

Marc Frappier Uwe Glässer
Sarfraz Khurshid Régine Laleau
Steve Reeves (Eds.)

LNCS 5977

Abstract State Machines, Alloy, B and Z

Second International Conference, ABZ 2010
Orford, QC, Canada, February 2010
Proceedings

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Marc Frappier Uwe Glässer
Sarfraz Khurshid Régine Laleau
Steve Reeves (Eds.)

Abstract State Machines, Alloy, B and Z

Second International Conference, ABZ 2010
Orford, QC, Canada, February 22-25, 2010
Proceedings

Volume Editors

Marc Frappier
Université de Sherbrooke, Dept. d'informatique
Sherbrooke, Québec, J1K 2R1, Canada
E-mail: Marc.Frappier@USherbrooke.ca

Uwe Glässer
Simon Fraser University, School of Computing Science
Burnaby, BC, V5A 1S6, Canada
E-mail: glaesser@cs.sfu.ca

Sarfraz Khurshid
University of Texas at Austin, Dept. of Electrical and Comp. Engineering
1 University Station C5000, Austin, TX 78712-0240, USA
E-mail: khurshid@ece.utexas.edu

Régine Laleau
Université Paris-Est Créteil
IUT Sénart/Fontainebleau, Dept. informatique
Route forestière Hurtault, 77300 Fontainebleau, France
E-mail: laleau@univ-paris12.fr

Steve Reeves
The University of Waikato, Dept. of Computer Science
Hamilton 3240, New Zealand
E-mail: stever@cs.waikato.ac.nz

Library of Congress Control Number: 2010920043

CR Subject Classification (1998): F.4, G.2, I.2.3, D.3.2, F.3, I.2.4, F.4.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-11810-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-11810-4 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2010
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12990580 06/3180 5 4 3 2 1 0

Preface

ABZ 2010 was held in the beautiful natural setting of Orford in the Eastern Townships of Québec, during February 22–25, 2010, midway through the Canadian winter and the 21st Winter Olympics, bringing participants from all over the world to brave this rigorous climate.

ABZ covers recent advances in four equally rigorous methods for software and hardware development: Abstract State Machines (ASM), Alloy, B and Z. They share a common conceptual framework, centered around the notions of state and operation, and promote mathematical precision in the modeling, verification, and construction of highly dependable systems.

These methods have continuously matured over the past decade, reaching a stage where they have been successfully integrated into industrial practice in various areas like trains, automobiles, aerospace, smart cards, virtual machines, and business processes. Their development is influenced by both research and practice, which mutually nurture each other.

ABZ has both a long and a short history. With the aim of stimulating cross-fertilization between these four methods, it has merged their individual conference and workshop series which started in 1986 for Z, 1994 for ASM, 1996 for B, and 2006 for Alloy. The first ABZ conference was held in London in 2008; ABZ 2010 is the second edition. The conference remains organized as four separate Program Committees.

The first day of the conference was devoted to tutorials on Alloy and BART (B automatic refinement tool) and to a workshop on tool building in formal methods (WTBFM 2010). The main program of the conference started with a one-day plenary session, with two invited speakers and four special presentations, one for each method. The invited speakers were Daniel Jackson from MIT and Sofiène Tahar from Concordia University. The special presentations were selected among the contributions submitted to the conference for their cross-fertilization potential and their research quality. The next two days of the main program were divided into two parallel tracks, merging presentations of long and short papers to stimulate interactions between all participants. Long papers cover a broad spectrum of research, from foundational to applied work. A total of 60 long papers from 15 countries were submitted, of which 26 were accepted. Short papers, included here as one-page abstracts, address work in progress, industrial experience reports, and tool descriptions. An extended version of these abstracts is available on the conference website at <http://abzconference.org>.

Holding such an event requires a lot of effort from several people. We wish to express them our deepest gratitude for making ABZ 2010 a success to: members of the Program Committees and reviewers, for their rigorous evaluations and discussions, Springer, for their support in publishing these proceedings, Université de Sherbrooke and Université Paris-Est Créteil, for their financial and

organizational support, Orford Arts Centre, for their logistical support. Special thanks to Jérémy Milhau for designing and managing the conference website, Lynn Lebrun, for managing conference registrations, and Chantal Proulx, for on-site support. The conference was managed with Easychair, which rightfully bears its name.

More information on ABZ can be found at <http://abzconference.org>.

February 2010

Marc Frappier
Uwe Glässer
Sarfraz Khurshid
Régine Laleau
Steve Reeves



Conference Organization

Program Chairs

Marc Frappier (General Chair)	University of Sherbrooke, Canada
Uwe Glässer (ASM Chair)	Simon Fraser University, Canada
Sarfraz Khurshid (Alloy Chair)	University of Texas at Austin, USA
Régine Laleau (B Chair)	University of Paris-Est, France
Steve Reeves (Z Chair)	University of Waikato, New Zealand

ASM Program Committee

Egon Börger	University of Pisa, Italy
Andreas Friesen	SAP Research, Germany
Uwe Glässer (Chair)	Simon Fraser University, Canada
Susanne Graf	Verimag, France
Elvinia Riccobene	University of Milan, Italy
Klaus-Dieter Schewe	Massey University, New Zealand
Anatol Slissenko	University of Paris-Est, France
Jan Van den Bussche	University of Hasselt, Belgium
Margus Veanes	Microsoft Research, USA
Charles Wallace	Michigan Technological University, USA

Alloy Program Committee

Juergen Dingel	Queen's University, Canada
Andriy Dunets	Universität Augsburg, Germany
Kathi Fisler	Worcester Polytechnic Institute, USA
Daniel Jackson	Massachusetts Institute of Technology, USA
Jeremy Jacob	University of York, UK
Sarfraz Khurshid (Chair)	University of Texas at Austin, USA
Viktor Kuncak	EPFL, Switzerland
Daniel LeBerre	Universite d'Artois, France
Darko Marinov	University of Illinois, USA
Jose Oliveira	University of Minho, Portugal
Burkhardt Renz	FH Gießen-Friedberg, Germany
Kevin Sullivan	University of Virginia, USA
Mana Taghdiri	Universität Karlsruhe, Germany
Pamela Zave	AT&T Laboratories, USA

B Program Committee

Yamine Ait Ameer	LISI/ENSMA-UP, France
Richard Banach	University of Manchester, UK

VIII Organization

Juan Bicarregui	STFC Rutherford Appleton Laboratory, UK
Michael Butler	University of Southampton, UK
Daniel Dollé	Siemens Transportation Systems, France
Steve Dunne	University of Teesside, UK
Neil Evans	AWE plc Aldermaston, UK
Mamoun Filali Amine	University of Toulouse, France
Frédéric Gervais	University of Paris-Est, France
Jacques Julliard	University of Besançon, France
Régine Laleau (Chair)	University of Paris-Est, France
Thierry Lecomte	Clearsy, France
Michael Leuschel	University of Düsseldorf, Germany
Dominique Méry	University of Nancy, France
Anamaria Martins Moreira	UFRN, Natal, Brazil
Annabelle McIver	Macquarie University, Australia
Marie-Laure Potet	VERIMAG, France
Ken Robinson	University of New South Wales, Australia
Emil Sekerinski	McMaster University, Canada
Helen Treharne	University of Surrey, UK
Laurent Voisin	Systerel, France
Marina Waldèn	Åbo Akademi University, Finland

Z Program Committee

Rob Arthan	Lemma 1 Ltd., UK
Eerke Boiten	University of Kent, UK
Jonathan Bowen	Museophile Ltd / King's College London, UK
Ana Calvacanti	University of York, UK
John Derrick	University of Sheffield, UK
Anthony Hall	independent consultant, UK
Ian Hayes	University of Queensland, Australia
Rob Hierons	Brunel University, UK
Jonathan Jacky	University of Washington, USA
Steve Reeves (Chair)	University of Waikato, New Zealand
Thomas Santen	European Microsoft Innovation Ctr, Germany

Local Organization

Michel Embe Jiague	University of Sherbrooke, Canada / University of Paris-Est, France
Benoît Fraikin	University of Sherbrooke, Canada
Marc Frappier	University of Sherbrooke, Canada
Frédéric Gervais	University of Paris-Est, France
Pierre Konopacki	University of Sherbrooke, Canada / University of Paris-Est, France
Régine Laleau	University of Paris-Est, France

Sylvie Lavoie
Lynn Le Brun
Jérémy Milhau

University of Sherbrooke, Canada
University of Sherbrooke, Canada
University of Sherbrooke, Canada /
University of Paris-Est, France
University of Sherbrooke, Canada
University of Sherbrooke, Canada

Chantal Proulx
Richard St-Denis

External Reviewers

Benaïssa Benaïssa
Jens Bendisposto
Jean-Paul Bodeveix
Eerke Boiten
Pontus Boström
Alexandre Cortier
Alcino Cunha
David Déharbe
Cristian Dittamo
Roozbeh Farahbod
Elie Fares
Angelo Gargantini
Milos Gligoric
Gudmund Grov
Stefan Hallerstede
Piper Jackson
Rajesh Karmani
Olga Kouchnarenko
Jens Lemcke
Issam Maamria
Pierre-Alain Masson
Jérémy Milhau

Aleksandar Milicevic
Hassan Mountassir
Wolfgang Mueller
Florian Nafz
Marcel Oliveira
Marta Olszewska (Plaska)
Edgar Pek
Tirdad Rahmani
Joris Rehm
Patrizia Scandurra
Gerhard Schellhorn
James Sharp
Neeraj Singh
Ove Soerensen
Jennifer Sorge
Kurt Stenzel
Bill Stoddart
Bogdan Tofan
Edward Turner
Qing Wang
Kuat Yessenov
Frank Zeyda

Table of Contents

Invited Talks

A Structure for Dependability Arguments (Abstract)	1
<i>Daniel Jackson and Eunsuk Kang</i>	
Formal Probabilistic Analysis: A Higher-Order Logic Based Approach	2
<i>Osman Hasan and Sofiène Tahar</i>	

ASM Papers

Synchronous Message Passing and Semaphores: An Equivalence Proof	20
<i>Iain Craig and Egon Börger</i>	
AsmL-Based Concurrency Semantic Variations for Timed Use Case Maps	34
<i>Jameleddine Hassine</i>	
Bârun: A Scripting Language for CoreASM	47
<i>Michael Altenhofen and Roozbeh Farahbod</i>	
AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications	61
<i>Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene</i>	
An Executable Semantics of the SystemC UML Profile	75
<i>Elvinia Riccobene and Patrizia Scandurra</i>	

Alloy Papers

Specifying Self-configurable Component-Based Systems with FracToy . . .	91
<i>Alban Tiberghien, Philippe Merle, and Lionel Seinturier</i>	
Trace Specifications in Alloy	105
<i>Jeremy L. Jacob</i>	
An Imperative Extension to Alloy	118
<i>Joseph P. Near and Daniel Jackson</i>	
Towards Formalizing Network Architectural Descriptions	132
<i>Joud Khoury, Chaouki T. Abdallah, and Gregory L. Heileman</i>	

Lightweight Modeling of Java Virtual Machine Security Constraints	146
<i>Mark C. Reynolds</i>	
Alloy+HotCore: A Fast Approximation to Unsat Core	160
<i>Nicolás D’Ippolito, Marcelo F. Frias, Juan P. Galeotti,</i> <i>Esteban Lanzarotti, and Sergio Mera</i>	

B Papers

Supporting Reuse in Event B Development: Modularisation Approach	174
<i>Alexei Iliasov, Elena Troubitsyna, Linas Laibinis,</i> <i>Alexander Romanovsky, Kimmo Varpaaniemi,</i> <i>Dubravka Ilic, and Timo Latvala</i>	
Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance	189
<i>Andrew Ireland, Gudmund Grov, and Michael Butler</i>	
Applying the B Method for the Rigorous Development of Smart Card Applications	203
<i>Bruno Gomes, David Déharbe, Anamaria Moreira, and Katia Moraes</i>	
Automatic Verification for a Class of Proof Obligations with SMT-Solvers	217
<i>David Déharbe</i>	
A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking	231
<i>Edd Turner, Michael Butler, and Michael Leuschel</i>	
Development of a Synchronous Subset of AADL	245
<i>Mamoun Filali-Amine and Julia Lawall</i>	
Matelas: A Predicate Calculus Common Formal Definition for Social Networking	259
<i>Nestor Catano and Camilo Rueda</i>	
Structured Event-B Models and Proofs	273
<i>Stefan Hallerstede</i>	
Refinement-Animation for Event-B — Towards a Method of Validation	287
<i>Stefan Hallerstede, Michael Leuschel, and Daniel Plagge</i>	
Reactivising Classical B	302
<i>Steve Dunne and Frank Zeyda</i>	

Event-B Decomposition for Parallel Programs	319
<i>Thai Son Hoang and Jean-Raymond Abrial</i>	

Z Papers

Communication Systems in ClawZ	334
<i>Michael Vernon, Frank Zeyda, and Ana Cavalcanti</i>	

Formalising and Validating RBAC-to-XACML Translation Using Lightweight Formal Methods	349
<i>Mark Slaymaker, David Power, and Andrew Simpson</i>	

Towards Formally Templated Relational Database Representations in Z	363
<i>Nicolas Wu and Andrew Simpson</i>	

Translating Z to Alloy	377
<i>Petra Malik, Lindsay Groves, and Clare Lenihan</i>	

ABZ Short Papers (Abstracts)

B-ASM: Specification of ASM <i>à la</i> B	391
<i>David Michel, Frédéric Gervais, and Pierre Valarcher</i>	

A Case for Using Data-Flow Analysis to Optimize Incremental Scope-Bounded Checking	392
<i>Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry</i>	

On the Modelling and Analysis of Amazon Web Services Access Policies	394
<i>David Power, Mark Slaymaker, and Andrew Simpson</i>	

Architecture as an Independent Variable for Aspect-Oriented Application Descriptions	395
<i>Hamid Bagheri and Kevin Sullivan</i>	

ParAlloy: Towards a Framework for Efficient Parallel Analysis of Alloy Models	396
<i>Nicolás Rosner, Juan P. Galeotti, Carlos G. Lopez Pombo, and Marcelo F. Frias</i>	

Introducing Specification-Based Data Structure Repair Using Alloy	398
<i>Razieh Nokhbeh Zaeem and Sarfraz Khurshid</i>	

Secrecy UML Method for Model Transformations	400
<i>Waël Hassan, Nadera Slimani, Kamel Adi, and Luigi Logrippo</i>	

Improving Traceability between KAOS Requirements Models and B Specifications	401
<i>Abderrahman Matoussi and Dorian Petit</i>	
Code Synthesis for Timed Automata: A Comparison Using Case Study	403
<i>Anaheed Ayoub, Ayman Wahba, Ashraf Salem, and Mohamed Sheirah</i>	
Towards Validation of Requirements Models	404
<i>Atif Mashkooor and Abderrahman Matoussi</i>	
A Proof Based Approach for Formal Verification of Transactional BPEL Web Services	405
<i>Idir Aït Sadoune and Yamine Aït Ameer</i>	
On an Extensible Rule-Based Prover for Event-B	407
<i>Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh</i>	
B Model Abstraction Combining Syntactic and Semantic Methods	408
<i>Jacques Julliand, Nicolas Stouls, Pierre-Christophe Bué, and Pierre-Alain Masson</i>	
A Basis for Feature-Oriented Modelling in Event-B	409
<i>Jennifer Sorge, Michael Poppleton, and Michael Butler</i>	
Using Event-B to Verify the Kmelia Components and Their Assemblies	410
<i>Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix</i>	
Starting B Specifications from Use Cases	411
<i>Thiago C. de Sousa and Aryldo G. Russo Jr</i>	
Integrating SMT-Solvers in Z and B Tools	412
<i>Alessandro Cavalcante Gurgel, Valério Gutemberg de Medeiros Jr., Marcel Vinicius Medeiros Oliveira, and David Boris Paul Déharbe</i>	
Formal Analysis in Model Management: Exploiting the Power of CZT	414
<i>James R. Williams, Fiona A.C. Polack, and Richard F. Paige</i>	
Author Index	415

A Structure for Dependability Arguments

Daniel Jackson and Eunsuk Kang

Massachusetts Institute of Technology
{dnj, eskang}@csail.mit.edu

Abstract. How should a software system be verified? Much research is currently focused on attempts to show that code modules meet their specifications. This is important, but bugs in code are not the weakest link in the chain. The larger problems are identifying and articulating critical properties, and ensuring that the components of a system - not only software modules, but also hardware peripherals, physical environments, and human operators - together establish them. Another common assumption is that verification must take system design and implementation as given. I'll explain the rationale for, and elements of, a new approach to verification, in which design is driven by verification goals, and verification arguments are structured in a way that exposes the relationship between critical properties and the components that ensure them.

Formal Probabilistic Analysis: A Higher-Order Logic Based Approach

Osman Hasan and Sofiène Tahar

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8, Canada
{o_hasan,tahar}@ece.concordia.ca

Abstract. Traditionally, simulation is used to perform probabilistic analysis. However, it provides less accurate results and cannot handle large-scale problems due to the enormous CPU time requirements. Recently, a significant amount of formalization has been done in higher-order logic that allows us to conduct precise probabilistic analysis using theorem proving and thus overcome the limitations of the simulation. Some major contributions include the formalization of both discrete and continuous random variables and the verification of some of their corresponding probabilistic and statistical properties. This paper describes the infrastructures behind these capabilities and their utilization to conduct the probabilistic analysis of real-world systems.

1 Introduction

“In short, we can only pretend to achieve a relative faultless construction, not an absolute one, which is clearly impossible. A problem solution for which is still in its infancy is finding the right methodology to perform an environmental model that is a “good” approximation of the real environment. It is clear that a probabilistic approach would certainly be very useful for doing this.”

J. Abrial, Faultless Systems: Yes We Can!, IEEE Computer Magazine, 42(9):30-36, 2009.

Probabilistic analysis is a tool of fundamental importance for the analysis of hardware and software systems. These systems usually exhibit some random or unpredictable elements. Examples include, failures due to environmental conditions or aging phenomena in hardware components and the execution of certain actions based on a probabilistic choice in randomized algorithms. Moreover, these systems act upon and within complex environments that themselves have certain elements of unpredictability, such as noise effects in hardware components and the unpredictable traffic pattern in the case of telecommunication protocols. Due to these random components, establishing the correctness of a system under all circumstances usually becomes impractically expensive. The engineering

approach to analyze a system with these kind of unpredictable elements is to use probabilistic analysis. The main idea is to mathematically model the unpredictable elements of the given system and its environment by appropriate random variables. The probabilistic properties of these random variables are then used to judge system's behaviors regarding parameters of interest, such as downtime, availability, number of failures, capacity, and cost. Thus, instead of guaranteeing that the system meets some given specification under all circumstances, the probability that the system meets this specification is reported.

Simulation is the most commonly used computer based probabilistic analysis technique. Most simulation softwares provide a programming environment for defining functions that approximate random variables for probability distributions. The random elements in a given system are modeled by these functions and the system is analyzed using computer simulation techniques, such as the Monte Carlo Method [31], where the main idea is to approximately answer a query on a probability distribution by analyzing a large number of samples. Statistical quantities, such as average and variance, may then be calculated, based on the data collected during the sampling process, using their mathematical relations in a computer. Due to the inherent nature of simulation, the probabilistic analysis results attained by this technique can never be termed as 100% accurate. The accuracy of the hardware and software system analysis results has become imperative these days because of the extensive usage of these systems in safety-critical areas, such as, medicine and transportation. Therefore, simulation cannot be relied upon for the analysis of such systems.

In order to overcome the above mentioned limitations, we propose to use higher-order-logic theorem proving for probabilistic analysis. Higher-order logic [11] is a system of deduction with a precise semantics and is expressive enough to be used for the specification of almost all classical mathematics theories. Due to its high expressive nature, higher-order-logic can be utilized to precisely model the behavior of any system, while expressing its random or unpredictable elements in terms of formalized random variables, and any kind of system property, including the probabilistic and statistical ones, as long as they can be expressed in a closed mathematical form. Interactive theorem proving [16] is the field of computer science and mathematical logic concerned with precise computer based formal proof tools that require some sort of human assistance. Due to its interactive nature, interactive theorem proving can be utilized to reason about the correctness of probabilistic or statistical properties of systems, which are usually undecidable.

In this paper, we present a higher-order-logic theorem proving based framework that can be utilized to conduct formal probabilistic analysis of systems. We provide a brief overview of higher-order-logic formalizations that facilitate the formal modeling of random systems [18,19,26] and formal reasoning about their probabilistic and statistical properties [17,20,21]. We show how these capabilities fit into the overall formal probabilistic analysis framework and also point out some of the missing links that need further investigations. For illustration purposes, we discuss the formal probabilistic analysis of some real-world

systems from the areas of telecommunications, nanoelectronics and computational algorithms.

The rest of the paper is organized as follows: Section 2 describes the proposed probabilistic analysis framework and how the already formalized mathematical concepts of probability theory fit into it. The case studies are presented in Section 3. Section 4 summarizes the state-of-the-art in the formal probabilistic analysis domain and compares these approaches with higher-order-logic theorem proving based analysis. Finally, Section 5 concludes the paper.

2 Formal Probabilistic Analysis Framework

A hypothetical model of a higher-order-logic theorem proving based probabilistic analysis framework is given in Fig. 1, with some of its most fundamental components depicted with shaded boxes. The starting point of probabilistic analysis is a system description and some intended system properties and the goal is to check if the given system satisfies these given properties. Due to the differences in the underlying mathematical foundations of discrete and continuous random variables [42], we have divided system properties into two categories, i.e., system properties related to discrete random variables and system properties related to continuous random variables.

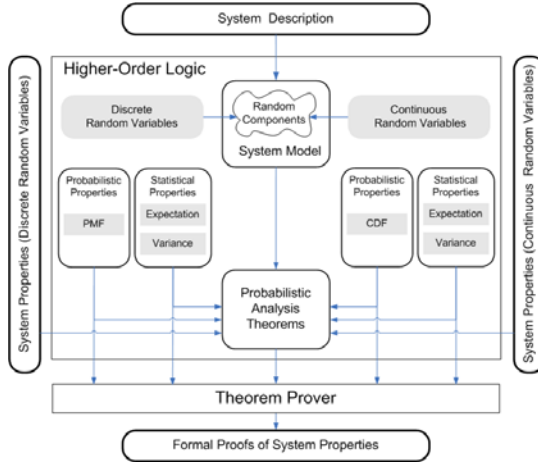


Fig. 1. Higher-order Logic based Probabilistic Analysis Framework

The first step in the proposed approach is to construct a model of the given system in higher-order-logic. For this purpose, the foremost requirement is the availability of infrastructures that allow us to formalize all kinds of discrete and continuous random variables as higher-order-logic functions, which in turn can be used to represent the random components of the given system in its higher-order-logic model. The second step is to utilize the formal model of the system

to express system properties as higher-order-logic theorems. The prerequisite for this step is the ability to express probabilistic and statistical properties related to both discrete and continuous random variables in higher-order-logic. All probabilistic properties of discrete and continuous random variables can be expressed in terms of their *Probability Mass Function* (PMF) and *Cumulative Distribution Function* (CDF), respectively. Similarly, most of the commonly used statistical properties can be expressed in terms of the expectation and variance characteristics of the corresponding random variable. Thus, we require the formalization of mathematical definitions of PMF, CDF, expectation and variance for both discrete and continuous random variables in order to be able to express the given system's reliability characteristics as higher-order-logic theorems. The third and the final step for conducting probabilistic analysis in a theorem prover is to formally verify the higher-order-logic theorems developed in the previous step using a theorem prover. For this verification, it would be quite handy to have access to a library of some pre-verified theorems corresponding to some commonly used properties regarding probability distribution functions, expectation and variance. Since, we can build upon such a library of theorems and thus speed up the verification process. The formalization details regarding the above mentioned steps are briefly described now.

