiceanu, R.I.: A systematic approach to model checking human-automation interaction using task analytic models. IEEE Transactions on Systems, Man, and Cybernetics–Part A: Systems and Humans 41(5), 961–976 (2011)
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1-3), 386–414 (2006)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Dowek, G., Muñoz, C., Păsăreanu, C.: A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA (2008)
6. Dowek, G., Muñoz, C., Păsăreanu, C.: A formal analysis framework for PLEXIL. In: Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems (September 2007)
7. Dowek, G., Muñoz, C., Rocha, C.: Rewriting logic semantics of a plan execution language. In: Klin, B., Sobocinski, P. (eds.) SOS. EPTCS, vol. 18, pp. 77–91 (2009)
8. Estlin, T., Jónsson, A., Păsăreanu, C., Simmons, R., Tso, K., Verma, V.: Plan Execution Interchange Language (PLEXIL). Technical Memorandum TM-2006-213483, NASA (2006)

9. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for maude strategies. Electronic Notes in Theoretical Computer Science 238(3), 227–247 (2009)
10. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
11. Rocha, C., Muñoz, C., Cadavid, H.: A graphical environment for the semantic validation of a plan execution language. In: IEEE International Conference on Space Mission Challenges for Information Technology, pp. 201–207. IEEE Computer Society, Los Alamitos (2009)
12. Rocha, C., Muñoz, C.: Simulation and Verification of Synchronous Set Relations in Rewriting Logic. In: da Silva Simão, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 60–75. Springer, Heidelberg (2011)
13. Rocha, C., Muñoz, C., Dowek, G.: A formal library of set relations and its application to synchronous languages. Theoretical Computer Science 412(37), 4853–4866 (2011)
14. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
15. Santiago, S., Talcott, C.L., Escobar, S., Meadows, C., Meseguer, J.: A graphical user interface for Maude-NPA. Electronic Notes in Theoretical Computer Science 258(1), 3–20 (2009)
16. Strauss, P.J.: Executable semantics for PLEXIL: simulating a task-scheduling language in Haskell. Master's thesis, Oregon State University (2009)
17. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. Formal Methods in System Design 27(1-2), 113–172 (2005)
18. Verma, V., Jónsson, A., Păsăreanu, C., Iatauro, M.: Universal Executive and PLEXIL: Engine and language for robust spacecraft control and operations. In: Proceedings of the American Institute of Aeronautics and Astronautics Space Conference (2006)

Author Index

Baeten, Jos 253
Bakhshi, Rena 158
Balasingham, Ilangko 143
Barker, Philip 84
Barreto, Raimundo 128
Berthing, Jesper 69
Besnard, Loïc 113
Blackmore, Tim 84
Börger, Egon 1
Bosser, Anne-Gwenn 206
Boström, Pontus 69
Bresciani, Riccardo 191
Brooke, Phillip J. 206
Bubel, Richard 283
Butterfield, Andrew 191

Cadavid, Héctor 343
Calder, Muffy 21
Cisternino, Antonio 1
Cordeiro, Lucas 128
Cunha, Alcino 312

Diaconescu, Denisa 221
Dias Neto, Arilo 128
Dongol, Brijesh 39

Eder, Kerstin 84

Faria, João P. 296

Garis, Ana 312
Gautier, Thierry 113
Gervasi, Vincenzo 1

Halliwell, David 84
Hansen, Dominik 24
Hayes, Ian J. 39
Hooman, Jozef 268

Isobe, Yoshinao 54

Ji, Ran 283
Johnsen, Einar Broch 143

Kazemeyni, Fatemeh 143

Laibinis, Linas 237
Le Guernic, Paul 113
Lensink, Leonard 174
Leuschel, Michael 24
Leustean, Ioana 221
Lopes, Antónia 296

Maggi, Fabrizio Maria 327
Markovski, Jasen 253
Moller, Faron 54
Mooij, Arjan J. 268
Mousavi, Mohammad Reza 158
Muñoz, César 343

Ngo, Van Chan 113
Nguyen, Hoang Nga 54
Nielson, Flemming 99

Owe, Olaf 143

Paiva, Ana C.R. 296, 312
Petre, Luigia 221

Ramaram, Naresh 84
Rebello de Andrade, Francisco 296
Riesco, Daniel 312
Riis Nielson, Hanne 99
Rocha, Camilo 343
Rocha, Herbert 128
Roggenbach, Markus 54

Schunselaar, Dennis M.M. 327
Sere, Kaisa 69, 221
Sevegnani, Michele 21
Sidorova, Natalia 327
Siminiceanu, Radu 343
Smetsers, Sjaak 174
Stefanescu, Gheorghe 221

Talpin, Jean-Pierre 113
Tarasyuk, Anton 237

Tian, HaiYun 206
Troubitsyna, Elena 237
Tsiopoulos, Leonidas 69

Vain, Jüri 69
van Beek, Dirk A. 253
van Eekelen, Marko 174
van Wezep, Hans 268

Woehrle, Matthias 158

Zhang, Fuyuan 99