2.1 Discrete Random Variables and the PMF

A random variable is called discrete if its range, i.e., the set of values that it can attain, is finite or at most countably infinite [42]. Discrete random variables can be completely characterized by their PMFs that return the probability that a random variable X is equal to some value x , i.e., $Pr(X = x)$. Discrete random variables are quite frequently used to model randomness in probabilistic analysis. For example, the Bernoulli random variable is widely used to model the fault occurrence in a component and the Binomial random variable may be used to represent the number of faulty components in a lot.

Discrete random variables can be formalized in higher-order-logic as deterministic functions with access to an infinite Boolean sequence B^∞ ; an infinite source of random bits with data type ($natural \rightarrow bool$) [26]. These deterministic functions make random choices based on the result of popping bits in the infinite Boolean sequence and may pop as many random bits as they need for their computation. When the functions terminate, they return the result along with the remaining portion of the infinite Boolean sequence to be used by other functions. Thus, a random variable that takes a parameter of type α and ranges over values of type β can be represented by the function

$$\mathcal{F} : \alpha \rightarrow B^\infty \rightarrow (\beta \times B^\infty)$$

For example, a *Bernoulli*($\frac{1}{2}$) random variable that returns 1 or 0 with probability $\frac{1}{2}$ can be modeled as

```
⊢ bit = λs. (if shd s then 1 else 0, stl s)
```

where the variable s represents the infinite Boolean sequence and the functions `shd` and `stl` are the sequence equivalents of the list operations 'head' and 'tail'. A function of the form $\lambda x.t$ represents a lambda abstraction function that maps x to $t(x)$. The function `bit` accepts the infinite Boolean sequence and returns a pair with the first element equal to either 0 or 1 and the second element equal to the unused portion of the infinite Boolean sequence.

The higher-order-logic formalization of probability theory [26] also consists of a probability function \mathbb{P} from sets of infinite Boolean sequences to *real* numbers between 0 and 1. The domain of \mathbb{P} is the set \mathcal{E} of events of the probability. Both \mathbb{P} and \mathcal{E} are defined using the Carathéodory's Extension theorem, which ensures that \mathcal{E} is a σ -algebra: closed under complements and countable unions. The formalized \mathbb{P} and \mathcal{E} can be used to formally verify all basic axioms of probability. Similarly, they can also be used to prove probabilistic properties for random variables. For example, we can formally verify the following probabilistic property for the function `bit`, defined above,

$$\vdash \mathbb{P} \{s \mid \text{fst}(\text{bit } s) = 1\} = \frac{1}{2}$$

where the function `fst` selects the first component of a pair and $\{x \mid C(x)\}$ represents a set of all elements x that satisfy the condition C .

The above mentioned infrastructure can be utilized to formalize most of the commonly used discrete random variables and verify their corresponding PMF relations [26]. For example, the formalization and verification of Bernoulli and Uniform random variables can be found in [26] and of Binomial and Geometric random variables can be found in [21].

2.2 Continuous Random Variables and the CDF

A random variable is called continuous if it ranges over a continuous set of numbers that contains all real numbers between two limits [42]. Continuous random variables can be completely characterized by their CDFs that return the probability that a random variable X is exactly less than or equal to some value x , i.e., $Pr(X \leq x)$. Examples of continuous random variables include measuring the arrival time T of a data packet at a web server ($S_T = \{t \mid 0 \leq t < \infty\}$) and measuring the voltage V across a resistor ($S_V = \{v \mid -\infty < v < \infty\}$).

The sampling algorithms for continuous random variables are non-terminating and hence require a different formalization approach than discrete random variables, for which the sampling algorithms are either guaranteed to terminate or satisfy probabilistic termination, meaning that the probability that the algorithm terminates is 1. One approach to address this issue is to utilize the concept of the nonuniform random number generation [9], which is the process of obtaining arbitrary continuous random numbers using a Standard Uniform random number generator. The main advantage of this approach is that we only need to formalize the Standard Uniform random variable from scratch and use it to model other continuous random variables by formalizing the corresponding nonuniform random number generation method.

Based on the above approach, a methodology for the formalization of all continuous random variables for which the inverse of the CDF can be represented in a closed mathematical form is presented in [18]. The first step in this methodology is the formalization of the Standard Uniform random variable, which can be done by using the formalization approach for discrete random variables and the formalization of the mathematical concept of limit of a *real* sequence [15]:

$$\lim_{n \rightarrow \infty} (\lambda n. \sum_{k=0}^{n-1} (\frac{1}{2})^{k+1} X_k) \quad (1)$$

where X_k denotes the outcome of the k^{th} random bit; *True* or *False* represented as 1 or 0, respectively. The formalization details are outlined in [19].

The second step in the methodology for the formalization of continuous probability distributions is the formalization of the CDF and the verification of its classical properties. This is followed by the formal specification of the mathematical concept of the inverse function of a CDF. This definition along with the formalization of the Standard Uniform random variable and the CDF properties, can be used to formally verify the correctness of the Inverse Transform Method (ITM) [9]. The ITM is a well known nonuniform random generation technique for generating nonuniform random variables for continuous probability distributions for which the inverse of the CDF can be represented in a closed mathematical form. Formally, it can be verified for a random variable X with CDF F using the Standard Uniform random variable U as follows [18].

$$Pr(F^{-1}(U) \leq x) = F(x) \quad (2)$$

The formalized Standard Uniform random variable can now be used to formally specify any continuous random variable for which the inverse of the CDF can be expressed in a closed mathematical form as $X = F^{-1}(U)$. Whereas, the formally verified ITM, given in Equation (2), can be used to prove the CDF for such a formally specified random variable. This approach has been successfully utilized to formalize and verify Exponential, Uniform, Rayleigh and Triangular random variables [18].

2.3 Statistical Properties for Discrete Random Variables

In probabilistic analysis, statistical characteristics play a major role in decision making as they tend to summarize the probability distribution characteristics of a random variable in a single number. Due to their widespread interest, the computation of statistical characteristics has now become one of the core components of every contemporary probabilistic analysis framework.

The expectation for a function of a discrete random variable, which attains values in the positive integers only, is defined as follows [30]

$$Ex_fn[f(X)] = \sum_{n=0}^{\infty} f(n)Pr(X = n) \quad (3)$$

where X is the discrete random variable and f represents a function of X . The above definition only holds if the associated summation is convergent, i.e., $\sum_{n=0}^{\infty} f(n)Pr(X = n) < \infty$. The expression of expectation, given in Equation (3), has been formalized in [20] as a higher-order-logic function using the probability function \mathbb{P} . The expected value of a discrete random variable that attains values in positive integers can now be defined as a special case of Equation (3)

$$Ex[X] = Ex_fn[(\lambda n.n)(X)] \quad (4)$$

when f is an identity function. In order to verify the correctness of the above definitions of expectation, they are utilized in [20,21] to formally verify the following classical expectation properties.

$$Ex\left[\sum_{i=1}^n R_i\right] = \sum_{i=1}^n Ex[R_i] \quad (5)$$

$$Ex[a + bR] = a + bEx[R] \quad (6)$$

$$Pr(X \geq a) \leq \frac{Ex[X]}{a} \quad (7)$$

These properties not only verify the correctness of the above definitions but also play a vital role in verifying the expectation characteristics of discrete random components of probabilistic systems, as will be seen in Section 3 of this paper.

Variance of a random variable X describes the difference between X and its expected value and thus is a measure of its dispersion.

$$Var[X] = Ex[(X - Ex[X])^2] \quad (8)$$

The above definition of variance has been formalized in higher-order-logic in [20] by utilizing the formal definitions of expectation, given in Equations (3) and (4). This definition is then formally verified to be correct by proving the following classical variance properties for it [20,21].

$$Var[R] = Ex[R^2] - (Ex[R])^2 \quad (9)$$

$$Var\left[\sum_{i=1}^n R_i\right] = \sum_{i=1}^n Var[R_i] \quad (10)$$

$$Pr(|X - Ex[X]| \geq a) \leq \frac{Var[X]}{a^2} \quad (11)$$

These results allow us to reason about expectation, variance and tail distribution properties of any formalized discrete random variable that attains values in positive integers, e.g., the formal verification for Bernoulli, Uniform, Binomial and Geometric random variables is presented in [21].

2.4 Statistical Properties for Continuous Random Variables

The most commonly used definition of expectation, for a continuous random variable X , is the probability density-weighted integral over the real line [34].

$$E[X] = \int_{-\infty}^{+\infty} xf(x)dx \quad (12)$$

The function f in the above equation represents the *Probability Density Function* (PDF) of X and the integral is the well-known Reimann integral. The above definition is limited to continuous random variables that have a well-defined PDF. A more general, but not so commonly used, definition of expectation for a random variable X , defined on a probability space (Ω, Σ, P) [10], is as follows.

$$E[X] = \int_{\Omega} XdP \quad (13)$$

This definition utilizes the Lebesgue integral and is general enough to cater for both discrete and continuous random variables. The reason behind its limited usage in the probabilistic analysis domain is the complexity of solving the Lebesgue integral, which takes its foundations from the measure theory that most engineers and computer scientists are not familiar with.

The obvious advantage of using Equation (12) for formalizing expectation of a continuous random variable is the user familiarity with Reimann integral that usually facilitates the reasoning process regarding the expectation properties in the theorem proving based probabilistic analysis approach. On the other hand, it requires extended real numbers, $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$, whereas all the foundational work regarding theorem proving based probabilistic analysis, outlined above, has been built upon the standard real numbers \mathbb{R} , formalized by Harrison [15]. The expectation definition given in Equation (13) does not involve extended real numbers, as it accommodates infinite limits without any ad-hoc devices due to the inherent nature of the Lebesgue integral. It also offers a more general solution. The limitation, however, is the compromise on the interactive reasoning effort, as it is not a straightforward task for a user to build on this definition to formally verify the expectation of a random variable.

We have formalized the expectation of a continuous random variable as in Equation (13) by building on top of a higher-order-logic formalization of Lebesgue integration theory [6]. Starting from this definition, two simplified expressions for the expectation are verified that allow us to reason about expectation of a continuous random variable in terms of simple arithmetic operations [17]. The first expression is for the case when the given continuous random variable X is bounded in the positive interval $[a, b]$.

$$E[X] = \lim_{n \rightarrow \infty} \left[\sum_{i=0}^{2^n-1} \left(a + \frac{i}{2^n}(b-a) \right) P \left\{ a + \frac{i}{2^n}(b-a) \leq X < a + \frac{i+1}{2^n}(b-a) \right\} \right] \quad (14)$$

The second expression is for an unbounded positive random variable [10].

$$E[X] = \lim_{n \rightarrow \infty} \left[\sum_{i=0}^{n2^n-1} \frac{i}{2^n} P \left\{ \frac{i}{2^n} \leq X < \frac{i+1}{2^n} \right\} + nP(X \geq n) \right] \quad (15)$$

Both of the above expressions do not involve any concepts from Lebesgue integration theory and are based on the well-known arithmetic operations like summation, limit of a real sequence, etc. Thus, users can simply utilize them, instead of Equation (13), to reason about the expectation properties of their random variables and gain the benefits of the original Lebesgue based definition. The formal verification details for these expressions are given in [17]. These expressions are further utilized to verify the expected values of Uniform, Triangular and Exponential random variables [17]. The above mentioned definition and simplified expressions will also facilitate the formalization of variance and the verification of its corresponding properties.

3 Applications

We now illustrate the usage of the above mentioned formalization, for conducting probabilistic analysis of some real-world systems.

3.1 Probabilistic Analysis of the Coupon Collector's Problem

The Coupon Collector's problem [34] refers to the problem of probabilistically evaluating the number of trials required to acquire all unique, say n , coupons from a collection of multiple copies of these coupons that are independently and uniformly distributed. The problem is similar to the example when each box of cereal contains one of n different coupons and once you obtain one of every type of coupon, you win a prize. The Coupon Collector's problem is a commercially used computational problem and is commonly used for the identification of routers that are encountered in packet communication between two hosts [34].

Based on the probabilistic analysis framework, presented in Section 2, particularly the capabilities to formally specify discrete random variables and formally reason about the statistical properties of systems, a formal probabilistic analysis of the Coupon Collector's problem is presented in [21]. The first goal is to verify that the expected value of acquiring all n coupons is $nH(n)$, where $H(n)$ is the *harmonic number* ($\sum_{i=1}^n 1/i$). Based on this expectation value, the next step is to reason about the tail distribution properties of the Coupon Collector's problem using the formally verified Markov's and Chebyshev's inequalities.

The first step in the proposed approach is to model the behavior of the given system as a higher-order-logic function, while representing its random component using the formalized random variables. The Coupon Collector's problem can be formalized by modeling the total number of trials required to obtain all n unique coupons, say T , as a sum of the number of trials required to obtain each distinct coupon, i.e., $T = \sum_{i=1}^n T_i$, where T_i represents the number of trials to

obtain the i^{th} coupon, while $i - 1$ distinct coupons have already been acquired. The advantage of breaking the random variable T into the sum of n random variables $T_1, T_2 \dots, T_n$ is that each T_i can be modeled by the Geometric random variable function. It is important to note here that the probability of success for these Geometric random variables would be different from one another and would be equal to the probability of finding a new coupon while conducting uniform selection trials on the available n coupons. Thus, the success probability depends on the number of already acquired coupons and can be modeled using the higher-order-logic function for the discrete Uniform random variable. Using this approach the Coupon Collector's problem has been modeled in [21] as a higher-order-logic function, `coupon_collector`, that accepts a positive integer greater than 0, $n + 1$, which represents the total number of distinct coupons that are required to be collected. The function returns the number of trials for acquiring these $n + 1$ distinct coupons. Now, using this function along with the formal definitions of expectation and variance and their formally verified corresponding properties, given in Section 2.3, the following statistical characteristics can be verified [21].

$$\vdash \forall n. \text{expec}(\text{coupon_collector}(n + 1)) = (n + 1) \left(\sum_{i=0}^{n+1} \frac{1}{i+1} \right)$$

$$\vdash \forall n a. 0 < a \Rightarrow \mathbb{P} \{s \mid (\text{fst}(\text{coupon_collector}(n + 1) s)) \geq a\} \leq \left(\frac{n+1}{a} \right) \left(\sum_{i=0}^{n+1} \frac{1}{(i+1)} \right)$$

$$\vdash \forall n a. 0 < a \Rightarrow \mathbb{P} \{s \mid \text{abs}((\text{fst}(\text{coupon_collector}(n + 1) s)) - \text{expec}(\text{coupon_collector}(n + 1))) \geq a\} \leq \left(\frac{(n+1)^2}{a^2} \right) \left(\sum_{i=0}^{n+1} \frac{1}{(i+1)^2} \right)$$

where `expec` and `abs` represent the higher-order-logic functions for expectation and absolute functions, respectively.

The first theorem gives the expectation of the Coupon Collector's problem, while the next two correspond to the tail distribution bounds of the Coupon Collector's problem using Markov and Chebyshev's inequalities, respectively. The above results exactly match the results of the analysis based on paper-and-pencil proof techniques [34] and are thus 100 % precise, which is a novelty that cannot be achieved, to the best of our knowledge, by any existing computer based probabilistic analysis tool. The results were obtained by building on top of the formally verified linearity of expectation and variance properties and the Markov and Chebyshev's inequalities and thus the proof script corresponding to the formalization and verification of the Coupon Collector's problem translated to approximately 1000 lines of code and the analysis took around 100 man-hours.

3.2 Performance Analysis of the Stop-and-Wait Protocol

The Stop-and-Wait protocol [29] utilizes the principles of error detection and retransmission to ensure reliable communication between computers. The main idea is that the transmitter keeps on transmitting a data packet unless and until it receives a valid acknowledgement (ACK) of its reception from the receiver.

The message delay of a communication protocol is the most widely used performance metric. In the case of the Stop-and-Wait protocol, the message delay is an unpredictable quantity since it depends on the random behavior of channel noise and thus probabilistic techniques are utilized for its assessment.

The Stop-and-Wait protocol is a classical example of a real-time system and thus involves a subtle interaction of a number of distributed processes. The behavior of these processes over time may be specified by higher-order-logic predicates on positive integers [5] that represent the ticks of a clock counting physical time in any appropriate units, e.g., nanoseconds. The granularity of the clock's tick is believed to be chosen in such a way that it is sufficiently fine to detect properties of interest. Using this approach, the Stop-and-Wait protocol can be formalized in higher-order logic as a logical conjunction of six processes (Data Transmission, Data Channel, Data Reception, ACK Transmission, ACK Channel, ACK Reception) and some initial conditions [22]. The random component in the Stop-and-Wait protocol is channel noise, which can be expressed using the formal Bernoulli random variable function.

The next step is to utilize the formal model of the Stop-and-Wait protocol to formally verify the average message delay relation of the Stop-and-Wait protocol, for the case when the processing time of a message is equal to 1, as the following theorem [22].

$$\begin{aligned}
&\vdash \forall \text{ source sink rem s i r ws sn ackty maxP abort dataS dataR} \\
&\quad \text{ackS ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag} \\
&\quad \text{bseqt bseq p.} \\
&\quad \text{STOP_WAIT_NOISY source sink rem s i r ws sn ackty maxP abort} \\
&\quad \text{dataS dataR ackS ackR d tprop dtout dtf dta tf} \\
&\quad \text{ack_msg ta tout rec_flag bseqt bseq} \wedge \\
&\quad \text{LIVE_ASSUMPTION abort} \wedge 0 \leq p \wedge p < 1 \wedge \neg \text{NULL source} \wedge \\
&\quad \text{tprop} + 1 + \text{ta} + \text{tprop} + 1 \leq \text{tout} \Rightarrow \\
&\quad (\text{expec} (\text{DELAY_STOP_WAIT_NOISY rem source bseqt}) = \\
&\quad ((\text{tf} + \text{tout})p/(1-p) + (\text{tf} + \text{tprop} + 1 + \text{ta} + \text{tprop} + 1)))
\end{aligned}$$

The antecedent of the above theorem contains the formal definition of the Stop-and-Wait protocol under noisy channel conditions (STOP_WAIT_NOISY), liveness constraints and the fact that the probability of channel error p is bounded in the real interval $[0, 1)$. The function DELAY_STOP_WAIT_NOISY formally represents the delay of the Stop-and-Wait protocol and thus the left-hand-side of the conclusion of the above theorem represents the average delay of the Stop-and-Wait protocol. On the right-hand-side of the conclusion of the above theorem, the variables tf , ta , $tprop$ and $tout$ denote the time delays associated with data transmission, ACK transmission, message propagation, message processing and time-out delays, respectively. More details on the variables used above and the proof sketch of this theorem can be found in [22].

It is important to note here that the relation for the average delay of a Stop-and-Wait protocol is not new. In fact its existence dates back to the early days of introduction of the Stop-and-Wait protocol. However, it has always been verified

using theoretical paper-and-pencil proof techniques, e.g. [29]. Whereas, the analysis described above is based on mechanical verification using a theorem prover, which is a superior approach to both paper-and-pencil proofs and simulation based analysis techniques. To the best of our knowledge, it is the first time that a statistical property for a real-time system has been formally verified.

3.3 Reliability Analysis of Reconfigurable Memory Arrays

Reconfigurable memory arrays with spare rows and columns are quite frequently used as reliable data storage components in present age System-on-Chips. The spare memory rows and columns can be utilized to automatically replace rows or columns that are found to contain a cell fault, such as stuck-at or coupling fault [33]. One of the biggest design challenges is to estimate, prior to the actual fabrication process, the right number of these spare rows and spare columns for meeting the reliability specifications. Since the fault occurrence in a memory cell is an unpredictable event, probabilistic techniques are utilized to estimate the number of spare rows and columns [39].

The analysis for this example is done by formally expressing a fault model for reconfigurable memory arrays in higher-order logic [23]. The formalization utilizes the precise Binomial random variable function to express the random components in the model. This model is then utilized to express and verify statistical properties, such as expectation and variance of the number of faults in terms of memory array and spare rows and columns sizes, as higher-order logic theorems. Finally, this formal statistical information is built upon to formally verify repairability and irreparability conditions for a square memory array with stuck-at and coupling faults that are independent and identically distributed. For example, the repairability condition for a square $n \times n$ memory array, with $a \times n$ spare rows and $b \times n$ spare columns, has been verified as the following higher-order-logic theorem.

$$\begin{aligned} \vdash \forall a \ b \ w. \ (0 \leq a) \wedge (a \leq 1) \wedge (0 \leq b) \wedge (b \leq 1) \wedge \\ (c1 + c2 = a + b) \wedge (1 < n) \wedge (\forall n. (0 < w(n)) \wedge \\ (w(n) < (\min c1\sqrt{n} \ c2\sqrt{n}))) \wedge (\lim (\lambda n. \frac{1}{w(n)}) = 0) \Rightarrow \\ (\lim (\lambda n. \mathbb{P}\{s \mid (\text{fst}(\text{num_of_faults } n \ c1 \ c2 \ w \ s)) \leq (a+b)n\}) = 1)) \end{aligned}$$

where $\lim M$ represents the higher-order logic formalization of the limit of a real sequence M (i.e., $\lim M = \lim_{n \rightarrow \infty} M(n)$) [15]. The first four assumptions in the above theorem ensure that the fractions a and b are bounded by the interval $[0, 1]$ as the number of spares can never exceed the number of original rows. The relationship between a and b with two arbitrary real numbers $c1$ and $c2$ is given in the fifth assumption. The precondition $1 < n$ has been used in order to ensure that the given memory array has more than one cell. The next two assumptions are about the real sequence w and basically provides its upper and lower bounds. These bounds have been used in order to prevent the stuck-at and coupling fault occurrence probabilities p_s and p_c from falling outside their allowed interval $[0, 1]$ [23]. The last assumption $(\lim(\lambda n. \frac{1}{w(n)}) = 0)$ has been

added to formally represent the intrinsic characteristic of *real* sequence w that it tends to infinity as its *natural* argument becomes very very large. The theorem proves that under these assumptions a very large square memory array is almost always repairable (with probability 1) since the probability that the number of faults is less than the number of spare rows and columns is 1.

The above theorem leads to the accurate estimation of the number of spare rows and columns required for reliable operation against stuck-at and coupling faults of any reconfigurable memory array without any CPU time constraints. The distinguishing feature of this analysis is its generic nature as our theorems are verified for all sizes of memories $n \times n$ with any number of spare rows (axn) or columns (bxn).

This case study clearly demonstrate the effectiveness of theorem proving based probabilistic analysis. Due to the formal nature of the models, the high expressiveness of higher-order logic, and the inherent soundness of theorem proving, we have been able to verify generic properties of interest that are valid for any given memory array with 100% precision; a novelty which is not available in simulation. Similarly, we have been able to formally analyze properties that cannot be handled by model checking. The proposed approach is also superior to the paper-and-pencil proof methods [39] in a way as the chances of making human errors, missing critical assumptions and proving wrongful statements are almost nil since all proof steps are applied within the sound core of a higher-order-logic theorem prover. These additional benefits come at the cost of the time and effort spent, while formalizing the memory array and formally reasoning about its properties. But, the fact that we were building on top of already verified probability theory foundations, described in Section 2, helped significantly in this regard as the memory analysis only consumed approximately 250 man-hours and 3500 lines of proof code.

3.4 Round-Off Error Analysis in Floating-Point Representation

Algorithms involving floating-point numbers are extensively used these days in almost all digital equipment ranging from computer and digital processing to telecommunication systems. Due to their complexity and wide spread usage in safety critical domains, formal methods are generally preferred over traditional testing to ensure correctness of floating-point algorithms. A classical work in this regard is Harrison's error analysis of floating-point arithmetic in higher-order logic [14]. Harrison presents a formalization of floating point numbers, verification of upper bounds on the error in representing a real number in floating-point and the error in floating-point arithmetic operations. Even though this analysis is very useful in identifying the worst case conditions, it does not reflect upon typical or average errors. In fact, the assumed worst case conditions rarely occur in practice. So the error analysis, based under these worst-case conditions can improperly suggest that the performance of the algorithm is poor.

In paper-and-pencil analyses, probabilistic techniques are thus utilized in the error analysis of floating-point algorithms [41]. The main idea behind this probabilistic approach is to model the error in a single floating-point number by an

appropriate random variable and utilize this information to judge the expected value of error while representing a real number in floating-point system. This expected value of error can then be used to find the expected value of error in different floating-point arithmetic operations.

The above mentioned probabilistic analysis involves reasoning about the expectation value of a continuous random variable, since the error between a real number and its corresponding floating-point representation is continuous in nature. Thus, our proposed infrastructure can be directly utilized to conduct such analysis, something that to the best of our knowledge was not possible before.

We built upon Harrison's error bounds for floating-point representations of *big* ($|x| \in [2^k, 2^{k+1})$), *small* ($|x| \in [\frac{1}{2^{k+1}}, \frac{1}{2^k}] : k < 126$), and *tiny* ($|x| \in [0, \frac{1}{2^{126}}]$) real numbers [14]. The error is defined as the difference between the real value of the floating-point representation and the actual value of the corresponding real number ($\text{error}(x) = \text{float}(x) - x$), with round-to-nearest rounding mode. Based on this definition, upper bounds on the absolute value of error are verified to be equal to $\frac{2^k}{2^{24}}$, $\frac{1}{2^{k+1}2^{24}}$ and $\frac{1}{2^{150}}$, for the three cases above, respectively.

Assuming any value of error to be equally likely [41], we constructed formal probabilistic models for representing the above mentioned rounding errors using Uniform random variables defined in the intervals $[0, \frac{2^k}{2^{24}}]$, $[0, \frac{1}{2^{k+1}2^{24}}]$ and $[0, \frac{1}{2^{150}}]$, respectively. The formally verified expectation of the Uniform random variable [17] was then used to verify the expectation values of these floating-point errors using a theorem prover.

$$\begin{aligned} \vdash \forall k \ x. \quad & (\text{expec}(\text{uniform_rv } 0 \ \frac{2^k}{2^{24}}) = \frac{2^{k-1}}{2^{24}}) \wedge \\ & (\text{expec}(\text{uniform_rv } 0 \ \frac{1}{2^{k+1}2^{24}}) = \frac{1}{2^{k+1}2^{25}}) \wedge \\ & (\text{expec}(\text{uniform_rv } 0 \ \frac{1}{2^{150}}) = \frac{1}{2^{151}}) \end{aligned}$$

This theorem plays a vital role in the statistical error analysis of floating-point arithmetic. Based on these averages of error in a single floating-point number, the average errors in floating point operations, like addition and multiplication, that involve multiple floating-point numbers, can be evaluated. Similarly, this information can be utilized in conducting the statistical error analysis of digital signal processing (DSP) systems by building on top of the DSP verification framework in higher-order logic [1], which does not include any probabilistic considerations.

The verification of the above result was automatic as the verified theorem is a direct consequence of the expectation property of the continuous Uniform random variable, which is available in the proposed framework. This fact clearly demonstrates the usefulness of the proposed infrastructure that calls for formalizing and verifying the fundamental concepts of probability theory in order to facilitate the formal probabilistic analysis of real-world systems.

4 Related Work

Due to the vast application domain of probability in safety-critical applications, many researchers around the world are trying to improve the quality of computer

based probabilistic analysis. The ultimate goal is to come up with a formal probabilistic analysis framework that includes robust and accurate analysis methods, has the ability to perform analysis for large-scale problems and is easy to use. In this section, we provide a brief account of the state-of-the-art in this field.

Probabilistic model checking [3,37] is one of the commonly used formal probabilistic analysis technique. It involves the construction of a precise state-based mathematical model of the given probabilistic system, which is then subjected to exhaustive analysis to formally verify if it satisfies a set of formally represented probabilistic properties. Numerous probabilistic model checking algorithms and methodologies have been proposed in the open literature, e.g., [8,35], and based on these algorithms, a number of tools have been developed, e.g., PRISM [36,28], E \vdash MC² [24], Rapture [27] and VESTA [38]. Besides the accuracy of the results, the most promising feature of probabilistic model checking is the ability to perform the analysis automatically. On the other hand, it is limited to systems that can only be expressed as probabilistic finite state machines. Another major limitation of the probabilistic model checking approach is state space explosion [4]. The state space of a probabilistic system can be very large, or sometimes even infinite. Thus, at the outset, it is impossible to explore the entire state space with limited resources of time and memory. Similarly, we cannot reason about mathematical expressions in probabilistic model checking. This is a big limitation as far as reasoning about PMF, CDF or expectation or variance of a random behavior is concerned, which are basically functions of the range of a random variable. Thus, the probabilistic model checking approach, even though is capable of providing exact solutions, is quite limited in terms of handling a variety of probabilistic analysis problems. Whereas higher-order-logic theorem proving is capable of overcoming all the above mentioned problems but at a significant cost of user interaction.

Besides higher-order-logic theorem proving and model checking, another formal approach that is capable of providing exact solutions to probabilistic properties is proof based languages that have been extended with probabilistic choice. The main idea behind this approach is to use refinement or utilize the expectations (or probabilistic invariants) to reason about probabilistic properties. Many formalisms have been extended with probabilistic choice, e.g., B (pB) [25], Hoare logic (pL) [7], Z [40] and Event-B [13]. Besides their precision, another major benefit of these approaches is their automatic or semi-automatic nature. Out of these formalisms, Probabilistic B (pB) is one of the more commonly used mainly because of its ability to obtain algebraic relationships between the different parameters of the model and of the design requirements. On the other hand, even though some efforts have been reported, e.g. [2], it is not mature enough to model and reason about random components of the system that involve all different kinds of continuous probability distributions. Similarly, all of the above mentioned formalisms cannot be used to reason about generic mathematical expressions for probabilistic or statistical properties, such as PMF, CDF, expectation or variance, due to their limited expressiveness, which is not an issue with the proposed higher-order-logic theorem proving based approach. For

example, the Chaums Dining Cryptographers (DC) problem, a well-known security problem, has been recently analyzed formally using the pB approach and the mean and variance of utterances have been computed for only a finite number of DC nets and specific set of fixed values for coin fairness [32]. By contrast, higher-order-logic theorem proving can be utilized to prove generic mathematical expressions, for the mean and variance characteristics of interest, that are quantified over n cryptographers and all values of coin fairness.

5 Conclusions

This paper provides a brief overview of the existing capabilities of higher-order-logic theorem proving based probabilistic analysis approach. The main idea behind this emerging trend is to use random variables formalized in higher-order logic to model systems, which contain some sort of randomness, and to verify the corresponding probabilistic and statistical properties in a theorem prover. Because of the formal nature of the models, the analysis is 100% accurate and due to the high expressive nature of higher-order logic a wider range of systems can be analyzed. Thus, the theorem proving based probabilistic analysis approach can prove to be very useful for the performance and reliability optimization of safety critical and highly sensitive engineering and scientific applications.

The proposed approach has been illustrated by providing the formal probabilistic analysis of four real-world systems. The analysis results exactly matched the results obtained by paper-and-pencil proof techniques and are thus 100 % precise. The successful handling of these diverse problems by the proposed approach clearly demonstrates its feasibility for real-world probabilistic analysis issues. In all these applications, we have been able to formally reason about real valued expressions of probabilistic or statistical properties of systems, something that cannot be achieved by probabilistic model checking or probabilistic language based approaches.

All higher-order-logic formalizations, presented in this paper, have been done using the HOL theorem prover [12]. The main reason being that the foundational measure and probability theories were formalized in HOL first [26] and then the rest of the infrastructure kept building upon that. Though, it is important to note that the presented methodologies and framework are not specific to the HOL theorem prover and can be adapted to any other higher-order-logic theorem prover, such as Isabelle, Coq or PVS, as well.

The theorem proving based probabilistic analysis framework can no way be considered to be mature enough to be able to handle all kind of problems. There are many open research issues that need to be resolved in order to achieve this goal. To name a few, first of all the capability to reason about multiple continuous random variables is not available. Secondly, some of the most commonly used random variables, like the Normal random variable, have not been formalized so far. Thirdly, no formalization related to stochastic processes and Markov chains is available, which are widely used concepts in probabilistic analysis.

References

1. Akbarpour, B., Tahar, S.: An Approach for the Formal Verification of DSP Designs using Theorem Proving. *IEEE Transactions on CAD of Integrated Circuits and Systems* 25(8), 1141–1457 (2006)
2. Andrews, Z.: Towards a Stochastic Event B for Designing Dependable Systems. In: *Proc. Workshop on Quantitative Formal Methods: Theory and Applications*, Eindhoven, The Netherlands (November 2009)
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model Checking Algorithms for Continuous time Markov Chains. *IEEE Transactions on Software Engineering* 29(4), 524–541 (2003)
4. Baier, C., Katoen, J.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
5. Cardell-Oliver, R.: *The Formal Verification of Hard Real-time Systems*. PhD Thesis, University of Cambridge, UK (1992)
6. Coble, A.: *Anonymity, Information, and Machine-Assisted Proof*. Ph.D Thesis, University of Cambridge, UK (2009)
7. Corin, R.J., Den Hartog, J.I.: A Probabilistic Hoare-style Logic for Game-based Cryptographic Proofs. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) *ICALP 2006*. LNCS, vol. 4052, pp. 252–263. Springer, Heidelberg (2006)
8. de Alfaro, L.: *Formal Verification of Probabilistic Systems*. PhD Thesis, Stanford University, Stanford, USA (1997)
9. Devroye, L.: *Non-Uniform Random Variate Generation*. Springer, Heidelberg (1986)
10. Galambos, J.: *Advanced Probability Theory*. Marcel Dekker Inc., New York (1995)
11. Gordon, M.J.C.: Mechanizing Programming Logics in Higher-Order Logic. In: *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 387–439. Springer, Heidelberg (1989)
12. Gordon, M.J.C., Melham, T.F.: *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge (1993)
13. Hallerstede, S., Hoang, T.S.: Qualitative Probabilistic Modelling in Event-B. In: Davies, J., Gibbons, J. (eds.) *IFM 2007*. LNCS, vol. 4591, pp. 293–312. Springer, Heidelberg (2007)
14. Harrison, J.: *Floating Point Verification in HOL Light: The Exponential Function*. Technical Report 428, Computing Laboratory, University of Cambridge, UK (1997)
15. Harrison, J.: *Theorem Proving with the Real Numbers*. Springer, Heidelberg (1998)
16. Harrison, J.: *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge (2009)
17. Hasan, O., Abbasi, N., Akbarpour, B., Tahar, S., Akbarpour, R.: Formal reasoning about expectation properties for continuous random variables. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009: Formal Methods*. LNCS, vol. 5850, pp. 435–450. Springer, Heidelberg (2009)
18. Hasan, O., Tahar, S.: Formalization of the Continuous Probability Distributions. In: Pfenning, F. (ed.) *CADE 2007*. LNCS (LNAI), vol. 4603, pp. 3–18. Springer, Heidelberg (2007)
19. Hasan, O., Tahar, S.: Formalization of the Standard Uniform Random Variable. *Theoretical Computer Science* 382(1), 71–83 (2007)
20. Hasan, O., Tahar, S.: Using Theorem Proving to Verify Expectation and Variance for Discrete Random Variables. *Journal of Automated Reasoning* 41(3–4), 295–323 (2008)

21. Hasan, O., Tahar, S.: Formal Verification of Tail Distribution Bounds in the HOL Theorem Prover. *Mathematical Methods in the Applied Sciences* 32(4), 480–504 (2009)
22. Hasan, O., Tahar, S.: Performance Analysis and Functional Verification of the Stop-and-Wait Protocol in HOL. *Journal of Automated Reasoning* 42(1), 1–33 (2009)
23. Hasan, O., Tahar, S., Abbasi, N.: Formal Reliability Analysis using Theorem Proving. *IEEE Transactions on Computers* (2009), doi:10.1109/TC.2009.165
24. Hermanns, H., Katoen, J.P., Meyer-Kayser, J., Siegle, M.: A Markov Chain Model Checker. In: Schwartzbach, M.I., Graf, S. (eds.) *TACAS 2000*. LNCS, vol. 1785, pp. 347–362. Springer, Heidelberg (2000)
25. Hoang, T.S.: The Development of a Probabilistic B Method and a Supporting Toolkit. PhD Thesis, The University of New South Wales, UK (2005)
26. Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD Thesis, University of Cambridge, UK (2002)
27. Jeannot, B., Argenio, P.D., Larsen, K.: Rapture: A Tool for Verifying Markov Decision Processes. In: *Tools Day, 13th Int. Conf. Concurrency Theory*, Brno, Czech Republic (2002)
28. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative Analysis with the Probabilistic Model Checker PRISM. *Electronic Notes in Theoretical Computer Science* 153(2), 5–31 (2005)
29. Leon-Garcia, A., Widjaja, I.: *Communication Networks: Fundamental Concepts and Key Architectures*. McGraw-Hill, New York (2004)
30. Levine, A.: *Theory of Probability*. Addison-Wesley series in Behavioral Science, Quantitative Methods (1971)
31. MacKay, D.J.C.: Introduction to Monte Carlo Methods. In: *Learning in Graphical Models*, NATO Science Series, pp. 175–204. Kluwer Academic Press, Dordrecht (1998)
32. McIver, A., Meinicke, L., Morgan, C.: Security, Probability and Nearly Fair Coins in the Cryptographers’ Café. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009: Formal Methods*. LNCS, vol. 5850, pp. 41–71. Springer, Heidelberg (2009)
33. Miczo, A.: *Digital Logic Testing and Simulation*. Wiley Interscience, Hoboken (2003)
34. Mitzenmacher, M., Upfal, E.: *Probability and Computing*. Cambridge University Press, Cambridge (2005)
35. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic System. PhD Thesis, University of Birmingham, UK (2001)
36. PRISM (2008), <http://www.cs.bham.ac.uk/~dxp/prism>
37. Rutten, J., Kwiatkowska, M., Norman, G., Parker, D.: *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*. CRM Monograph Series, vol. 23. American Mathematical Society (2004)
38. Sen, K., Viswanathan, M., Agha, G.: VESTA: A Statistical Model-Checker and Analyzer for Probabilistic Systems. In: *Proc. IEEE International Conference on the Quantitative Evaluation of Systems*, pp. 251–252 (2005)
39. Shi, W., Fuchs, W.K.: Probabilistic Analysis and Algorithms for Reconfiguration of Memory Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(9), 1153–1160 (1992)
40. White, N.: Probabilistic Specification and Refinement. Masters Thesis, Oxford University, UK (1996)
41. Widrow, B.: Statistical Analysis of Amplitude-quantized Sampled Data Systems. *AIEE Transactions on Applications and Industry* 81, 555–568 (1961)
42. Yates, R.D., Goodman, D.J.: *Probability and Stochastic Processes: A Friendly Introduction for Electrical and Computer Engineers*. Wiley, Chichester (2005)

Synchronous Message Passing and Semaphores: An Equivalence Proof

Iain Craig¹ and Egon Börger²

¹ Visiting Researcher, Department of Computer Science, University of York
(Correspondence address: Flat 4, 34 Sherbourne Road, Birmingham, UK)

`idcraig@talktalk.net`

² Dip. di Informatica, Università di Pisa, Italy

`boerger@di.unipi.it`

On sabbatical leave at CS Department, ETH Zürich

Abstract. A natural encoding of synchronous message exchange with direct wait-control is proved to be equivalent in a distributed environment to a refinement which uses semaphores to implement wait control. The proof uses a most general scheduler, which is left as abstract and assumed to satisfy a few realistic, explicitly stated assumptions. We hope to provide a scheme that can be implemented by current theorem provers.

1 Introduction

This paper is part of an endeavor a) to rigorously model kernels of small but real-life operating systems (OS) at a high level of abstraction, b) to verify mathematically the major OS properties of interest in the models and c) to refine the models in a provably correct way by a series of steps to implementation code. So that the refinement steps reflect the major design decisions, which lead from the abstract models to executable code, and to make their correctness mathematically controllable, we use algorithmic (also called operational) models. Thus we deviate from the axiomatic (purely declarative) point of view underlying the Z-based formal models of operating system kernels in the two recent books [5,6], which are, however, our starting point. We are careful to ensure that our models, technically speaking, Abstract State Machines [4], whose semantic foundation justifies our considering them an accurate version of pseudo-code, are understandable by programmers without further training in formal methods, so that they can be effectively used in practice by designers and teachers for a rigorous analysis of OS system functionalities.

In a first paper [3] we presented the method, focussing on modeling the behavior of the clock process, more specifically the clock interrupt service routine, which interacts with a priority-based scheduler of device, system and user processes. The paper uses an abstract form of synchronous message passing. In this paper we show that a natural high-level specification of synchronous message exchange which uses a direct wait control mechanism has a provably correct refinement (in fact is equivalent to it) which uses semaphores to implement the wait control.

The equivalence of a direct implementation of synchronous message passing and of one using semaphores is often cited in the literature (e.g., [7]). The purpose of this paper is to illustrate how one can turn this equivalence claim, in a way which supports the intuitive operational understanding of the involved concepts, into a precise mathematical statement and prove it, with reasonable generality, that is, in terms of a faithful abstract model and its refinement.

A natural direct implementation of the synchronous message-passing mechanism uses a queue of processes, whose running mode is controlled by explicit intervention of the scheduler (Sect. 3¹). An alternative implementation uses a semaphore structure, which allows one to separate the issues related to process control (handing the permission to processes to run in a critical section) from the message transfer functionality (Sect. 5). In a specification which already includes semaphores as a component, using semaphores in the refinement of an implementation that uses direct process control makes sense because it reduces the complexity of individual code modules as well as the related verification tasks. Proceeding this way is an example of the use of standard components for stepwise refinement in a sense similar to the library types often provided with object-oriented programming languages.

We collect, in Sect. 2, the minimal assumptions that must be guaranteed for the underlying scheduling mechanism. We exhibit these assumptions to clarify the sense in which our model and its verification apply to any OS scheduling policy encountered in real life. In Sect. 4, we specify the underlying semaphore concept. The definitions in these two sections are re-elaborations of those given in [3] to which we refer for their motivation. The equivalence proof as presented in Sect. 6 applies only to the uniprocessor case.

2 Scheduling

Both semaphores and synchronous message passing require a scheduler and its associated operations. In this section, we briefly provide this background, formulating the minimal assumptions needed for the equivalence proof.

In the uniprocessor case, an operating system comes with a notion of a unique currently-executing process, which we denote by *currrp*. It is an element of a dynamic set *Process* of processes, or it is the *idleProcess*. The *currrp* process can be thought of as the only one with $status(currrp) = run$ (read: instruction pointer *ip* pointing into its code). The scheduler selects it from a queue, *readyq*, of processes whose *status* is *ready* to execute, possibly respecting some predefined priorities (e.g. selecting device processes before system processes and those before user processes, see [3]). We make no specific assumptions as to how *readyqueue* is ordered, so that any fair scheduling policy (as specified by the assumptions stated below) is covered by our equivalence proof.

Only two scheduler operations are required, the exact nature of which can remain abstract. One operation, defined by `SUSPENDCURR` below, suspends a

¹ One reviewer suggested to use instead the “await” blocking rule constructor defined in [1] for concurrent ASMs. We want to keep the construction here as elementary as possible to ease the implementation of the equivalence theorem by theorem provers.

sender process, *currp*, when it has sent a message to a *destination* process. The other operation, defined by $\text{MAKEREADY}(src)$ below, makes the sender (also called source) process, *src*, ready again when the message has been communicated to (read: received by) the receiver process, *dest*.

The main effect of SUSPENDCURR is to update *currp* to the next process, which is selected by the scheduler from the *readyq*. We denote the selected element by $head(readyq)$ to convey the idea that the selection is made based upon the order of the queue; the function $head$ is defined by the scheduler and need not be further specified here. In addition, the selected process is deleted from the *readyq* and assigned the *status* run. Also, the current *state* of the previous current process must be saved and the *state* of the new current process must be restored. Since these state update operations are orthogonal to the mere scheduling part of SUSPENDCURR , we denote them here by two abstract machines that are not further specified: SAVESTATE and RESTORESTATE . Their execution is considered to be atomic, which amounts to using a locking mechanism whose specifics we can ignore here without loss of generality.² This leads to the following definition, where we use an abstract machine DEQUEUE to denote the deletion of an element that has been chosen from *readyq*.³

$$\begin{aligned} \text{SUSPENDCURR} = & \\ & \text{let } h = head(readyq) \text{ in} \\ & \quad \text{UPDATESTATE}(currp, h) \\ & \quad currp := h \\ & \quad status(h) := \text{run} \\ & \quad \text{DEQUEUE}(h, readyq) \\ & \text{where} \\ & \quad \text{UPDATESTATE}(p, p') = \\ & \quad \quad \text{SAVESTATE}(p) \\ & \quad \quad \text{RESTORESTATE}(p') \end{aligned}$$

The effect of $\text{MAKEREADY}(p)$ is to $\text{ENQUEUE}(p, readyq)$ and to update its *status* to *ready*. As a result of executing $\text{MAKEREADY}(p)$, process *p* becomes a candidate for the scheduler's next choice for updating *currp*. Here, we do not specify the order of *readyq*, so ENQUEUE is left as an abstract machine, which is assumed to respect the intended *readyq* order when inserting an element.

$$\begin{aligned} \text{MAKEREADY}(p) = & \\ & \text{ENQUEUE}(p, readyq) \\ & status(p) := \text{ready} \end{aligned}$$

² The locking effect is captured here by the atomicity of ASM steps, see [3].

³ We remind the reader that due to the simultaneous parallel execution of all updates in an ASM rule, *currp* on the right hand side of $:=$ denotes the *previous* value and *currp* on the left hand side of $:=$ the newly assigned value.

⁴ In the special case of an empty *readyq*, the well known *idleProcess* is chosen to become the new *currp*. This case is of no importance here, so that, without loss of generality, we omit it in this rule to streamline the exposition.

3 Directly Controlled Synchronous Message Passing

In both this section and section Sect. 5, we assume an abstract set, Msg , of messages as given. We analyze synchronous message passing between processes as triples of three steps, each described using abstract machine components as follows:

- STARTSENDING out a message m from a source src to a *destination*
 - by recording m , say in a location $outbox(src)$, for the transfer and WAIT for the synchronization with *dest*;
- STARTSYNCHRONIZING with a source process src and WAIT for the synchronization with *dest* to become effective;
- Upon synchronization bw src and *dest*:
 - DELIVERMSG to the *destination* process, say by transfer of the value of $outbox(src)$ into a location $inbox(dest)$
 - TERMINATESYNCHRONIZATION.

For both the direct implementation and the one using semaphores, we define a version for each of the named components. Note that STARTSEND&WAIT and STARTSYNC&WAIT can be executed in any order, depending on whether the sender or the receiver initiates the message passing.

3.1 The MSGPASS_{Ctl} Machine (Ground Model)

A natural way directly to control synchronous message passing in a distributed environment consists of making sender and receiver wait for each other until each one has ‘seen’ the other. To describe the waiting phase, one can use an extension of the scheduler’s *status* control. To do this, on the sender side, execution of STARTSEND&WAIT_{Ctl}($m, src, dest$) will RECORDFORTRANSFER(m, src) the message m , suspend the sender src (read: the currently executing process), switch its *status* to *sndr* and ENQUEUE(src) in a collection, $wtstndr(dest)$, of senders waiting for their message to be received by the *destination* process. The operation STARTSYNC&WAIT_{Ctl} of *dest* consists of, first, testing whether there is a message waiting to be received. If $wtstndr(dest)$ is empty, *dest* switches to receiver *status* *rcvr* and suspends itself, whereafter only a sender move can MAKEREADY(*dest*) again. If $wtstndr(dest)$ is found not to be empty (any more), both parties are synchronized. This triggers the receiver’s second move PASSMSG_{Ctl} to DELIVERMSG in the receiver side and to terminate the synchronization (i.e. MAKEREADY the sender and delete it from $wtstndr(dest)$).

The preceding protocol description is formalized by the following definitions. We deliberately leave the ENQUEUE and DEQUEUE operations abstract; their instantiation depends on the scheduling policy⁵.

⁵ Note, however, that for our equivalence proof in Sect. 6, the uniprocessor assumption is used. It guarantees that at each moment in MSGPASS_{Ctl}, at most one ENQUEUE($src, wtstndr(dest)$) move is made (as part of a STARTSEND&WAIT_{Ctl}($m, src, dest$) move) and, in MSGPASS_{Sema}, at most one corresponding WAITSEMA($insema(dest)$) move (as part of a STARTSEND&WAIT_{Sema}($m, src, dest$) move—see the definition of SEND_{Sema}($m, src, dest$), below).

```

SENDCtl(m, src, dest) = STARTSEND&WAITCtl(m, src, dest) where
  STARTSEND&WAITCtl(m, src, dest) =
    RECORDFORTRANSFER(m, src)
    status(src) := sndr
    ENQUEUE(src, wtsndr(dest))
    if status(dest) = rcvr then MAKEREADY(dest)
    SUSPENDCURR
  RECORDFORTRANSFER(m, src) = (outbox(src) := m)6

```

For the definition of the submachine PASSMSG_{Ctl} of RECEIVE_{Ctl} , it remains to decide whether the protocol should simultaneously permit multiple senders to $\text{STARTSEND\&WAIT}_{Ctl}$ a message to the same *destination* process and in the positive—the more general—case, whether, and possibly how, such sender processes should be ordered in $\text{wtsndr}(\text{dest})$. This decision influences the property that can be proved in the equivalence theorem. Since the usual model of semaphores is that they work with queues, we assume in the following that $\text{wtsndr}(\text{dest})$ is a (possibly priority) queue.

RECEIVE_{Ctl} splits into a step $\text{STARTSYNC\&WAIT}_{Ctl}$ followed by PASSMSG_{Ctl} . To formalize this sequentiality in the context of simultaneous parallel execution of ASM rules, we use the interruptable version of sequential execution introduced for ASMs in [3]7. It is borrowed from the traditional FSM-control mechanism and denoted **step**8. Since $\text{wtsndr}(\text{dest})$ is treated as a queue, we again use a ‘head’ function, denoted *hd*, to select (possibly in a priority-based manner) the next element to be DEQUEUED from $\text{wtsndr}(\text{dest})$.

```

RECEIVECtl(dest) =
  STARTSYNC&WAITCtl(dest) step PASSMSGCtl(dest)
where
  STARTSYNC&WAITCtl(p) =
    if wtsndr(p) = ∅ then

```

⁶ Since $\text{STARTSEND\&WAIT}_{Ctl}$ suspends the sender, there is no buffering of messages at the sender side; i.e. *outbox* is only an internal location for recording a message a sender wants to send in a synchronized fashion, it is not a message box.

⁷ One reviewer observed that given the mutually exclusive guards of $\text{STARTSYNC\&WAIT}_{Ctl}$ and PASSMSG_{Ctl} , we could have avoided here (but not in the analogous situation of SEND_{Sema} in Sect. 5) to use the **step** notation, which forces the receiver to each time perform two steps (the first of which may result in only changing the implicit control state). However, eliminating **step** here would slightly complicate the comparative analysis of sender and receiver moves in the two protocols in Sect. 6, where we exploit the simple correspondence of message exchange triples.

⁸ M_1 **step** M_2 consists of two rules:

```

if ctl_state = 1 then
  M1
  ctl_state := 2

```

and the same with interchanging 1 and 2.

```

    status( $p$ ) := rcvr
    SUSPENDCURR
    PASSMSGCtl( $p$ ) =
    if wtsndr( $p$ )  $\neq \emptyset$  then
      let  $src = hd(wtsndr(p))$  in
        DELIVERMSG( $src, p$ )
        TERMINATESYNC( $scr, p$ )
    DELIVERMSG( $q, p$ ) = ( $inbox(p) := outbox(q)$ )
    TERMINATESYNC( $s, p$ ) =
    DEQUEUE( $s, wtsndr(p)$ )
    MAKEREADY( $s$ )

```

MSGPASS_{Ctl} denotes an asynchronous (also called distributed) ASM where sender agents are equipped with the STARTSEND&WAIT_{Ctl} program and receiver agents with the RECEIVE_{Ctl} program.

3.2 Properties of MSGPASS_{Ctl} Runs

In this section, we justify the definition of MSGPASS_{Ctl} by proving that it specifies a correct synchronous message passing scheme, which, under minimal scheduler assumptions, is also a fair one. First of all we have to clarify what correctness and fairness mean in this context.

Due to the distributed context and depending on the scheduler, it may happen that multiple senders send a message to the same destination process before the latter has had a chance to synchronize with any of the former. This produces an asymmetry between sender and receiver moves: it is decided at the receiver's side which one of the waiting senders is considered next for synchronization. Therefore the waiting phase a sender src enters to send a message is terminated only by the synchronization with the destination process (see the *SndrWait* property in Theorem [1](#)), whereas the receiver $dest$ may enter and terminate various waiting phases (namely for receiving messages from other senders) before entering its waiting phase for src (if at all), which can be terminated only by synchronization with src (see the *RcvrWait* condition in Theorem [1](#)).

To support the reader's general intuitions, we speak in this section of sender (source) or receiver (destination) process to refer to an agent with program STARTSEND&WAIT_{Ctl} or RECEIVE_{Ctl}, respectively. Saying that a process p is scheduled, is a shorthand for $p = currp$. In the following theorem, we first formulate the (intuitive requirements for the) correctness property in general terms and then make them precise and prove the resulting statement for MSGPASS_{Ctl}.

Theorem 1. (*Correctness Property for Message Passing*) *The following properties hold in every run of MSGPASS_{Ctl}.*

SndrWait: *Whenever a source process src is scheduled for sending a message, m , to a receiver process $dest$, it will record the message for transfer and then wait (without making any further move) until it is synchronized with $dest$; it will wait forever if it cannot be synchronized with $dest$. In MSGPASS_{Ctl},*

src is said to be synchronized with *dest* when the receiver process *dest* has performed a `STARTSYNC&WAIT` move and is scheduled, ready to receive a message sent from *src* (i.e. $src = hd(wtsndr(p))$).

RcvrWait: Whenever a process, *dest*, is scheduled for receiving a message, it will wait (i.e. not perform any further move) until it is synchronized with a sender process *src*.

Delivery: Whenever the sender *src* of a message and its receiver *dest* are synchronized, the message is delivered at the receiver process, one at a time and at most once, and the synchronization of the two processes terminates.

To turn the wording of this theorem into a precise statement for `MSGPASSCtrl` so that it can be proved, we use the notion of runs of an asynchronous (multi-agent) ASM. Such a run is defined as a partially ordered set of moves (execution of ASM steps by agents) which a) satisfies an axiomatic coherence condition, b) yields a linear order when restricted to the moves of any single agent (sequentiality condition), c) for each single move of any agent has only finitely many predecessor moves (so-called finite history condition). The coherence condition guarantees that for each finite run segment, all linearizations yield runs with the same final state (see [4] for a detailed definition).

The axiomatic description of the notion of a run of asynchronous (multi-agent) ASMs fits well with the purpose of this paper. Without providing any information on how to construct a class of admissible runs (read: to implement a scheduler), it characterizes the minimal ordering conditions needed so that, when of a run the ordering of some moves of some agents could matter for the outcome, this ordering appears explicitly in the ordering conditions (read: the partial order). Therefore the properties of runs of an arbitrary `MSGPASSCtrl` that we formulate and prove in this section, hold in full generality for every implementation or refinement of `MSGPASSCtrl` by a scheduling policy that extends the partial order (e.g. to a linear order).

Proof. For the rest of this section, whenever we speak of a run we refer to an arbitrarily given, fixed run of `MSGPASSCtrl` with respective sender and receiver agents. The proof of Theorem 11 is by induction on the number of times a process is scheduled for sending or receiving a message in `MSGPASSCtrl` runs.

The first two claims of property *SndrWait* follow from the execution of the first three updates and the `SUSPENDCURR` submachine of `STARTSEND&WAITCtrl`. The *waiting phase of a sender* is characterized here by the following two properties of the sender: it must

- be in *sndr status*—which prevents it from being scheduled because, in order to be scheduled, a process must be in *ready status* and in (usually at the head of) the scheduler’s *readyqueue*;
- be an element of *wtsndr* of some destination process.

The third claim follows because, by the definition of `MSGPASSCtrl`, only a `PASSMSGCtrl` move can `MAKEREADY(src)`; for this to happen, the *destination process* involved, upon being scheduled, 9 must have determined *src* as its next

⁹ Obviously, if *dest* is never scheduled, *src* can never synchronize with it.

waiting sender to synchronize with. Before $\text{PASSMSG}_{C_{tl}}(dest)$ checks this 'readiness to receive from src ' condition (namely by the guard $src = hd(wtsndr(dest))$), by definition of $\text{RECEIVE}_{C_{tl}}(dest)$, the $dest$ process must already have been scheduled to execute its $\text{STARTSYNC\&WAIT}_{C_{tl}}(dest)$ move once (possibly a **skip** move, which does not cause the *status* to change to $rcvr\ status$).

For the proof of the *RcvrWait* property, let us assume that a process, $dest$, is scheduled to receive a message from some sender process and has executed its $\text{STARTSYNC\&WAIT}_{C_{tl}}(dest)$ step. Case 1: there is no waiting sender. Then $dest$ switches to *status rcvr*. At this point, by definition of $\text{MSGPASS}_{C_{tl}}$, only a $\text{STARTSEND\&WAIT}_{C_{tl}}$ move can $\text{MAKEREADY}(dest)$, after which $dest$ can again be scheduled by the scheduler. Case 2: $wtsndr(dest) \neq \emptyset$. Then $dest$ is still scheduled (unless, for some reason, the scheduler deschedules it, possibly rescheduling it again later). In both cases, whenever $dest$ is scheduled, it is ready to receive a message from the sender process, src , it finds at the head of its queue of waiting senders ($src = hd(wtsndr(dest))$). Therefore $dest$ and src are synchronized so that now, and only now, the DELIVERMSG can take place, as part of the $\text{PASSMSG}_{C_{tl}}(dest)$ move, for the message sent by src .

The *Delivery* property holds for the following reason. When two processes, src and $dest$, are synchronized, by definition of $\text{PASSMSG}_{C_{tl}}(dest)$, exactly one execution of $\text{DELIVERMSG}(src, dest)$ is triggered, together with the machine $\text{TERMINATESYNC}(scr, dest)$ which terminates the sender's waiting phase. Note that $dest$, by being scheduled, has just terminated its waiting phase. Note that the one-at-a-time property holds only for the uniprocessor case. \square

Remark on Fairness. Although in the presence of timeouts fairness plays a minor role, fairness issues for $\text{MSGPASS}_{C_{tl}}$ can be incorporated into Theorem [□](#).

An often-studied fairness property is related to overtaking. For example, to guarantee that messages are delivered (if at all) in the order in which their senders present themselves to the receiver (read: enter its *wtsndr* collection), it suffices to declare $wtsndr(p)$ as a queue where the function hd in PASSMSG is the head function. In addition, one has to clarify the order in which senders simultaneously presenting to the same *destination* process are enqueued into $wtsndr(dest)$.

Any fairness property of the underlying scheduler results in a corresponding fairness property of $\text{MSGPASS}_{C_{tl}}$. For example, if the scheduler repeatedly schedules every (active) process, every message sent can be proved eventually to be delivered to the receiver.

4 Semaphores

We borrow the ASM specification of semaphores from [\[3\]](#). For the equivalence proof in Sect. [6](#) binary semaphores, often called *mutual exclusion* or *mutex* semaphores, suffice. They permit at most one process at any time in their critical section, thus bind their counter to one of two values (e.g. 0,1) at any time.

A (counting) semaphore, s , has two locations, a counter and a queue of processes waiting to enter the critical section guarded by s , written $semacount(s)$,

resp. $semaq(s)$ or just $semacount$ resp. $semaq$ if s is clear from the context. The semaphore counter is usually initialized to a value $allowed(s) > 0$, the number of processes simultaneously permitted by s in its associated critical section; the semaphore queue is assumed to be initialized to an empty queue.

Semaphores have two characteristic operations: WAITSEMA, which is executed when trying to access the critical section, and SIGNALSEMA, which is executed when leaving the critical section. WAITSEMA subtracts 1 from $semacount$; SIGNALSEMA adds 1 to it. As long as $semacount$ remains non-negative, nothing else is done when WAITSEMA is scheduled, so that $currprocess$ can enter the critical section. If $semacount$ is negative, at least $allowed$ processes are currently in the critical section. Therefore, in this case, the WAITSEMA move will SUSPENDCURR, add $currp$ to the semaphore queue $semaq$ and put it into $status$ $semawait(s)$. Only a later SIGNALSEMA move can bring the suspended process back to $ready$ status (see below). This leads to the following definition, where we use abstract ENQUEUE and DEQUEUE operations. For the sake of generality we use a *caller* parameter, which we will use below only for $caller = currp$.¹⁰

```

WAITSEMA( $s, caller$ ) =
  let  $newcount = semacount(s) - 1$  in
     $semacount(s) := newcount$ 
  if  $newcount < 0$  then
    ENQUEUE( $caller, semaq(s)$ )
     $status(caller) := semawait(s)$ 
    SUSPENDCURR
WAITSEMA( $s$ ) = WAITSEMA( $s, currp$ )

```

The SIGNAL operation, which is assumed to be performed each time a process leaves the critical section, adds one to $semacount$. If the new value of $semacount$ is not yet positive, $semaq$ still contains some process that is waiting to enter the critical section. Then the process which first entered $semaq$ is removed from the queue and made ready, so that (when scheduled) it can enter the critical section.

```

SIGNALSEMA( $s$ ) =
  let  $newcount = semacount(s) + 1$  in
     $semacount(s) := newcount$ 
  if  $newcount \leq 0$  then
    let  $h = head(semaqueue(s))$  in
      DEQUEUE( $h, semaqueue(s)$ )
      MAKEREADY( $h$ )

```

$SEMA(s)$ is the ASM with rules $WAITSEMA(s, currp)$, $SIGNALSEMA(s)$.

Theorem 2. (*Semaphore Correctness Theorem*) *For every semaphore, s , and every properly initialized $SEMA(s)$ run the following properties hold:*

¹⁰ The operations performed by WAIT and SIGNAL must be atomic, as guaranteed by an ASM step. In the literature, auxiliary LOCK and UNLOCK operations guarantee the atomicity.

Exclusion: *There are never more than $allowed(s)$ processes within the critical section guarded by s .*

Fairness: *If every process that enters the critical section eventually leaves it with a $SIGNALSEMA(s)$ move and if the head function is fair (i.e. eventually selects every element that enters $semaqueue(s)$) and if the scheduler is fair, then every process that makes a $WAITSEMA(s)$ move in an attempt to enter the critical section will eventually enter it.*

Proof. The exclusion property initially holds because the semaphore queue is initialized to empty. Since the initial counter value satisfies $semacount(s) = allowed(s)$ and in each $WAITSEMA(s)$, resp. $SIGNALSEMA(s)$ move, the counter is decremented, resp. incremented, after successive $allowed(s)$ $WAITSEMA(s)$ moves that are not yet followed by a $SIGNALSEMA(s)$ move, every further move $WAITSEMA(s)$ before the next $SIGNALSEMA(s)$ move has $newcount$ as a negative number. Thus it triggers an $ENQUEUE(currp, semaq(s))$ operation, blocking $currp$ from entering the critical section until enough $SIGNALSEMA(s)$ moves turn the content of $semacount(s)$ into a non negative value.

To prove the fairness property, assume that in a given state a process makes a $WAITSEMA(s)$ move. If in this move, $newcount$ is not negative, then the process is not $ENQUEUED$ into the semaphore queue and thus can directly enter the critical section. Otherwise, the process is $ENQUEUED$ into $semaq(s)$ where, by the first two fairness assumptions, it will eventually become the value of $head(semaq(s))$ and thus be made *ready*. Then the scheduler, which is assumed to be fair, will eventually schedule it, so that the process, from the point where it was suspended by its $WAITSEMA(s)$ move, does enter the critical section. \square

5 Semaphore-Based Synchronous Messages

In this section, we define a specification $MSGPASS_{Sema}$ of synchronous message passing using semaphores, which can be viewed as a refinement of $MSGPASS_{CU}$. We start from scratch, thinking about what is needed for a most general solution of the problem, i.e. a semaphore-based solution with minimal assumptions. We then define the meaning of correctness of $MSGPASS_{Sema}$ and provide a direct proof for the correctness theorem. The equivalence theorem in Sect. 6 together with the correctness Theorem 1 for $MSGPASS_{CU}$ provide an alternative correctness proof for $MSGPASS_{Sema}$.

5.1 The $MSGPASS_{Sema}$ Machine

The idea is to refine entering a sender's waiting period, which has to take place as part of a $STARTSEND\&WAIT_{Sema}(m, src, dest)$ move, to a $WAITSEMA$ move for a binary semaphore $insema(dest)$, which for handshaking is signalled by the receiver process $dest$. Thus the receiver controls the permission, given at any time to at most one process src , to write to its $inbox(dest)$. To let the receiver get ready again after its handshaking move, but only after the

DELIVERMSG($src, dest$) move has been made, its $inbox(dest)$ -write-permission move SIGNALSEMA($insema(dest)$) is coupled to a WAITSEMA move for another binary semaphore $outsema(dest)$, which in turn is signalled by the sender process when DELIVERMSG($src, dest$) is performed.

In the following and for the equivalence theorem in Sect. 6, both semaphores $insema(dest)$ and $outsema(dest)$ are assumed to be initialized with 0.

This leads to the following definition of an asynchronous MSGPASS $_{Sema}$ by sender, resp. receiver, agents with rules SEND $_{Sema}$, resp. RECEIVE $_{Sema}$, where SEND $_{Sema}$ is sequentially composed out of a STARTSEND&WAIT $_{Sema}$ and PASSMSG $_{Sema}$ submachine.

$$\begin{aligned} \text{SEND}_{Sema}(m, src, dest) &= \\ &\text{STARTSEND\&WAIT}_{Sema}(m, src, dest) \text{ step } \text{PASSMSG}_{Sema}(src, dest) \\ \text{where} \\ \text{STARTSEND\&WAIT}_{Sema}(m, src, dest) &= \\ &\text{RECORDFORTRANSFER}(m, src)^{\boxed{11}} \\ &\text{WAITSEMA}(insema(dest)) \\ \text{PASSMSG}_{Sema}(src, dest) &= \\ &\text{DELIVERMSG}(src, dest)^{\boxed{12}} \\ &\text{SIGNALSEMA}(outsema(dest)) \end{aligned}$$

$$\text{RECEIVE}_{Sema}(dest) = \text{STARTSYNC\&WAIT}_{Sema}(dest)$$

where

$$\begin{aligned} \text{STARTSYNC\&WAIT}_{Sema}(d) &= \\ &\text{SIGNALSEMA}(insema(d)) \\ &\text{WAIT}(outsema(d)) \end{aligned}$$

Note that a SEND $_{Sema}(m, src, dest)$ move is executed when $currp = src$ and a RECEIVE $_{Sema}(dest)$ move when $currp = dest$.

5.2 Correctness Proof for MSGPASS $_{Sema}$

Theorem 3. (*Correctness Property for Message Passing in MSGPASS $_{Sema}$*) *The message passing correctness properties SndrWait, RcvrWait and Delivery hold in every run of MSGPASS $_{Sema}$.*

Proof. To turn the theorem into a precise statement, it remains to define when exactly two processes src and $dest$ are synchronized in MSGPASS $_{Sema}$ runs. We define this to be true in any one of the two following situations, depending on whether the sender or the receiver starts the attempt to synchronize for a message exchange:

SenderStarts: src has made a STARTSEND&WAIT $_{Sema}(m, src, dest)$ move that ENQUEUED it into the $insema(dest)$ -queue; thereafter $dest$ has made a (first) RECEIVE $_{Sema}(dest)$ move that DEQUEUED src from the $insema(dest)$ -queue and src is again scheduled.

¹¹ RECORDFORTRANSFER is defined as for STARTSEND&WAIT $_{Cu}$.

¹² DELIVERMSG is defined as for RECEIVE $_{Cu}$.

ReceiverStarts: $dest$ has made a $\text{RECEIVE}_{Sema}(dest)$ move which is followed by a (first) $\text{STARTSEND\&WAIT}_{Sema}(m, src, dest)$ move that does not ENQUEUE src into the $insema(dest)$ -queue.

SenderStarts implies that, from its $\text{STARTSEND\&WAIT}_{Sema}(m, src, dest)$ move until reaching the synchronization point, src stayed in the $insema(dest)$ -queue without making any further move. *ReceiverStarts* implies that between the two moves $\text{RECEIVE}_{Sema}(dest)$ and $\text{STARTSEND\&WAIT}_{Sema}(m, src, dest)$, $dest$ has made no move; it will be DEQUEUED from the $outsema(dest)$ -queue by the subsequent src -move $\text{PASSMSG}_{Sema}(src, dest)$. Note that in the *ReceiverStarts* case, after the indicated $\text{STARTSEND\&WAIT}_{Sema}(m, src, dest)$ move src remains scheduled (or will be rescheduled after an interrupt).

From the definition of synchronization and of MSGPASS_{Sema} , the correctness properties *SndrWait*, *RcvrWait* and *Delivery* follow by an induction on MSGPASS_{Sema} runs. \square

Remark. The above algorithm for exchanging messages using semaphores was implemented as a collection of threads in C by the first author. Experiments with this implementation demonstrated that it behaves in a fashion equivalent to synchronous message passing. In the next section we mathematically define and prove this equivalence for the above two specifications.

6 Equivalence Proof

In this section, we formulate what it means (and prove) that MSGPASS_{Sema} and MSGPASS_{Ctl} are equivalent. We base the analysis upon the notion of correct (and complete) ASM refinement defined in [2] and show that MSGPASS_{Sema} is a correct refinement of MSGPASS_{Ctl} , and vice versa (completeness). As a by-product, this implies, by Theorem [1], an alternative proof for the correctness of MSGPASS_{Sema} . The two machines have different operations; also the ways the operations are structured slightly differ from each other. Therefore, we have to investigate in what sense one can speak of pairs of corresponding and possibly equivalent runs in the two machines.

In MSGPASS_{Ctl} or MSGPASS_{Sema} runs, each successful message exchange is characterized by a *message exchange triple* of moves

$$\begin{aligned} & (\text{STARTSEND\&WAIT}(m, src, dest) \mid \text{STARTSYNC\&WAIT}(dest)) ; \\ & \text{PASSMSG}(dest) \end{aligned}$$

where by $m \mid m'$, we indicate that the two moves m, m' can occur in any order in the run in question (remember: both a sender and a receiver can initiate an attempt to synchronize with a partner for message exchange), and by $m; m'$ the sequential order of m preceding m' in the run (note: a synchronized PASSMSG move can come only after the two corresponding moves STARTSYNC\&WAIT and STARTSEND\&WAIT). The message exchange triple components are furthermore characterized by the following requirements:

- The $\text{STARTSYNC\&WAIT}_{Ctl}(dest)$ move is the last $\text{STARTSYNC\&WAIT}_{Ctl}$ move of the receiver agent $dest$ that precedes the $\text{PASSMSG}_{Ctl}(dest)$ move executed when src and $dest$ are synchronized.
- By the $\text{SIGNALSEMA}(insema(dest))$ move of $\text{STARTSYNC\&WAIT}_{Sema}(dest)$ src is made ready to be scheduled to $\text{PASSMSG}_{Sema}(src, dest)$.

We call message exchange triple moves in the two runs corresponding to each other if they have the same name and concern the same parameters among $(m, src, dest)$. Similarly we speak about correspondence of message exchange triples. Pairs of corresponding MSGPASS_{Ctl} and MSGPASS_{Sema} runs are those runs which are started in equivalent corresponding initial states and perform corresponding message exchange triple moves in the same order. Since single moves correspond to each other, the segments of computation of interest to be compared in corresponding runs consist only of the given single moves. The locations of interest to be compared are $inbox(dest)$ and $outbox(dest)$, the ones updated by RECORDFORTRANSFER resp. DELIVERMSG moves in the two states of interest, the same in both machines. The equivalence is here simply the identity of the values of these locations in the corresponding states of interest.

By the ASM refinement framework defined in [2], these definitions turn the following sentence into a mathematically precise statement.

Theorem 4. MSGPASS_{Sema} is a correct refinement of MSGPASS_{Ctl} .

Proof. One has to show that given corresponding MSGPASS_{Sema} , MSGPASS_{Ctl} runs, for each message exchange triple move in the MSGPASS_{Sema} run, one can find a corresponding message exchange triple move in the MSGPASS_{Ctl} run such that the locations of interest in the corresponding states of interest are equivalent. This follows by an induction on runs and the number of message exchange triple moves. The basis of the induction is guaranteed by the stipulation that the two runs are started in equivalent corresponding states. The induction step follows from the one-to-one relation between (occurrences of) corresponding message exchange triple moves described above, using the definitions of the respective machine moves and the fact that the two crucial operations RECORDFORTRANSFER and DELIVERMSG , which update the locations of interest, are by definition the same in both machines. \square

Symmetrically, one can prove the following theorem. The two theorems together constitute the equivalence theorem (also called bisimulation).

Theorem 5. MSGPASS_{Ctl} is a correct refinement of MSGPASS_{Sema} .

7 Concluding Remarks and Future Work

Possible extensions of interest concern stronger machines, as in the example below. Similar examples include the consideration of timeouts, etc. A central question is whether, and how, the equivalence-proof scheme can be extended

to work in the multiprocessor case. We also suggest investigating whether the models and proofs in this paper can be implemented by current theorem provers.

Assume we want to use MSGPASS_{Ctl} for the case that RECEIVE_{Ctl} is restricted to receive from a set of expectedsndrs , which is assumed to be defined when RECEIVE_{Ctl} is called. The first issue to decide is whether the receiver waits until a message from an expected sender shows up (blocking case) or whether in absence of such a message the receiver may receive other messages.

In the non-blocking case the issue is simply a question of priority. Therefore it suffices to refine the function hd used in PASSMSG to choose the first element from $\text{wtsndr}(p) \cap \text{expectedsndr}(p)$ if this set is not empty, and the first element from $\text{wtsndr}(p)$ otherwise. This extension includes the case that the set expectedsndr itself may contain elements of different priorities.

In the blocking case, it suffices to strengthen the RECEIVE_{Ctl} rule by replacing $\text{wtsndr}(p)$ with $\text{wtsndr}(p) \cap \text{expectedsndr}(p)$ in the guards of the two subrules. The notion of src being synchronized with $dest$ in Theorem [11](#) has to be refined by restricting scr to elements in $\text{expectedsndr}(dest)$. This refinement represents a conservative (i.e. purely incremental) extension of the abstract model.

Acknowledgement. We thank Gerhard Schellhorn for a discussion of an earlier version of the proof, which led to its simplification.

References

1. Altenhofen, M., Börger, E.: Concurrent abstract state machines and ^+CAL programs. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 1–17. Springer, Heidelberg (2009)
2. Börger, E.: The ASM refinement method. *Formal Aspects of Computing* 15, 237–257 (2003)
3. Börger, E., Craig, I.: Modeling an operating system kernel. In: Diekert, V., Weicker, K., Weicker, N. (eds.) Informatik als Dialog zwischen Theorie und Anwendung, pp. 199–216. Vieweg+Teubner, Wiesbaden (2009)
4. Börger, E., Stärk, R.F.: *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003)
5. Craig, I.: *Formal Models of Operating System Kernels*. Springer, Heidelberg (2007)
6. Craig, I.: *Formal Refinement for Operating System Kernels*. Springer, Heidelberg (2007)
7. Tanenbaum, A.S.: *Modern Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs (1987)

AsmL-Based Concurrency Semantic Variations for Timed Use Case Maps

Jameleddine Hassine

Cisco Systems, 3000 Innovation Drive, Kanata, Ontario, K2K 3J9, Canada
jhassine@cisco.com

Abstract. Scenario-driven requirement specifications are widely used to capture and represent high-level requirements. Timed Use Case Maps (TUCM) is a high-level scenario based modeling technique that can be used to capture and integrate behavioral and time-related aspects at a high level of abstraction. The Timed Use Case Maps language assumes durational semantics which introduces semantic variation points when dealing with concurrent flows. In this paper, we introduce three AsmL-based operational semantics, making the semantic variation points of TUCM concurrent behavior explicit. The proposed semantics are illustrated using an example.

1 Introduction

The Use Case Maps language (UCM), part of the ITU-T standard User Requirements Notation (URN) Z.151 [1], is a high-level visual scenario-based modeling technique that can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction. However, non-functional aspects are often overlooked during the initial system design and treated as non-related behavioral issues and described therefore in separate models.

Recognizing the need to incorporate non-functional aspects, and in particular time-related aspects, into requirement languages in order to correctly model and analyze time dependent applications at early stages of system development process, Use Case Maps language has been extended with the notion of time to create the Timed Use Case Maps language (TUCM) [2]. Timed Use Case Maps offers the opportunity to perform quantitative analysis [3,4] during the requirement stage, helping detect errors early and reducing the cost of later redesign activities.

The Timed Use Case Maps language [2] assumes durational semantics which introduces semantic variation points when dealing with concurrent flows. These semantic variation points provide intentional degrees of freedom for the interpretation of timed UCM concurrent models, allowing for variabilities that one can customize for a given application domain. The ability to identify and implement these variabilities without modifying the original specification, represents the major motivation for this paper. This paper serves different purposes:

- It addresses the problem of making the semantic variation points of TUCM concurrent behavior explicit. This would allow for the specialization of concurrency of TUCM for a particular situation or domain.

- It provides three AsmL-based [5] variations for timed UCM concurrent models:
 - Interleaving semantics.
 - True Concurrency semantics.
 - Run to completion semantics.
- It extends the ongoing research towards the construction of a formal framework for the Use Case Maps language [1] to describe, simulate and analyze real-time systems [2,6,3,4].

The remainder of this paper is organized as follows. The next section describes and discusses some related work. In an attempt to make this paper self-contained, we provide in Section 3 a brief overview of the UCM time extensions introduced in Hassine et al. [2]. Section 4 represents the core of our paper, where we present three AsmL-based simulation engines for timed UCM models. An illustrative example, showing the applicability of the proposed TUCM operational semantics along with their corresponding execution traces, is presented in Section 5. Finally, conclusions are drawn in Section 6.

2 Related Work

In the early days of the Use Case Maps language, UCM models were developed and maintained using *UCMNav* (*UCM Navigator*) tool [7]. In addition to its reusability, extensibility and scalability issues, UCMNav suffers from a rigid scenario traversal mechanism without semantic variation points and without tolerance for errors. Some of these limitations have been addressed by the early versions of *jUCMNav* [8], its Eclipse-based successor. Later, Kealey and Amyot [9] have proposed and implemented an enhanced scenario traversal mechanism into *jUCMNav*. Furthermore, the authors [9] have identified a set of semantic variation points for *jUCMNav*. However, neither time nor concurrency concepts were addressed. In this work, we focus mainly on semantic variation points related to time and concurrency in the Timed Use Case Maps language.

The Use Case Maps language [1] has been extended with the time constraints [2]. Three formalization approaches [1] for the Timed Use Case Maps language have been proposed:

1. Hassine et al. [2] have proposed a Clocked Transition System (CTS) based semantics for timed use case maps models. The authors have defined two sets of transition rules (i.e. two step semantics) aiming to cover both interleaving and true concurrency models. However, no executable model has been provided.
2. Based on a dense time model, Hassine et al. [3] have defined a timed automaton [11] template for each timed UCM construct. A timed UCM specification can be modeled as a network of concurrent timed automata with interleaving semantics. The resulting Timed Automata (TA) models can be validated and verified using the UPPAAL model checker [12].
3. Recently and in a closely related work, Hassine [4] has extended the Timed Use Case Maps language with resource constraints, allowing for schedulability analysis. The proposed scheduler implements a priority driven non-preemptive scheduling with interleaving semantics.

¹ For a detailed description of timed UCM formalization, the reader is invited to consult [2,3,10,4].

On the Abstract State Machines front, Börger et al. [13] have used *multi-agent ASMs* [14] [15] to model the concurrent sub-states and the internal activities of UML state machines. The authors [13] have claimed that their approach solves many semantic variation points, such as event deferring, completion mechanism and concurrent internal activities. More recently, Ouimet and Lundqvist [16] have extended ASMs by including facilities for compact and legible specification of non-functional behavior, namely time and resource consumption. The concurrency semantics of the proposed Timed ASM (TASM) [16] is synchronous with respect to durative steps.

In what follows, we provide a brief overview of the Timed Use Case Maps language, initially introduced in [2].

3 The Timed Use Case Maps Language

Timed UCMs visually model and integrate a number of operational scenarios (in a map-like diagram) cutting through a system's component architecture. In this research, in order to focus on the concurrency aspect in timed Use Case Maps models, a simplified set of the time criteria introduced in [2] is considered.

3.1 Selected Time Criteria

The following summarizes the selected time-related criteria:

- A global and centralized clock (i.e. MasterClock (*MClock*)) for measuring and increasing time globally over the system is used.
- A discrete time model is adopted. The smallest time unit (i.e. clock tick) used to track system evolution over time is named δ . It defines the granularity of the master clock. For the sake of simplicity, $\delta=1$ is used.
- *Durational semantic model*: Time is mainly consumed by responsibilities. Each responsibility is associated with a duration (i.e. *Dur*). Timed UCM control constructs such as OR-Forks (branching construct involving condition evaluation), AND-Forks (used to split a single flow into two or more concurrent flows), OR-Joins (capture the merging of two or more independent scenario paths) and AND-Joins (capture the synchronization of two or more concurrent scenario paths) may take some time to complete. In [2], a best and a worst case execution times of a responsibility are considered.
- *Delay*: A responsibility may be enabled after a specified delay (i.e. *Delay*). A such delay is introduced in order to describe, for instance, situations of queueing delay or when the resources needed to execute a responsibility are not immediately available. In [2], both a lower (i.e. *minDL*) and upper bound (i.e. *maxDL*) may be imposed on the enabling of a responsibility.
- Both *relative* and *absolute* time models are considered. Relative time is used to define the duration of responsibilities and their incurred delay. Absolute time is used to track the value of the master clock (i.e. *MClock*). It can be used in start points to record the scenario starting time and to define responsibilities' deadlines.

- *Urgency*: A responsibility is considered as *urgent* when enabled immediately after the execution of its predecessor (i.e. $Delay = 0$). Alternatively, it is considered as *delayable*. All UCM control constructs (i.e. OR-Fork, OR-Join, AND-Fork, etc.) are considered as urgent once enabled. Transitions are processed as soon as they are enabled allowing for a maximal progress.

3.2 Syntax of Timed Use Case Maps

The Use Case Maps language provides the stub concept allowing for hierarchical decomposition of complex maps. UCM path details can be hidden in separate sub-diagrams called plug-ins, contained in stubs (diamonds) on a path. Since stubs are simple containers of plugins, no special treatment is considered with respect to time and concurrency. A timed UCM specification is defined as follows:

Definition 1 (Timed Use Case Maps). A timed UCM is denoted by a 8-tuple $(D, H, \lambda, C, GVar, B_c, S, B_s)$ where:

- D is the UCM domain defining all timed UCM constructs.
- H is the set of edges connecting timed UCM constructs to each other.
- λ represents the transition relation defined as: $\lambda = D \times H \times D$.
- C is the set of defined components.
- $GVar$ is the set of specification's global variables.
- B_c is a component binding relation defined as $B_c = D \times C$. B_c specifies which element of D is associated with which component of C .
- S is a plug-in binding relation defined as $S = Stub \times Plugin \times Cond$, where $Stub$ is the set of TUCM stubs, $Plugin$ is the set of plugin maps and $Cond$ is the set of conditions governing the plugin selection.
- B_s is a stub binding relation and is defined as $B_s = Stub \times \{IN/OUT\} \times \{SP/EP\}$. B_s specifies how the start and end points of a plugin map would be connected to the path segments going into or out of the stub.

3.3 A Basic Timed UCM Example

Figure 1 illustrates a basic timed UCM with delayable and durative responsibilities (e.g. $R1$ has a delay of 1 and a duration of 2). The start point $S1$ is triggered at $MClock = 0$. UCM control constructs OR-Fork OF and AND-fork AF are urgent. OF takes one

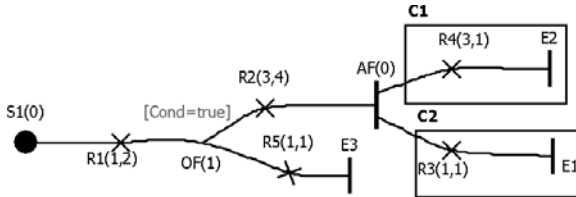


Fig. 1. UCM basic Example

time unit to complete while AF is instantaneous. End points $E1$ and $E2$ are urgent and instantaneous since they are not associated with any type of processing. Responsibilities $R4$ and $R3$ can be executed concurrently since they are bound to different components (i.e. $C1$ and $C2$).

4 AsmL Implementation of the Timed UCM Concurrency Models

4.1 Timed UCM Simulation Engine

All variations of the proposed timed UCM simulation engine (described in sections 4.2, 4.3 and 4.4) use the class $AGENT$ to specify the abstract set of agents ag . An agent moves through its associated timed UCM map, by executing the timed UCM construct(s) at its current active edge(s), i.e. edge where the agent's control lies. Every agent can mainly be characterized by:

- A static function, $id: AGENT \rightarrow String$, used to identify system agents.
- A dynamic function, $mode: AGENT \rightarrow \{running, inactive\}$, used to track an agent's mode. An agent may be running in normal mode or inactive once the agent has finished its computation.

Typically, a running agent has to look at the delay associated with the target timed UCM construct(s) of its active edge(s) to determine which construct should be executed next. Once a running agent finishes its execution thread (no more active edges), or encounters a blocking situation (for instance when no conditions evaluate to true at an OR-Fork), it changes its mode to become inactive ($me.mode=inactive$).

AsmL Common Data Structures. The data structures, initially introduced in [6], are extended to cover time aspects. Figure 2 describes an excerpt of the AsmL implementation of the data structures common to the three proposed operational semantics. The

```

structure UCMConstruct
//StatePoint
  case SP_Construct
    in_hy as HyperEdge
    out_hy as HyperEdge
    label as String
    preCondition as BooleanExp
    Delay as Integer
    location as Component
//Responsibility
  case R_Construct
    in_hy as HyperEdge
    out_hy as HyperEdge
    label as String
    Delay as Integer
    Duration as Integer
    location as Component
//OR-Fork
  case OF_Construct
    in_hy as HyperEdge
    selec as Set of
OR_Selection
  label as String
  Duration as Integer
  location as Component
//AND-Fork
  case AF_Construct
    in_hy as HyperEdge
    out_hy as Set of HyperEdge
    label as String
    Duration as Integer
    location as Component
//Stub
  case Stub_Construct
    entry_hy as Set of
HyperEdge
    exit_hy as Set of HyperEdge
    Selec_plugin as Set of
Stub_Selection
    Binding_Relation as Set of
Stub_Binding
    label as String
// List of hyperedges
enum HyperEdge
  e1
  e2
  h0 // null
// List of components
enum Component
  C1
  unbound // undefined
// UCM transition relation
structure UCMElement
  source as UCMConstruct
  hyper as HyperEdge
  target as UCMConstruct
// Selection conditions of
OR-Forks
structure OR_Selection
  out_hy as HyperEdge
  out_cond as BooleanExp
// Stub binding relation
structure Stub_Binding
  plugin as Maps
  stub_hy as HyperEdge
  start_End as UCMConstruct
// Plugin selection
structure Stub_Selection
  stub_plugin as Maps
  stub_cond as BooleanExp
// UCM Map
structure Maps
  label as String
  ele as Set of UCMElement
  ep as Set of EP_Construct

```

Fig. 2. Excerpt of the AsmL implementation of the timed UCM data structures

structure *UCMConstruct* incorporates many case statements as a way of organizing different variants of UCM constructs. The structure *UCMElement* implements the transition relation λ (see Section 3.2). The structure *OR_Selection* specifies the boolean conditions associated with the outgoing branches of OR-Forks. The data structures *Stub_Selection* and *Stub_Binding* implement respectively the stub selection policies and the stub binding relation. The enumerated sets *Hyperedges* and *Component* denote respectively the set of edges and the set of used components. The structure *MAPs* represented the timed UCM rootmap (i.e. main timed UCM map). It is composed of an identifier (i.e. *label*), a transition relation (i.e. *ele*) and a set of end points (i.e. *ep*).

AsmL Supportive Functions. At each execution step, an agent needs to know about the set of active edges, the existence of target constructs and their associated delays. Figure 3 illustrates some global functions used to support the execution of timed UCM specifications. For example, the function *GetDelayTargetConstruct* is used to capture the delay associated with the next timed UCM construct to be executed.

```
// GetInHyperEdge returns the set of incoming edges of a timed UCM construct
GetInHyperEdge (i as UCMConstruct) as Set of HyperEdge
match i
  R_Construct (a,b,c,d,e,f): return {a}
  AF_Construct (a,b,c,d,e): return {a}
  AJ_Construct (a,b,c,d,e): return a
  ...
// Get the delay of the target timed UCM construct
GetDelayTargetConstruct (h as HyperEdge, M as Maps) as Integer
choose k in M.ele where k.hyper = h
return ComputeMinimumDelay (k.target)
ifnone
writeLine("\n GetDelayTargetConstruct FAILED" )
return 0
// Compute minimum delay
ComputeMinimumDelay (i as UCMConstruct) as Integer
match i
  SP_Construct (a,b,c,d,e,f): return e
  R_Construct (a,b,c,d,e,f): return d
  AF_Construct (a,b,c,d,e): return 0
  ...
```

Fig. 3. Excerpt of the AsmL implementation of the supportive functions

Since the focus is on time and concurrency variations, we only illustrate the AsmL rules of responsibilities (durative and involving delay) and AND-Forks (used to split a single flow into many concurrent flows). For more details on the AsmL rules of other timed UCM constructs, the reader is invited to consult [10] and [4].

4.2 Interleaving Semantics

In presence of UCM components, concurrent paths bounded to the same component are sharing also the same component resources (for instance same CPU and memory). Therefore, these concurrent paths must behave in interleaving semantics.

In interleaving TUCM models, responsibilities represent atomic actions, not to be decomposable, and their execution is not interruptible. Figure 4 illustrates an excerpt of the AsmL implementation of the interleaving semantics.

```

structure activ
edge as HyperEdge // Active edge
level as Maps // Corresponding MAP (rootMap or plugin)
delay as Integer // Delay of the target construct
// Set of activ initially empty
var act as Set of activ = {}

class Agent
const id as String
var mode as Mode
Program()
writeLine("Start Executing : " + me.id)
step
until1 ((act = {}) or (me.mode = inactive))
do
let del = [t1.delay | t1 in act]
let minimumDL = (min x | x in del)
choose z in act where z.delay = minimumDL
choose s2 in z.level.ele where HyperExists(z.edge, GetInHyperEdge(s2.source))
match (s2.source)
// Rule of Responsibility
R_Construct (a,b,c,d,e,f): step
MClock := MClock + d
step
ExecuteResponsibility((s2.source) as R_Construct)
step
MClock := MClock + e
add activ(b, z.level, GetDelayTargetConstruct(b, z.level)) to act
step
choose r in act where r.edge = a
remove r from act

// Rule of AND-Fork
AF_Construct (a,b,c,d,e): step
writeLine("\n Executing AND-Fork: " + c)
forall i in b
add activ(i, z.level, GetDelayTargetConstruct(i, z.level)) to act
step
choose r in act where r.edge = a
remove r from act
MClock := MClock + d

Main()
var todo = StartPoints
step
add activ(in1, RootMap, 0) to act
step while todo.Count > 0
choose a in todo
todo(a) := false
let ag = new Agent("Root Map:" + a.label, running)
ag.Program()

```

Fig. 4. Excerpt of the AsmL implementation of the interleaving semantics

In addition to the global data structures presented in Section 4.1, a new structure named *activ* is created to track the different delays associated with the ready-to-execute timed UCM constructs. At each step, the timed UCM construct, with the minimum delay, is selected for execution. Hence, the execution of parallel responsibilities is reduced to a deterministic interleaving in case of a single minimum delay or to a nondeterministic interleaving in case of many responsibilities sharing the same minimum delay.

The interleaving solution is based on a single agent model, where only one single construct can be executing at a time. Once the responsibility with the minimum delay is selected for execution, the master clock is increased by its associated delay (i.e. $MClock := MClock + d$). Then, the master clock is increased by the value of the actual execution time (i.e. $MClock := MClock + e$). In [16], timed ASM steps have the same duration allowing for a linear time progression, while in our model the time is controlled by a global clock which is incremented in a non-linear way. Upon execution completion, the current *activ* element is removed from the set *act* and the next *activ* element (corresponding to the outgoing edge) is added. The execution of an AND-Fork is reduced to the addition of a new structure *act* for each outgoing edge.

4.3 True Concurrency Semantics

Truly concurrent models allow for the concurrent execution of not causally related actions. AsmL language doesn't have yet runtime support for true concurrency or simulation of true concurrency [17]. To address this limitation, we have designed a true concurrency solution based on a single agent handling multiple executions of timed UCM constructs at any given time. Instead of having the system executes a full timed UCM construct (as in interleaving semantics), the system makes progress on every single clock tick.

Figure 5 illustrates an excerpt of the AsmL implementation of the true concurrency solution. The *activ* data structure, presented in Figure 4, is slightly modified to include responsibility's remaining execution time (i.e. *executionTime*). At each step, the master clock *MClock* is incremented by 1, all active delays greater than 0 are decremented by 1, and constructs reaching a delay equal to zero are selected for execution. Although, this

```

structure activ
  edge as HyperEdge // Active edge
  level as Maps // Corresponding MAP (rootMap or plugin)
  delay as Integer // Delay of the target Construct
  executionTime as Integer // Remaining execution time of the target Construct
  label as String // Label of the target Construct
  // Set of activ initially empty
  var act as Set of activ = {}

class Agent
  const id as String
  var mode as Mode
  Program()
  step
  until ((act = {}) or (me.mode = inactive))
  do
    let h = {t1.edge | t1 in act }
    forall actEdge in act
      MClock := MClock + 1
      if (actEdge.delay > 0)
        let dlay = actEdge.delay as Integer - 1
        add activ(actEdge.edge, actEdge.level, dlay, actEdge.executionTime, actEdge.label) to act
        remove actEdge from act
      else
        choose s2 in actEdge.level.ele where HyperExists (actEdge.edge, GetInHyperEdge (s2.source))
        match (s2.source)
  // Rule of Responsibility
    R_Construct (a,b,c,d,e,f): step
      if (actEdge.executionTime > 0)
        ExecuteResponsibility ((s2.source) as R_Construct)
        let remainingTime = actEdge.executionTime as Integer - 1
        add activ(actEdge.edge, actEdge.level, actEdge.delay,
remainingTime, actEdge.label) to act
        remove actEdge from act
      else
        remove actEdge from act
        add activ(b, actEdge.level, GetDelayTargetConstruct (b,
actEdge.level), GetExecutionTargetConstruct (b, actEdge.level), GetLabelTargetConstruct (b,
actEdge.level)) to act
  // Rule of AND-Fork
    AF_Construct (a,b,c,d,e): step
      forall i in b
        add activ(i, actEdge.level, GetDelayTargetConstruct (i,
actEdge.level), GetExecutionTargetConstruct (i, actEdge.level),
GetLabelTargetConstruct (i,actEdge.level)) to act
      step
        choose r in act where r.edge = a
        remove r from act

  // Rule of Start Point
  // ...

```

Fig. 5. Excerpt of the AsmL implementation of the true concurrency solution

approach fulfills the need of having a true concurrent simulation engine for timed UCM models, it does not scale well. Indeed, simulation times would increase exponentially with responsibilities having large delays and long execution periods.

4.4 Multi-Agent Solution: Run to Completion

Both the interleaving and true concurrency operational semantics, implement a single agent that controls the whole system execution. Another alternative would be to use multi-agents ASM [14]. In addition to *id* and *mode*, an agent can be characterized by the following attributes:

- active: $AGENT \rightarrow HyperEdge$, represents the identifier of the active edge leading to the next timed UCM construct to be executed.
- level: $AGENT \rightarrow MAPS$, provides the UCM map that the agent is currently traversing.
- current_Stub: $AGENT \rightarrow Stub_Construct$, provides the stub that the agent is currently traversing.

Figure 6 presents an excerpt of the AsmL implementation of the multi-agent solution. When the control is on an edge entering an AND-Fork, the currently running agent

```

class Agent
const id as String
var active as HyperEdge
var mode as Mode
var level as Maps
var current_Stub as Stub_Construct
Program()
step
until me.mode = inactive
do
  choose s2 in level.e1e where HyperExists (active, GetInHyperEdge (s2.source))
  match (s2.source)
// Rule of Responsibility
  R_Construct (a,b,c,d,e,f): step
    MClock := MClock + d
  step
    ExecuteResponsibility ((s2.source) as R_Construct)
  step
    MClock := MClock + e
    me.active := b
// Rule for AND-Fork Construct
  AF_Construct (a,b,c,d,e): step
    forall i in b
    add activ(i, GetDelayTargetConstruct (i, me.level)) to act
  step
    until (act = {})
  do
    let del = [t1.delay | t1 in act]
    let minimumDL = (min x | x in del)
    choose z in act where z .delay= minimumDL
    let ag = new Agent("Agent-"+z.edge, z.edge, running,
me.level, me.current_Stub)
    ag.Program()
    remove z from act
    me.mode := inactive
// Rule for Start Point Construct
//...

```

Fig. 6. Excerpt of the AsmL implementation of the multi-agent solution

creates the necessary new subagents and sets their mode to running, then sets its mode to inactive. Each new AsmL subagent inherits the program for executing timed UCMs, and its control starts at the associated outgoing edge of the AND-Fork. The order of activation of subagents depends on the delay associated with the subsequent timed UCM construct (i.e. the subagent with the minimal delay is activated first). Each agent runs to completion (i.e. till it reached an end point or an AND-Join) before the next agent starts executing.

Implementing an interleaving or a true concurrency semantics using multi-agent ASMs, would require the design of a scheduler that coordinates the execution of participating agents. This alternative is not considered in this research.

5 Illustrative Example

Although simple, the following example (Figure 7) illustrates the applicability of the proposed three timed UCM simulation engines. The AsmL Implementation is executed within Spec Explorer [18], an advanced model-based specification and conformance testing tool.

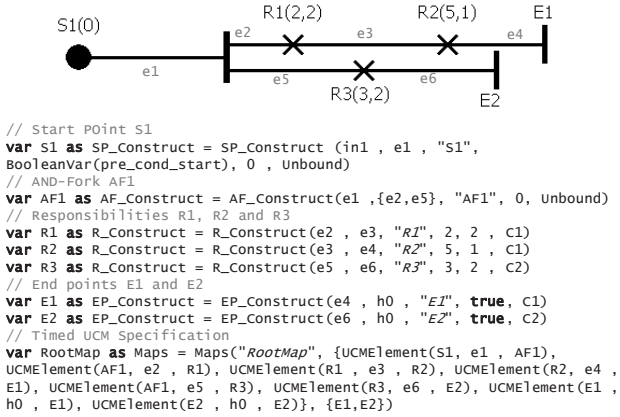


Fig. 7. Illustrative timed UCM example and its corresponding AsmL implementation

Figure 8 shows the timed trace generated using the interleaving simulation engine. Each step corresponds to the execution of a construct. Responsibility $R1$ is executed first, followed by $R3$ then $R2$ and the system completes its execution at $MClock=15$.

Figure 9 shows the timed trace generated using the true concurrency simulation engine. Each step corresponds to one clock tick. At $MClock=5$, responsibilities $R1$ and $R3$ are concurrently executed, leading to the system completing its execution at $MClock=14$ (one time unit earlier than the interleaving trace). It is worth noting that the number of execution steps in the true concurrency trace is the double of those of the interleaving trace (14 steps using true concurrency versus 7 steps using the interleaving).


```

Set of active edges={in1}
Set of delays=[0]; Minimum Delay=0
Start Point:S1 triggered in Component:Unbound at Mclock = 0
-----
Set of active edges={e1}
Set of delays=[0]; Minimum Delay=0
Executing AND-Fork: AF1 in component: Unbound at MCLock=0
-----
Set of active edges={e5, e2}
Set of delays=[3, 2]; Minimum Delay=2
Start Executing Responsibility: R1 in component: C1 at MCLock=2
-----
Set of active edges={e5, e3}
Set of delays=[3, 5]; Minimum Delay=3
Start Executing Responsibility: R3 in component: C2 at MCLock=7
-----
Set of active edges={e6, e3}
Set of delays=[5, 0]; Minimum Delay=0
End Point: E2 part of root map reached in Component:C2 at MCLock=9
-----
Set of active edges={e3}
Set of delays=[5]; Minimum Delay=5
Start Executing Responsibility: R2 in component: C1 at MCLock=14
-----
Set of active edges={e4}
Set of delays=[0]; Minimum Delay=0
End Point: E1 part of root map reached in Component:C1 at MCLock=15

```

Fig. 8. Timed trace corresponding to the interleaving semantics solution

<pre> MCLock=0 Active: (edge:in1,delay:0,remaining exec:0) Start Point:S1 in Component:Unbound at MCLock=0 ----- MCLock=1 Active: (edge:e1,delay:0,remaining exec:0) Executing AND-Fork: AF1 in component: Unbound at MCLock=1 ----- MCLock=2 Active: (edge:e5,delay:3,remaining exec:2) Active: (edge:e2,delay:2,remaining exec:2) ----- MCLock=3 Active: (edge:e2,delay:1,remaining exec:2) Active: (edge:e5,delay:2,remaining exec:2) ----- MCLock=4 Active: (edge:e5,delay:1,remaining exec:2) Active: (edge:e2,delay:0,remaining exec:2) Executing Responsibility: R1 in component: C1 at MCLock=4 ----- MCLock=5 Active: (edge:e2,delay:0,remaining exec:1) Executing Responsibility: R1 in component: C1 at MCLock=5 Active: (edge:e5,delay:0,remaining exec:2) Executing Responsibility: R3 in component: C2 at MCLock=5 ----- MCLock=6 Active: (edge:e5,delay:0,remaining exec:1) Executing Responsibility: R3 in component: C2 at MCLock=6 Active: (edge:e2,delay:0,remaining exec:0) </pre>	<pre> ----- MCLock=7 Active: (edge:e5,delay:0,remaining exec:0) Active: (edge:e3,delay:5,remaining exec:1) ----- MCLock=8 Active: (edge:e3,delay:4,remaining exec:1) Active: (edge:e6,delay:0,remaining exec:0) End Point: E2 part of root map reached in Component:C2 ----- MCLock=9 Active: (edge:e3,delay:3,remaining exec:1) ----- MCLock=10 Active: (edge:e3,delay:2,remaining exec:1) ----- MCLock=11 Active: (edge:e3,delay:1,remaining exec:1) ----- MCLock=12 Active: (edge:e3,delay:0,remaining exec:1) Executing Responsibility: R2 in component: C1 at MCLock=12 ----- MCLock=13 Active: (edge:e3,delay:0,remaining exec:0) ----- MCLock=14 Active: (edge:e4,delay:0,remaining exec:0) End Point: E1 part of root map reached in Component:C1 </pre>
--	--

Fig. 9. Timed trace corresponding to the true concurrency solution

```

Start Executing: MainS1
-----
MClOCK=0
MainS1.active:in1
MainS1:Start Point:S1 at MclOCK=0
-----
MClOCK=0
MainS1.active:e1
MainS1:Rule of AND-Fork:AF1 at MclOCK=0
Adding: (e2,2)
Adding: (e5,3)
Start Executing: Agent-e2
-----
MClOCK=0
Agent-e2.active:e2
Start Executing Responsibility: R1 in component: C1 at MClOCK=2
-----
MClOCK=4
Agent-e2.active:e3
Start Executing Responsibility: R2 in component: C1 at MClOCK=9
-----
MClOCK=10
Agent-e2.active:e4
Agent-e2: End point:E1 at MclOCK=10
Agent-e2:Execution Terminated successfully
Start Executing: Agent-e5
-----
MClOCK=10
Agent-e5.active:e5
Start Executing Responsibility: R3 in component: C2 at MClOCK=13
-----
MClOCK=15
Agent-e5.active:e6
Agent-e5: End point:E2 at MclOCK=15
Agent-e5:Execution Terminated successfully

```

Fig. 10. Timed trace corresponding to the multi-agent solution

Figure 10 shows the timed trace generated using the multi-agent simulation engine. The execution of the AND-Fork generates two subagents *Agent-e2* and *Agent-e5*. *Agent-e2* is run to completion and it is executed before *Agent-e5* since responsibility *R1* delay is less than *R3* delay. Responsibility *R1* is executed first, followed by *R2* then *R3* and the system completes its execution at *MClOCK=15*.

6 Conclusions

In this paper, we have proposed three AsmL-based operational semantics, making the semantic variation points of TUCM concurrent behavior explicit. Depending on the application domain, the user may choose between interleaving, true concurrency and run to completion semantics. We have shown that such variabilities can be identified and implemented without modifying the original specification. The generated timed traces capture various aspects of a system run (e.g. executed constructs, time of execution, components, etc.) allowing for analysis and validation. Furthermore, such timed traces would provide the designer, at an early stage, with better understanding of timing properties of the system and how concurrency may impact overall execution.

The proposed AsmL rules can be easily modified to accommodate language evolution. Indeed, the modification of the semantics of a timed UCM construct or the addition of a new construct result in the modification or the addition of a new AsmL rule that describes the semantics of the new construct.

As part of the future work, it would be interesting to establish meaningful case studies that allow us to evaluate the proposed concurrency models on real world examples. Furthermore, we would like to further extend the presented operational semantics to include additional aspects such as priority and preemption.

References

1. ITU-T: Draft recommendation Z.151, User Requirements Notation (URN) (2008)
2. Hassine, J., Rilling, J., Dssouli, R.: Timed Use Case Maps. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 99–114. Springer, Heidelberg (2006)
3. Hassine, J., Rilling, J., Dssouli, R.: Formal Verification of Use Case Maps with Real Time Extensions. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 225–241. Springer, Heidelberg (2007)
4. Hassine, J.: Early Schedulability Analysis with Timed Use Case Maps. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009, September 22-24. LNCS, vol. 5719, pp. 98–114. Springer, Heidelberg (2009)
5. AsmL: Microsoft Research: The Abstract State Machine Language (2006), <http://research.microsoft.com/foundations/AsmL/>
6. Hassine, J., Rilling, J., Dssouli, R.: Abstract Operational Semantics for Use Case Maps. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 366–380. Springer, Heidelberg (2005)
7. Miga, A.: Application of Use Case Maps to system design with tool support. Master's thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1998)
8. jUCMNav: jUCMNav Project (tool, documentation, and meta-model) (2006), <http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome> (Last accessed, October 2007)
9. Kealey, J., Amyot, D.: Enhanced Use Case Map Traversal Semantics. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 133–149. Springer, Heidelberg (2007)
10. Hassine, J.: Formal Semantics and Verification of Use Case Maps. PhD thesis, Concordia University, Montreal, Quebec, Canada (2008)
11. Alur, R., Dill, D.L.: A theory of Timed Automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)
12. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer* 1(1-2), 134–152 (1997)
13. Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASMs. *Information and Software Technology* 46(5), 287–292 (2004)
14. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (ed.) *Specification and Validation Methods*, pp. 9–36. Oxford University Press, Oxford (1995)
15. Glässer, U., Gurevich, Y., Veanes, M.: Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering* 30(7), 458–472 (2004)
16. Ouimet, M., Lundqvist, K.: The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. *Journal of Universal Computer Science* 14(12), 2007–2033 (2008)
17. Veanes, M.: Modeling software: From theory to practice. In: Agrawal, M., Seth, A.K. (eds.) *FSTTCS 2002*. LNCS, vol. 2556, pp. 37–46. Springer, Heidelberg (2002)
18. SpecExplorer: Microsoft Research: Spec Explorer tool (2006), <http://research.microsoft.com/specexplorer/>

Bârun: A Scripting Language for CoreASM

Michael Altenhofen and Roozbeh Farahbod

SAP Research, Karlsruhe, Germany

michael.altenhofen@sap.com, sap@roozbeh.ca

Abstract. Scenarios have been used in various stages of the software development process, in particular in requirement elicitation and software validation and testing. In this paper, we present our recent work on the specification, design and implementation of a CoreASM plugin, called Bârun, that offers a powerful scripting language on top of the CoreASM extensible modeling framework and tool environment for high-level design and analysis of distributed systems. We illustrate the application of Bârun and demonstrate its features using an industrial case study.

1 Introduction

The CoreASM framework [1,2] provides a lean language and a tool environment for writing *executable* high-level system specifications as Abstract State Machines (ASMs) [3]. These specifications can serve as the basis for experimental validation through simulation and testing in the early phases of the overall system design. As a special subset, *reactive* systems expose their desired behavior through interactions with an external environment. In other words, non-deterministic choices made by the external environment steer the observable state transitions made by the machine. As indicated by the word *external*, the steps performed by the environment should not be part of the specification itself, but should rather be captured by separate artifacts, often called *scenarios* [4].

Depending on their ultimate purpose, scenarios may have to take different views of the system. If we're solely interested in validating the system—checking whether the specification meets the user requirements, it will be sufficient to treat the system as a *black box* neglecting any internal state transitions and focusing on the externally visible state changes. For this type of scenarios, we actually want notational support for (repeatedly) checking system properties (in the form of *assertions*) during a scenario run. On the other hand, scenarios may also be used to perform more rigid (unit) tests and thus may require a more thorough *gray box* view and more subtle control of the internal state of the system. Here, our experiences have shown that scenario designers would benefit from having a *local state* in the scenario in order to have a fine-grained control over the flow of interactions between the system and the environment. Without such local states, scenario descriptions can become fairly clumsy and hard to read. On the other hand, having a local state helps keeping a clear distinction between state information that is needed to specify the functionality of the system and state information that is used to drive a specific scenario.

When writing down a scenario, one can easily think of different concrete textual and maybe even graphical notations. Based on our observations above, we opted for minimally extending the language of CoreASM, yet allowing a scenario designer to employ the full power of all the other available language features. In that sense, a scenario can be seen as a small “*specification script*” for a separate machine acting as the environment which drives the execution of the machine that needs to be validated or tested. This view actually fits very naturally into the overall design philosophy of the CoreASM framework where the core functionality of the execution environment can be easily extended by providing additional loadable plugins. Following that architectural style, we have designed and implemented a plugin, called Bârun, that offers a powerful scripting language on top of the standard execution engine.

The rest of this paper is structured as follows: We start with describing the requirements and design decisions in Section 2 and provide the specification of the Bârun plugin and details on how the plugin extends the CoreASM framework in Section 3. We continue with a case study in Section 4 that exemplifies the main features of the scripting extension. Section 5 offers a short overview on related work followed by our conclusions and plans for future work in Section 6.

2 Requirements and Design Decisions

The need for better simulation support results from experiences that one of the authors made while trying to reverse engineer an executable specification for an application (developed within SAP) that deals with consistent object management in a dynamic and distributed environment [5]. It turned out that the overall complexity and dynamism made validation very difficult. Actually, most of the time was spent into considering corner and error cases, probably a typical phenomenon in distributed systems that employ coordination protocols of some kind. What complicated matters even further was the fact that the system tries hard to deal with two orthogonal features *in parallel*: Object management is continuously supported even in the event of a dynamic restructuring of the distributed system (landscape). It soon became clear that we need to be able to test and validate independent system properties *in isolation*, e.g., checking the robustness of the failover protocol with respect to changes in the landscape while disabling the orthogonal object management part. As we will see in Section 4, some of the validation scenarios actually require very fine-grained control over the execution of the underlying system in order to explore the state space into a certain direction.

In summary, we would like a simulation scripting tool for CoreASM that would allow us to:

1. write scenario scripts that could be used to guide the execution of CoreASM specifications;
2. define assertions that would be checked after each successful step of the simulated machine;

3. write scenario scripts in form of ASM rules that can potentially modify the state of the simulated machine between the computation steps of the simulated machine;
4. use the same set of rule constructs that are available in the CoreASM specification for which the scenario is written.

Based on the requirements listed above, we designed a language for Bârun scenarios that would support the declaration of the following four scenario components: 1) local functions, i.e., locations that are local to the scenario, 2) option statements, borrowing the grammar rule offered by the Options plugin [1], 3) assertions, and 4) ASM macro rules. The following EBNF grammar captures the structure of scenarios. A scenario starts with the keyword ‘Scenario’ and a scenario identifier followed by a collection of declarations, including one statement to declare the main rule of the scenario.

```
Scenario ::= ‘Scenario’ ID
           ScenarioDeclaration* MainRule ScenarioDeclaration*
ScenarioDeclaration ::=
           LocalFunction | Option | Assertion | RuleDeclaration
MainRule ::= ‘main’ ID
LocalFunction ::= ‘local’ ID
Option ::= ‘option’ ID Term
```

Assertions are defined by an identifier, the assertion term and an optional output term to be printed when the assertion fails.

```
Assertion ::= ‘assert’ ID ‘as’ Term (‘onfailure’ ‘print’ Term)?
```

In order to support definition of ASM macro rules in the scenario, Bârun borrows RuleDeclaration grammar rule from the CoreASM kernel.

3 Extending CoreASM

Bârun could be implemented either as an external tool that utilizes the CoreASM engine to run specifications according to the scenario, or as a plugin that directly extends the engine and “injects” scenarios into the normal execution of specifications. Considering the pros and cons of each approach, we opted for a plugin implementation based on the following arguments:

- In order to offer a rich scripting language, we want the Bârun language to offer the same ASM rule constructs that are available in CoreASM specifications. Since CoreASM plugins have access to the internal components of the engine, Bârun implemented as a plugin can easily extend the dynamic grammar of CoreASM specifications and utilize different parts of the CoreASM language that is provided by the engine.
- Implementing Bârun as an external tool would require replication of an *engine driver* that controls the execution process of the engine. While there are such drivers already available for CoreASM (such as Carma and the CoreASM Eclipse Plugin [1]), implementing Bârun as a plugin integrates the simulation scripting features into the currently available user interfaces of CoreASM.

- As a plugin, Bârun can extend the CoreASM language to allow declaration of a set of *reasonable* or pre-defined scenarios into the specification itself. These scenarios can be shipped with the specification as test cases or demo scripts.

3.1 Extending the Control Flow of the Engine

In order to load a scenario, check assertions and apply the scenario related changes to the state of the simulated machine, Bârun extends the Control State ASM of the CoreASM engine at four extension points: 1) after the engine parses the specification and before it initializes the state, 2) before the engine starts a computation step, 3) after the engine selects the subset of agents for the next computation step, and 4) when the engine completes a successful step [1].

Before the CoreASM engine starts the execution of a specification, Bârun has to utilize the specification-specific parser of the engine to load the active scenario. This has to be done after the engine has created its parser and loaded the specification. Thus, Bârun extends the control state ASM of the engine at the point where the specification is loaded and the engine is ready to prepare the initial state of the simulated machine; this is when the control state of the engine switches to *Initializing State*.

The idea here is to interleave the normal execution of CoreASM specifications with execution of the main rule of the scenario. Bârun introduces a new CoreASM agent with the main rule of the scenario as its program and modifies the control flow of the engine so that on every other step the engine will choose only the Bârun agent for execution. This is implemented by temporarily setting *bârunAgent* as the only available agent in the state during the phase in which CoreASM engine is choosing the next set of agents to be executed. To keep a consistent view of the state, once *bârunAgent* is chosen, the agents set is set back to the actual agents set so that the value of *Agents* is kept consistent during both “normal” and scenario executions. In order to provide this interleaving behavior, Bârun maintains a flag called *bârunTurn*, initially set to *false*, that indicates if it is time to switch agents set to $\{\textit{bârunAgent}\}$.

After every successful update, Bârun evaluates the assertions and notifies the user (or terminates the run, depending on the configuration of Bârun) if any assertion fails. The behavior of Bârun during the aforementioned control state transitions is specified by the $\textit{FireOnModeTransition}_{\textit{Bârun}}$ rule defined on page 51.

Before CoreASM engine initializes the state, the ASM rules *LoadScenarioDeclarations* and *LoadActiveScenario* (above) respectively load the scenario declarations of the specification together with the main (*active*) scenario that will be executed by the Bârun plugin. In the following subsections, we discuss the other three extension points in more detail.

Evaluating the Main Rule: The Bârun plugin extends the CoreASM engine such that it runs the Bârun agent (and only the Bârun agent) every other step of the run. Just before the CoreASM engine starts a step, if it is time for the Bârun

```

FireOnModeTransitionBârun(sourceMode, targetMode) ≡
  case targetMode of
    initializingState :
      bârunTurn := true
      seq LoadScenarioDeclarations
      next LoadActiveScenario

    startingStep :
      if bârunTurn ∧ isScenarioActive then SwitchToBârunAgent

    initiatingExecution :
      if bârunTurn ∧ isScenarioActive then SwitchToSpecAgents
      bârunTurn := ¬bârunTurn

    stepSucceeded :
      if bârunTurn then
        CheckAssertions
        CheckScenarioProgram
      endcase
  where
    isScenarioActive ≡ scenarioMainRule ≠ undef ∧ ¬isScenarioTerminated

```

agent to be executed, Bârun makes a cache copy of the agents set (the *Agents* universe), then replaces the agents set with $\{\textit{bârunAgent}\}$. If it is the first step of the scenario ($\textit{isScenarioProgramInitialized} = \textit{false}$), it also sets the program of *bârunAgent* to the main rule of the scenario.

```

SwitchToBârunAgent ≡
  let agents = stateUniverse(state, "Agents") in
    specAgentsSet := enumerate(agents)
    forall a ∈ enumerate(agents) do
      memberue(agents, a) := false
      memberue(agents, bârunAgent) := true
    if ¬isScenarioProgramInitialized then
      SetValue(("program", ⟨bârunAgent⟩), scenarioMainRule)
      isScenarioProgramInitialized := true

```

Once *bârunAgent* is chosen for execution, Bârun sets the value of *Agents* universe back to its original value.

Checking Assertions: The Bârun plugin checks assertions after every successful step of the original specification; i.e., when the engine mode is changing to *Step Succeeded* and *bârunTurn* is *true* (indicating that the next step is Bârun's turn).

```

CheckAssertions  $\equiv$ 
  if  $|assertions| > 0$  then
    forall  $a \in assertions$  do
      let  $exp = CopyTree(assertionBody(a), false)$  in
        seq
          Evaluate( $exp$ )
        next
          if  $value(exp) \neq true_e$  then
            PrintAssertionFailed( $a$ )
            if  $assertionPrintExpr(a) \neq undef$  then
              EvaluateAndPrintAssertionPrintExpr( $a$ )
            if  $isTerminalAssertion(a)$  then
              ShutdownOnAssertionFailure( $a$ )

```

Bârun scenarios may define a list of *terminal assertions*, failure of which should terminate the simulation. The set can be defined in form of a comma separated list of assertion identifiers as the value of `Baarun.terminateOn` engine property. In order to make Bârun terminate on any assertion failure, the value of this property should be set to ALL.

The predicate $isTerminalAssertion(a)$ is defined as follows,

$$isTerminalAssertion(a) \equiv terminalAssertions = \{ "ALL" \} \vee assertionId(a) \in terminalAssertions$$

where $terminalAssertions$ is the set of assertion identifiers listed as the value of `Baarun.terminateOn` property.

Terminating the Bârun Agent: Scenarios may create an environment for the simulated machine, guide the execution to some point and then let the machine run without further interference. In such cases, the scenario writer may want to explicitly stop the execution of *bârunAgent* by setting its program to *undef*. After every successful step, the Bârun plugin monitors the program of *bârunAgent*, and if it is set to *undef*, the plugin sets a flag ($isScenarioTerminated$) that would prevent Bârun from further switching the agents set to $\{bârunAgent\}$.

```

CheckScenarioProgram  $\equiv$ 
  if  $isScenarioProgramInitialized \wedge \neg isScenarioTerminated$  then
    if  $getValue(("program", \langle bârunAgent \rangle)) = undef_e$  then
       $isScenarioTerminated := true$ 

```

3.2 Extending the Parser

The Bârun plugin extends the CoreASM language by adding the following declaration syntax to define scenarios (mapping of scenario identifiers to scenario script files) and an optional *active* scenario for the specification:

```
Header ::= ScenarioDeclarations | Header
ScenarioDeclarations ::=
    ScenarioDeclaration | ActiveScenarioDeclaration
ScenarioDeclaration ::= 'scenario' ID ANYTOKEN
ActiveScenarioDeclaration ::= 'active' 'scenario' ID
```

where `Header` is the grammar rule for header declarations of CoreASM.

For example, the following lines can be added to a specification to define a set of scenarios for the specification and to mark one of them as the default active scenario that will be taken into account in every run of the specification:

```
scenario communicationError "commerror.csc"
scenario invalidInput "invalidinput.csc"
scenario newPurchaseRequest "newpurchase.csc"
active scenario invalidInput
```

The active scenario can also be defined by setting the value of engine property `Baaron.scenario`. This can be done either explicitly in the specification by using the Options plugin, for example

```
option Baaron.scenario "invalidinput.csc"
```

or by setting the value of this property through the engine driver. For instance, using `Carma`, one can set the values of engine properties with the “-D” option:

```
carma -D Baaron.scenario=invalidinput.csc myspec.coreasm
```

3.3 Extending Rule Declarations

In order to facilitate writing structured scenario scripts, Bârun supports the declaration of new macro rules in simulation scenarios. Since we are using the CoreASM engine interpreter to evaluate the main rule of the scenario in the state of the simulated machine, these scenario-specific rule declarations has to be added to the state of the simulated machine as well. Thus, Bârun implements the *Vocabulary Extender* interface of CoreASM plugin architecture to extend the set of macro rule declarations in the initial state of the simulated machine by the rule declarations specified in the active scenario.

4 Case Study

We now provide a more detailed example scenario that demonstrates the different features of Bârun. For this, we re-examine the application introduced in Section 2, i.e., a distributed system that deals with consistent object management in a dynamic environment. For the purpose of the following example scenario it is sufficient to know that “dynamic environment” refers to the fact that the system operates as a set of cooperating *nodes* that form a *cluster* and nodes may join or leave that cluster either intentionally or unintentionally (upon node failure).

Consistent object management requires that all nodes *eventually* agree on the cluster topology (size and order), which should be ensured by a message-based cluster membership and failover protocol. This protocol is based on the following assumptions:

- There is a dedicated *master* node that acts as the “source of truth” regarding the current cluster topology. Nodes that join the cluster contact the master to receive information about the current cluster topology and then start a *restructuring protocol* with all existing cluster members to establish an updated consistent view among all cluster members.
- Nodes that intentionally leave the cluster perform a similar restructuring operation informing the remaining cluster members about their departure.
- A system-wide global *master lock* indicates whether there is an active master node. Any node that starts up tries to become the master (by trying to acquire the lock) and if that fails, will contact the current master to join the cluster.
- At any point in time, there is at most one restructuring going on, i.e., joins and intentional leaves are serialized.
- Cluster nodes will be notified about node failures. Upon such notification, the active master node performs a restructuring to repair the cluster view.
- If the master node has failed, the remaining nodes compete for becoming the new master. Only the one that wins will perform a restructuring. All other nodes simply contact the new master to be informed about the new cluster topology.

We have turned this protocol into a multi-agent CoreASM specification consisting of 10 modules. For each cluster node, we have 5 agents running, performing different tasks in the overall protocol. Each agent is modeled as a control state ASM where the overall number of different control states is 79. The following table provides some details on the complexity of those modules:

MODULE	LINES	RULES	FUNCTIONS
Control State Handling	63	5	9
Control ASM States	50	0	1
Cluster Environment (Notification)	328	19	31
Lock Management	141	7	19
Message Passing	362	12	56
Cluster Master	161	12	10
Protocol Messages	138	0	25
Control Flow	88	10	5
Object Requests	128	10	3
Cluster Membership and Object Management	1796	114	159
Total	3255	189	318

As all these pieces are specified as modules, we use an additional minimal “driver” specification for execution runs.

```

1  CoreASM ClusterEnvironment
2
3  // used modules omitted for brevity
4
5  include ClusterProtocol.coreasm
6
7  function nodeList: -> SET
8
9  init Init
10
11 rule Init = {
12     nodeList := {"N1", "N2", "N3"}
13     // delay startup, so that scenario runs have a chance to
14     // change the list of nodes that should be started
15     program(self) := @Startup
16 }
17
18 rule Startup = {
19     StartCluster(nodeList)
20     program(self) := undef
21 }

```

As with any other distributed coordination protocol, it soon became clear that we need to simulate protocol runs for exceptional cases, especially situations where nodes leave the cluster unintentionally. When we started our work, we thought we would need to spend most of our efforts into simulating message transmission errors. But it soon turned out that the system takes a fairly defensive approach for dealing with such errors: most of the time, a message transmission failure will lead to a node restart. Thus, we decided to focus on exploring the alternative paths with regard to cluster topology changes and failover handling. While investigating that, we realized that the original failover protocol was based upon an faulty assumption, namely that notifications in the case of failure would be sent *immediately after the node failure*. But one can easily think of scenarios where this is not true: Just assume that the notification is delayed while a new node is starting up. Then, that node will become the master of a new cluster that would just consist of that one node. If the delayed notification is then passed on to the remaining nodes from the old cluster, they will try to become master, will all fail, and thus do nothing, assuming that the (unknown) winner will perform the outstanding restructuring. Since the new master is not aware of the old cluster, no repair will happen and we will end up with two independent clusters operating in parallel.

Here is the complete scenario script that will expose this problem after executing 896 steps:

```

1  Scenario ClusterSplit
2
3  local scenarioPhase           initially 0
4  local clusterIsStable        initially false
5  local killedMaster
6  local suspendedAgents       initially {}
7
8  main ClusterController
9
10 rule ClusterController = {
11     if (scenarioPhase = 0) then {

```

```

12         nodeList := {"N1", "N2"}
13         scenarioPhase := 1
14     }
15     if (scenarioPhase = 1) then {
16         if (AllNodesRunning()) then {
17             SuspendElemLossHandlers()
18             scenarioPhase := 2
19             clusterIsStable := true
20         }
21     }
22     if (scenarioPhase = 2) then {
23         KillMaster()
24         scenarioPhase := 3
25         clusterIsStable := false
26     }
27     if (scenarioPhase = 3) then {
28         if (NodeIsDown(killedMaster)) then {
29             StartNewMaster("NM")
30             scenarioPhase := 4
31         }
32     }
33     if (scenarioPhase = 4) then {
34         if (MasterNode() != undef and HasJoinedCluster("NM")) then {
35             ResumeElemLossHandlers()
36             scenarioPhase := 5
37         }
38     }
39     if (scenarioPhase = 5) then {
40         if (AllNodesRunning()) then {
41             clusterIsStable := true
42             scenarioPhase := 6
43         }
44     }
45 }
46
47 assert Indices_In_Sync as clusterIsStable implies IndicesInSync()
48
49 // additional rules
50 rule SuspendElemLossHandlers = {
51     forall node in RunningNodes() with node != MasterNode() do {
52         suspend ElemLossHandlerAgent(node)
53         add ElemLossHandlerAgent(node) to suspendedAgents
54     }
55 }
56
57 rule ResumeElemLossHandlers = {
58     forall agent in suspendedAgents do resume agent
59 }
60
61 rule KillMaster = {
62     killedMaster := MasterNode()
63     remove NodeID(MasterNode()) from nodeList
64     SignalNodeShutdown(MasterNode(), true)
65 }
66
67 rule StartNewMaster(newMasterID) = {
68     AddNode(newMasterID)
69     add newMasterID to nodeList
70 }

```

The scenario script is modeled as a control state ASM guarded by the local variable *scenarioPhase*. The correctness of the object management depends on distributed information that needs to be in sync across all members of a cluster, but only if the cluster is considered to be in a *stable state*, i.e., all nodes are and there is no restructuring going on. This property is stored in the boolean

variable *clusterIsStable*, which in turn is used to guard the assertion in line 49. Note that we have only two scenario phases where the cluster is considered stable; once the original set of nodes has completed startup (line 16) and once the new (single-node) cluster has finished rearrangement (line 34). In line 17, the scenario actively suspends all notification handler agents to delay node failure handling, which is needed to trigger the bug in the failover protocol. Before we can start the node, that is supposed to become the new master, we need to wait until the old master node has completely stopped operation. To check this, we keep a reference to the old master node in the local variable *killedMaster*. Once the new master is up (line 34), we resume the suspended notification handler agents again (line 35). Which agents have been suspended is stored in another local variable *suspendedAgents*. When the second cluster arrangement has finished, we set *clusterIsStable* to *true* (line 41), activating the assertion again, which now fails.

Looking at the initial phase and the additional rules in the scenario script, we see that there is a direct coupling with the driver specification from above via the shared location *nodeList*. Actually, this location is also used to define some of the predicates that guard the execution of the scenario which we have added to the driver specification to allow other scenarios to reuse these predicates:

```

1  derived AllNodesRunning =
2    |{n | n in ClusterNode with
3      NodeID(n) memberof nodeList and IsClusterMember(n)}| = | nodeList |
4
5  derived MasterNode = Node(MasterQueueAgent())
6
7  derived IsClusterMember(node) =
8    Regular(node) and ansPhase(node) = undef
9
10 derived HasJoinedCluster(nodeID) =
11 |{n | n in ClusterNode with
12   NodeID(n) = nodeID and IsClusterMember(n)}| = 1

```

5 Related Work

Scenarios have been used at various stages of the software development process, such as requirements elicitation and validation [6], evaluation and analysis of software architectures [7,8], and software validation and testing [9]. When combined with executable specifications or abstract prototypes, scenarios are specially beneficial for design validation at the early stages of the software development process [6].

The authors in [10] offer a nice overview of the role of scenarios in the software development process. In this work, we focused on the application of scenarios for validation and testing of ASM models. Among the various ASM tools currently available (see [1, Ch. 3]), only two provide support for utilizing scenarios in design and modeling of software systems. Spec# in combination with SpecExplorer [11,12] is extended in [13] to support scenario-oriented modeling, extending earlier work on the use of the AsmL test tool for generating finite state machines from use-case models [14]. The authors have also developed an engine, utilizing SpecExplorer, to perform conformance checking for scenarios.

The AVaLa scenario scripting language together with the Asmeta V validator developed for the Asmeta toolset offer scenario-based validation of ASM models [10]. AVaLa offers a minimal scripting language, with the ability to execute ASM rules written in the AsmetaL language, to define scenarios for any given ASM specifications written in AsmetaL [15]. AVaLa scripts have a sequential semantics (unlike ASM specifications) that facilitate algorithmic description of scenarios. The scenarios are executed by the Asmeta V validator, a tool different from the AsmetaS simulator for executing AsmetaL specifications, and they explicitly *drive* the execution of their corresponding ASM specifications with statements such as ‘Step’ and ‘StepUntil’. In addition to executing the scenarios, Asmeta V validator also reports rule coverage by printing the list of rules that have been called and evaluated during the execution. This is quite useful to check if the set of scenarios cover all the transition rules of the original model.

Our approach here is similar to the work of Carioni et al. [10] as we defined a minimal language to write scenario scripts that will be executed against CoreASM specifications. However, the two approaches differ in two main aspects. First, a Bârun scenario is in fact a specification of the environment of the original system, so it is very much like a CoreASM specification (e.g., no implicit notion of sequentiality). Second, the CoreASM engine is extended by the Bârun plugin to run scenarios along with their corresponding CoreASM specification in an interleaving fashion. There is no separate tool required to run the scenarios and as a result it is not the scenarios that drive the specification.

Third, AVaLa has no means to capture and aggregate local state, which we found crucial for supporting scenarios like the one described in Section 4. There, we needed a way to steer the execution run in a very fine-grained manner in order to delay certain operations (handling node failure notifications). If all that information were available from the environment of the scenario, a rough sketch of an AVaLa based scenario (in CoreASM) could look like this:

```

1  set nodeList = {"N1", "N2"} // the nodes that will be started up
2  step until AllNodesRunning()
3  check IndicesInSync()
4  exec {
5      KillMaster()
6      SuspendElemLossHandlers()
7      remove NodeID(MasterNode()) from nodeList
8  }
9  step until NodeIsDown(killedMaster)
10 exec {
11     StartNewMaster("NM")
12 }
13 step until MasterNode() != undef and HasJoinedCluster("NM")
14 exec {
15     ResumeElemLossHandlers()
16 }
17 step until AllNodesRunning()
18 check IndicesInSync()

```

It looks compellingly short compared to the Bârun script, mainly because the sequential execution of the control state is already defined by the semantics of the language constructs and does not need to be explicitly modeled. But without local state in the scenario, that information would need to become part of the specification in order to be accessible in the scenario. While this seems tolerable

for *nodeList*, *killedMaster* and *suspendedAgents* would introduce rather “artificial” locations into the specification and would essentially blur the distinction between specification and scenario just for the sake of making the scenario work.

6 Conclusion and Future Work

Scenarios play an important role in software development, in particular in requirement elicitation and software validation and testing. In combination with executable specifications, executable scenario descriptions support validation of abstract models at the early stages of the design, hence reducing the cost of design failures. In this paper we have introduced Bârun, a plugin for CoreASM allowing specification and execution of scenarios for CoreASM models. The plugin extends the control flow of the CoreASM engine to enable execution of scenarios along with the execution of the original model.

We have successfully applied Bârun in an industrial use case. We were able to detect and reproduce design errors in a failover protocol of a distributed application by capturing the implemented algorithms in an executable CoreASM specification and validate this specification through dedicated scenario runs written as Bârun scripts.

The Bârun plugin utilizes the CoreASM language such that all the rule constructs and expression forms that are available in the original specification are also available in their corresponding scenarios. As such, with respect to the dynamic aspects of scenarios descriptions, the Bârun language can grow with the CoreASM language as new plugins are developed. This led to a minimal first solution, however, we envision various improvements to the Bârun plugin as part of future work.

As of today, sequential scenario execution has to be modeled explicitly, but this could be alleviated by providing scenario designers with a higher-level syntax (in a similar fashion as the AValLa language). These high-level scripts could then be automatically translated into their basic language equivalents without the need to extend the core language.

As mentioned in Section 3, Bârun scenarios are executed along with CoreASM specifications in an interleaving fashion. The plugin can be extended to offer scenario writers more control over this behavior, for example, to allocate more steps to the Bârun agent before resuming the execution of the CoreASM model. A possibly better approach would be to use model composition to combine scenarios with CoreASM models, following a similar approach suggested in [11].

Bârun can also be extended to offer different execution modes with respect to type-checking (such as *black-box* or *white-box*), hence potentially restricting access to the internal state of the original machine to help with error checking. Currently, the Modularity plugin cannot be used inside Bârun scenarios. Extending the plugin with modularity features would allow scenarios (e.g., those written for a certain specification) to share certain definitions. We have also considered adding support for various simulation reports (such as rule coverage), however we believe such features to be orthogonal and can be added in form of an additional plugin for CoreASM.

We plan to make future versions of the Bârun plugin publicly available.

Acknowledgements. We would like to thank the anonymous reviewers for their constructive feedback and suggestions for improvement on this paper.

References

1. Farahbod, R.: CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems. PhD thesis, Simon Fraser University, Burnaby, Canada (May 2009)
2. Farahbod, R., et al.: The CoreASM Project (2009), <http://www.coreasm.org>
3. Börger, E.: Abstract state machines: a unifying view of models of computation and of system design frameworks. *Ann. Pure Appl. Logic* 133(1-3), 149–171 (2005)
4. Carroll, J.M.: Five reasons for scenario-based design. In: HICSS '99: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences, Washington, DC, USA, vol. 3, p. 3051. IEEE Computer Society, Los Alamitos (1999)
5. Altenhofen, M., Brucker, A.: Practical issues with formal software specifications: Lessons learned from an industrial case study (submitted for publication, 2009)
6. Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenarios in system development: Current practice. *IEEE Software* 15(2) (1998)
7. Babar, M.A., Gorton, I.: Comparison of scenario-based software architecture evaluation methods. In: 11th Asia-Pacific Software Engineering Conference, pp. 600–607 (2004)
8. Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-based analysis of software architecture. *IEEE Software* 13(6), 47–55 (1996)
9. Ryser, J., Glinz, M.: A practical approach to validating and testing software systems using scenarios. In: QWE 1999, 3rd International Software Quality Week Europe (1999)
10. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 71–84. Springer, Heidelberg (2008)
11. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008)
12. Microsoft FSE Group: Spec Explorer (2008), <http://research.microsoft.com/specexplorer> (Last visited, July 2008)
13. Grieskamp, W., Tillmann, N., Veanes, M.: Instrumenting scenarios in a model-driven development environment. *Information and Software Technology* 46(15), 1027–1036 (2004)
14. Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N., Veanes, M.: Validating use-cases with the AsmL test tool. In: Proceedings of Third International Conference On Quality Software (QSIC 2003). IEEE, Los Alamitos (2003)
15. Formal Methods laboratory of University of Milan: Asmeta (2006), <http://asmeta.sourceforge.net/> (Last visited, June 2008)

AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications^{*}

Paolo Arcaini^{1,**}, Angelo Gargantini², and Elvinia Riccobene¹

¹ Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
parcaini@gmail.com, elvinia.riccobene@dti.unimi.it

² Dip. di Ing. Gestionale e dell'Informazione, Università di Bergamo, Italy
angelo.gargantini@unibg.it

Abstract. This paper presents *AsmetaSMV*, a model checker for Abstract State Machines (ASMs). It has been developed with the aim of enriching the ASMETA (ASM mETAmodeling) toolset – a set of tools for ASMs – with the capabilities of the model checker NuSMV to verify properties of ASM models written in the AsmetaL language. We describe the general architecture of AsmetaSMV and the process of automatically mapping ASM models into NuSMV programs. As a proof of concepts, we report the results of using AsmetaSMV to verify temporal properties of various case studies of different characteristics and complexity.

Keywords: Abstract State Machines, Model Checking, NuSMV, ASMETA.

1 Introduction

To tackle the growing complexity of developing modern software systems that usually have embedded and distributed nature, and more and more involve safety critical aspects, formal methods have been affirmed as an efficient approach to ensure the quality and correctness of the design. Formal methods provide several advantages when involved in software system engineering. They allow producing unambiguous specifications about the features and behavior of a system; they allow catching and fixing design errors and inconsistencies early in the design process; they allow applying formal analyses methods (validation and verification) that assure correctness w.r.t. the system requirements and guarantee the required system properties.

The Abstract State Machines (ASMs) [7] are nowadays acknowledged as a formal method successfully employed as systems engineering method that guides the development of complex systems seamlessly from requirements capture to their implementation. To be used in an efficient manner during the system development process, ASMs should be endowed with tools supporting the major

^{*} This work is supported in part by the PRIN project D-ASAP (Dependable Adaptable Software Architecture for Pervasive computing).

^{**} The author was partially supported by the STMicroelectronics project on *Model-driven methodologies and techniques for the design and analyses of embedded systems*.

software life cycle activities: editing, simulation, testing, verification, model exchanging, etc. It is also mandatory that these tools have to be strongly integrated in order to permit reusing information about models.

The goal of the ASMETA (ASM mETAmodeling) project [3] was to engineer a standard metamodel-based modeling language for the ASMs, and to build a general framework suitable for developing new ASM tools and integrate existing ones. Up to now, the ASMETA tool-set [14,3,12] allows creation, storage, interchange, Java representation, simulation, testing, scenario-based validation of ASM models.

In this work, we present a new component, *AsmetaSMV*, that enriches the ASMETA framework with the capabilities of the model checker NuSMV [2] to verify properties of ASM models.

As discussed in Sect. 5, it is relatively clear that a higher level specification formalism as that provided in terms of ASMs enables a more convenient modeling than that provided by the language of a model checker as NuSMV. On the other hand, it is undoubted that a lower-level formalism will lead to more efficient model checking. However, we believe that a developing environment where several tools can be used for different purposes on the base of the same specification model can be much more convenient than having different tools, even if more efficient, working on input models with their own different languages. On the base of our experience on some case studies (as the Mondex or the Flash Protocol, see Sect. 5), having a model checker integrated with a powerful simulator provides great advantages for model analyses, especially in order to perform model and property validation. Indeed, verification of properties should be applied when a designer has enough confidence that the specification and the properties themselves capture all the informal requirements. By simulation (interactive or scenario driven [8]), it is possible to ensure that the specification really reflects the intended system behavior. Otherwise there is the risk that proving properties becomes useless, for example in case of property *vacuously true* [16]. Moreover, a simulator can help to reproduce counter examples provided by a model checker, which are sometimes hermetic to understand.

The paper is organized as follows. Sect. 2 presents related results. In Sect. 3, we briefly introduce the ASMETA framework and the NuSMV model checker. Sect. 4 describes the general architecture of *AsmetaSMV* and the process of automatically mapping ASM models into NuSMV programs. In Sect. 5, we report the results of using *AsmetaSMV* to verify temporal properties of various case studies of different characteristics and complexity. Sect. 6 concludes the paper.

2 Related Work

There are several attempts to translate ASM specifications to the languages of different model checkers. For explicit state model checkers as Spin, we can cite [11] and [10]. In [11], the authors show how to obtain Promela programs from simple ASMs in order to use Spin for test generation. The approach was significantly improved in [10] where the authors reported their experience in using Spin for verifying properties of CoreAsm specifications.

Regarding NuSMV, which is a symbolic model checker, a preliminary work was done by Spielmann [19]. It represents ASMs by means of a logic for computational graphs that can be combined with a Computation Tree Logic (CTL) property and together they can be checked for validity. This approach severely limits the ASMs that can be model checked. That work was overtaken by the research of Winter and Del Castillo, which is very similar to ours. In [20], the author discusses the use of the model checker SMV (Symbolic Model Verifier) in combination with the specification method of ASMs. A scheme is introduced for transforming ASM models into the language of SMV from ASM workbench specifications. These schema are very similar to the scheme we present later in this paper. The approach was later improved in [9] and applied to a complex case study in [21]. A comparison with their work is presented in Sect. 5. Our approach is very similar to theirs, although the starting notation is different. Moreover, their tools (both the ASM workbench and the translator) are no longer maintained and we were unable to use them.

Other approaches to model checking ASMs include works which perform a quasi-native model checking without the need of a translation to a different notation. For example, [15] presents a model checking algorithm for AsmL specifications. The advantage is that the input language is very rich and expressive, but the price is that the model checking is very inefficient and unable to deal with complex specifications, and it is not able to perform all the optimizations available for a well established technique as that of Spin or NuSMV. A mixed approach is taken by [5], which presents a way for model checking ASMs without the need of translating ASM specifications into the modeling language of an existing model checker. Indeed, they combine the model checker [mc]square with the CoreASM simulator which is used to build the state space.

3 Background

3.1 ASMETA Toolset

The ASMETA (ASM mETAmodeling) toolset [14,3,13] is a set of tools for the ASMs developed by exploiting the Model-driven development (MDD) approach.

We started by defining a metamodel, the *Abstract State Machine Metamodel* (AsmM), as abstract syntax description of a language for ASMs. From the AsmM, by exploiting the MDD approach and its facilities (derivative artifacts, APIs, transformation libraries, etc.), we obtained in a generative manner (i.e. semi-automatically) several artifacts (an interchange format, APIs, etc.) for the creation, storage, interchange, access and manipulation of ASM models [12]. The AsmM and the combination of these language artifacts have led to an instantiation of the Eclipse Modeling Framework (EMF) for the ASM application domain. The resulting ASMETA framework provides a global infrastructure for the interoperability of ASM tools (new and existing ones) [13].

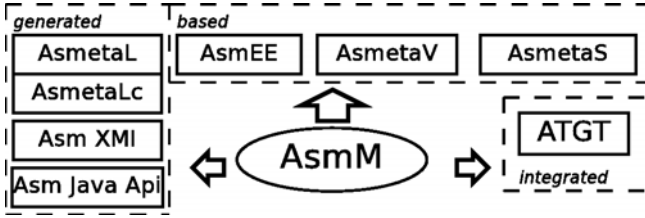


Fig. 1. The ASMETA tool set

The ASMETA tool set (see Fig. 1) includes (among other things) a textual concrete syntax, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse *AsmetaL* models and check for their consistency w.r.t. the AsmM OCL constraints; a simulator, *AsmetaS*, to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end called *ASMEE* (ASM Eclipse Environment) which acts as integrated development environment (IDE) and it is an Eclipse plug-in.

All the above artifacts/tools are classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate Ecore (i.e. the EMF metalanguage) projections to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the ASMETA environment and related derivatives; an example of such a tool is the simulator *AsmetaS*. Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

3.2 NuSMV

The NuSMV model checker [2], derived from the CMU SMV [17], allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL), using Binary Decision Diagrams (BDD)-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion.

NuSMV is a transactional system in which the states are determined by the values of variables; transactions between the states are determined by the updates of the variables. A NuSMV model is made of three principal sections:

- VAR that contains variables declaration. A variable type can be *boolean*, *Integer* defined over intervals or sets, *enumeration* of symbolic constants.
- ASSIGN that contains the initialization (by the instruction *init*) and the update mechanism (by the instruction *next*) of variables. A variable can be not initialized and in this case, at the beginning NuSMV creates as many states as the number of values of the variable type; in each state the variable

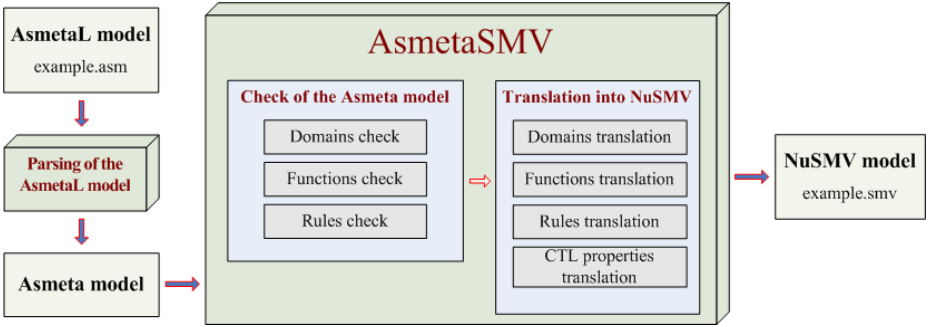


Fig. 2. Architecture of AsmetaSMV

assumes a different value. The next value can be determined in a straight way, or in a conditional way through the **case** expression.

- SPEC (resp. LTLSPEC) that contains the CTL (resp. LTL) properties to be verified.

In NuSMV it is possible to model *non deterministic behaviours* by (a) do not assigning any value to a variable that, in this case, can assume any value; (b) assigning to a variable a value randomly chosen from a set. It is also possible to specify *invariant conditions* by the command INVAR.

4 AsmetaSMV

AsmetaSMV has been developed as *based* tool of the ASMETA toolset, since it exploits some derivatives of the ASMETA environment. In particular, AsmetaSMV does not define its own input language, neither introduces a parser for a textual syntax. It reuses the parser defined for AsmetaL and reads the models as Java objects as defined by the ASMETA Java API. The aim of AsmetaSMV is that of enriching the ASMETA toolset with the capabilities of the model checker NuSMV. No knowledge of the NuSMV syntax is required to the user in order to use AsmetaSMV. To perform model checking over ASM models written in AsmetaL, a user must know, besides the AsmetaL language, only the syntax of the temporal operators.

Fig. 2 shows the general architecture of the tool. AsmetaSMV takes in input ASM models written in AsmetaL and checks if the input model is adequate to be mapped into NuSMV. Limitations are due to the model checker restriction over finite domains and data types. If this test fails, an exception is risen; otherwise, signature and transitions rules are translated as described in Sect. 4.1 and 4.2. The user can define temporal properties directly into the AsmetaL code as described in Sec. 4.3. We assume that the user provides the models in AsmetaL, but any other concrete syntax (like Asmeta XMI) could be used instead.

```

asm arity1_2
import ./StandardLibrary

signature:
  domain SubDom subsetof Integer
  enum domain EnumDom = {AA | BB}
  dynamic controlled foo1: Boolean -> EnumDom
  dynamic controlled foo2: Prod(SubDom ,
    EnumDom) -> SubDom
definitions:
  domain SubDom = {1..2}

```

```

MODULE main
VAR
  foo1_FALSE: {AA, BB};
  foo1_TRUE: {AA, BB};
  foo2_1_AA: 1..2;
  foo2_1_BB: 1..2;
  foo2_2_AA: 1..2;
  foo2_2_BB: 1..2;

```

Fig. 3. AsmetaL model and NuSMV translation

4.1 Mapping of States

Domains. AsmetaL domains are mapped into their corresponding types in NuSMV. The only supported domains are: Boolean, Enum domains and Concrete domains whose type domains are Integer or Natural. Boolean and Enum domains are straightforwardly mapped into boolean and symbolic enum types of NuSMV. Concrete domains of Integer and Natural, instead, become integer enums in NuSMV, on the base of the concrete domain definitions.

Functions. For each AsmetaL dynamic *nullary* function (i.e. variable) a NuSMV variable is created. ASM *n-ary* functions must be decomposed into function locations; each location is mapped into a NuSMV variable. So, the cardinality of the domain of a function determines the number of the corresponding variables in NuSMV. The codomain of a function, instead, determines the type of the variable. Therefore, given an *n-ary* function *func* with domain $Prod(D_1, \dots, D_n)$, in NuSMV we introduce $\prod_{i=1}^n |D_i|$ variables with names $func_elDom_1 \dots_elDom_n$, where $elDom_1 \in D_1, \dots, elDom_n \in D_n$.

Fig. 3 reports an example of two functions *foo1* of arity 1 and *foo2* of arity 2 and the result of the translation in NuSMV.

Controlled functions. They are updated by transitions rules. The initialization and the update of a dynamic location are mapped in the ASSIGN section through the *init* and *next* instructions. In Fig. 4, see the function *foo* as an example of a controlled function.

Monitored functions. Since their value is set by the environment, when mapped to NuSMV, monitored variables are declared but they are neither initialized nor updated. When NuSMV meets a monitored variable, it creates a state for each value of the variable. Values of monitored locations are set at the beginning of the transaction, that is before the execution of the transition rules; this means that transition rules deal with the monitored location values of the current state and not of the previous one. Therefore, when a monitored variable is used in

<pre>asm contrMon import ./StandardLibrary import ./CTLLibrary signature: dynamic monitored mon: Boolean dynamic controlled foo: Boolean definitions: //axiom for simulation axiom over foo: foo = mon //property to translate into NuSMV axiom over ctl: ag(foo = mon) main rule r_Main = foo := mon default init s0: function foo = mon</pre>	<pre>MODULE main VAR foo: boolean; mon: boolean; ASSIGN init(foo) := mon; next(foo) := next(mon); SPEC AG(foo = mon);</pre>
--	---

Fig. 4. Controlled and monitored functions

the ASSIGN section (this means that, in AsmetaL, the corresponding monitored location occurs on the right side of a transition rule), its correct value is obtained through the *next* expression.

This is shown by an example reported in Fig. 4 where the axiom checks that the controlled function `foo` is always equal to the monitored function `mon`. The correct NuSMV translation reports a CTL property, equivalent to the axiom, which checks that the NuSMV model keeps the same behaviour of the AsmetaL model.

Static and derived functions. Their value is set in the *definitions* section of the AsmetaL model and never changes during the execution of the machine. AsmetaSMV does not distinguish between static and derived functions, that, in NuSMV, are expressed through the DEFINE statement, as shown in Fig. 5. To obtain a correct NuSMV code, static and derived functions must be fully specified (i.e. specified in all the states of the machine), otherwise, NuSMV signals that the conditions of these function definitions are not exhaustive.

<pre>asm staticDerived import ./StandardLibrary signature: domain MyDomain subsetof Integer dynamic monitored mon1: Boolean dynamic monitored mon2: Boolean static stat: MyDomain derived der: Boolean definitions: domain MyDomain = {1..4} function stat = 2 function der = mon1 and mon2</pre>	<pre>MODULE main VAR mon1: boolean; mon2: boolean; DEFINE stat:= 2; der:= (mon1 & mon2);</pre>
---	--

Fig. 5. Static/derived functions

4.2 Mapping of Transition Rules

ASMs and NuSMV differ in the way they compute the next state of a transition, and such difference is reflected in their syntaxes as well.

In ASM, at each state, every enabled rule is evaluated and the update set is built by collecting all the locations and next values to which locations must be updated. The same location can be assigned to different values in several points of the specifications and the typical syntax of a single guarded update is **if cond then** $var' := val$.

In NuSMV, at each step, for every variable, the next value is computed by considering *all* its possible guarded assignments. The form of a guarded update is $var' :=$ **case** $cond1$ **then** $val1$ **case** $cond2$ **then** $val2$... which lists all the possible next values for the location.

In order to translate from AsmetaL to NuSMV, our translation algorithm visits the ASM specification. It starts from the main rule and by executing a depth visit of all the rules it encounters, it builds a *conditional update map*, which maps every location to its update value together with its guard. A global stack *Conds* (initialized to *true*) is used to store the conditions of all the outer rules visited. For each rule constructor, a suitable visit procedure is defined.

Update rule: The update rule syntax is $l := t$, where l is a location and t a term.

The visit algorithm builds c as the conjunction of all the conditions on the *Conds* stack and adds to the *conditional update map* the element $l \rightarrow (c, t)$

Conditional Rule: The conditional rule syntax is:

if $cond$ **then** R_{then} **else** R_{else} **endif**

where $cond$ is a boolean condition and R_{then} and R_{else} are transition rules. If $cond$ is true R_{then} is executed, otherwise R_{else} is executed.

The visit algorithm works as follows:

- $cond$ is put on stack *Conds* and rule R_{then} is visited; in such a way all the updates contained in R_{then} are executed only if $cond$ is true;
- $cond$ is removed from stack *Conds*.
- If else branch is not null:
 - condition $\neg cond$ is put on stack *Conds* and rule R_{else} is visited; in such a way all the updates contained in R_{else} are executed only if $cond$ is false;
 - $\neg cond$ is removed from stack *Conds*.

For example, the conditional update map of AsmetaL code shown in Fig. 6 is the following:

Location	Condition	Value
foo	$mon \wedge \neg mon2$	AA
	$mon \wedge mon2$	BB
foo1	<i>true</i>	AA

```

asm condRule
import ./StandardLibrary

signature:
enum domain EnumDom = {AA| BB| CC}
dynamic monitored mon: Boolean
dynamic monitored mon2: Boolean
dynamic controlled foo: EnumDom
dynamic controlled foo1: EnumDom

definitions:
main rule r_Main =
par
foo1 := AA
if(mon) then
if(mon2) then
foo := BB
else
foo := AA
endif
endif
endpar

```

```

MODULE main
VAR
foo: {AA, BB, CC};
foo1: {AA, BB, CC};
mon: boolean;
mon2: boolean;
ASSIGN
next(foo) :=
case
next(mon) & !(next(mon2)): AA;
next(mon) & next(mon2): BB;
TRUE: foo;
esac;
next(foo1) := AA;

```

Fig. 6. Conditional Rule mapping

Choose rule: The choose rule syntax is:

$$\text{choose } v_1 \text{ in } D_1, \dots, v_n \text{ in } D_n \text{ with } G_{v_1, \dots, v_n} \text{ do}$$

$$R_{v_1, \dots, v_n}$$

$$[\text{ifnone } R_{\text{ifnone}}]$$

where v_1, \dots, v_n are logical variables and D_1, \dots, D_n their domains. G_{v_1, \dots, v_n} is a boolean condition over v_1, \dots, v_n . R_{v_1, \dots, v_n} is a rule that contains occurrences of v_1, \dots, v_n . Optional branch **ifnone** contains the rule R_{ifnone} that must be executed if there are not values for variables v_1, \dots, v_n that satisfy G_{v_1, \dots, v_n} .

In NuSMV the logical variables become variables whose value is determined through an INVAR specification that reproduces the nondeterministic behavior of the choose rule. For each values tuple $d_1^{j_1}, \dots, d_n^{j_n}$ with $d_1^{j_1} \in D_1, \dots, d_n^{j_n} \in D_n$, the algorithm adds to the stack the condition $G_{d_1^{j_1}, \dots, d_n^{j_n}}$ and visits rule $R_{d_1^{j_1}, \dots, d_n^{j_n}}$.

Other rules: In addition to the update, conditional and choose rules, the other rules that are supported by AsmetaSMV are: macrocall rule, block rule, case rule, let rule and forall rule. They are not reported here. Details can be found in [4].

4.3 Property Specification

AsmetaSMV allows the user to declare CTL/LTL properties directly in the axiom section of an AsmetaL model.

¹ $G_{d_1^{j_1}, \dots, d_n^{j_n}}$ and $R_{d_1^{j_1}, \dots, d_n^{j_n}}$ are the condition and the rule where the variables v_1, \dots, v_n have been replaced with the current values $d_1^{j_1}, \dots, d_n^{j_n}$.

In AsmetaL, the syntax of an axiom is:

$$\mathbf{axiom\ over}\ id_1, \dots, id_n : \ ax_{id_1, \dots, id_n}$$

where id_1, \dots, id_n are names of domains, functions or rules; ax_{id_1, \dots, id_n} is a boolean expression containing occurrences of id_1, \dots, id_n .

In NuSMV, CTL [resp. LTL] properties are declared through the keyword SPEC [resp. LTLSPEC]:

$$\mathbf{SPEC}\ p_{CTL} \quad [\text{resp. } \mathbf{LTLSPEC}\ p_{LTL}]$$

where p_{CTL} [resp. p_{LTL}] is a CTL [resp. LTL] formula.

The syntax of a CTL/LTL property in AsmetaL is:

$$\mathbf{axiom\ over}\ [ctl \mid ltl] : \ p$$

where the over section specifies if p is a CTL or a LTL formula.

In order to write CTL/LTL formulas in AsmetaL, we have created the libraries *CTLlibrary.asm* and *LTLlibrary.asm* where, for each CTL/LTL operator, an equivalent function is declared. The following table shows, as example, all the CTL functions.

NuSMV CTL operator	AsmetaL CTL function
EG p	static eg: Boolean \rightarrow Boolean
EX p	static ex: Boolean \rightarrow Boolean
EF p	static ef: Boolean \rightarrow Boolean
AG p	static ag: Boolean \rightarrow Boolean
AX p	static ax: Boolean \rightarrow Boolean
AF p	static af: Boolean \rightarrow Boolean
E[p U q]	static e: Prod(Boolean, Boolean) \rightarrow Boolean
A[p U q]	static a: Prod(Boolean, Boolean) \rightarrow Boolean

AsmetaL code in Fig. 7 contains three CTL properties and their translation into NuSMV.

4.4 Property Verification

AsmetaSMV allows model checking an AsmetaL specification by translating it to the NuSMV language and directly run the NuSMV tool on this translation to verify the properties. The output produced by NuSMV is pretty-printed, replacing the NuSMV variables with the corresponding AsmetaL locations: it is our desire, in fact, to hide as much as possible the NuSMV syntax to the user.

The output produced by model checking the AsmetaL model shown in Fig. 7 is reported below. The first two properties are proved true: location `foo(AA)`, in fact, changes its value at each step. The third property, instead, is proved false as shown by the counterexample: a state exists where locations `foo(AA)`

```

asm ctlExample
import ./StandardLibrary
import ./CTLlibrary

signature:
  enum domain EnumDom = {AA | BB}
  controlled foo: EnumDom -> Boolean
  monitored mon: Boolean

definitions:
  //true
  axiom over ctl: ag(foo(AA) iff
                    ax(not(foo(AA))))
  //true
  axiom over ctl: ag(not(foo(AA)) iff
                    ax(foo(AA)))
  //false. Gives counterexample.
  axiom over ctl: not(ef(foo(AA) !=
                       foo(BB)))

  main rule r_Main =
    par
      foo(AA) := not(foo(AA))
      if(mon) then
        foo(BB) := not(foo(BB))
      endif
    endpar

default init s0:
  function foo($x in EnumDom) = true

```

```

MODULE main
VAR
  foo_AA: boolean;
  foo_BB: boolean;
  mon: boolean;
ASSIGN
  init(foo_AA) := TRUE;
  init(foo_BB) := TRUE;
  next(foo_AA) := !(foo_AA);
  next(foo_BB) :=
    case
      next(mon): !(foo_BB);
      TRUE: foo_BB;
    esac;
SPEC AG(foo_AA <-> AX(!(foo_AA)));
SPEC AG(!(foo_AA) <-> AX(foo_AA));
SPEC !(EF(foo_AA != foo_BB));

```

Fig. 7. CTL property translation

and `foo(BB)` are different (State: 1.2). Note that the pretty printer substitutes `foo_AA` with `foo(AA)`.

```

> AsmetaSMV ctlExample.asm
Checking the AsmetaL spec: OK
Translating to ctlExample.smv: OK
Executing NuSMV -dynamic -coi ctlExample.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct 4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

```

```

-- specification AG (foo(AA) <-> AX !foo(AA)) is true
-- specification AG (!foo(AA) <-> AX foo(AA)) is true
-- specification !(EF foo(AA) != foo(BB)) is false
-- as demonstrated by the following execution sequence

```

Trace Description: CTL Counterexample

Trace Type: Counterexample

```

-> State: 1.1 <-
  foo(AA) = 1
  foo(BB) = 1
  mon = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  foo(AA) = 0

```

5 Case Studies

AsmetaSMV has been tested on five case studies; the complete description of our tests can be found in [4] and are available at [3]. The first two case studies we have analyzed are two problems described in [6]:

1. A system made of two traffic lights placed at the beginning and at the end of an alternated one-way street; both traffic lights are controlled by a computer.
2. An irrigation system composed by a small sluice, with a rising and a falling gate, and a computer that controls the sluice gate.

For both problems we have written *ground* and *refined* model; in each model we have declared safety and liveness properties to test the correctness of the model.

Another case study we have analyzed is the Mondex protocol ([11]). The Mondex protocol implements electronic cash transfer between two purses; the transfer of money is implemented through the sending of messages over a lossy medium that can be, for example, a device with two slots or an Internet connection. We have written the AsmetaL model for one of the refinement steps described in [18]; a liveness property has helped us to discover that the model can enter in a deadlock state. Two different solutions are proposed in [4].

We have also analyzed the taxi booking problem: in a city some clients can request one or more taxis to a central that must satisfy all the requests. The taxis must bring the clients where they want to go. For this problem we had previously developed a NuSMV model (let's call it *originalNuSMV*); now we have developed an AsmetaL model containing the same properties that we wrote in the *originalNuSMV*. We have been able to compare the *originalNuSMV* code with the code obtained from the translation of the AsmetaL model (let us call it *mappedNuSMV*). We have seen that, for the same problem, it is easier to write an AsmetaL code rather than a NuSMV one: the ASMs in fact, thanks to a wide set of transition rules, are much more expressive than NuSMV. The verification of the properties in *originalNuSMV* and in *mappedNuSMV* gave the same results. Obviously this cannot be considered as a demonstration of the correctness of the mapping, but shows that, for a problem, there are different equivalent models. Generally, the code obtained from a mapping is more computational onerous than a code written directly in NuSMV; the mapping, in fact, introduces some elements that can be avoided in the direct encoding.

Finally, we have applied our tool to the flash cache coherence protocol, which integrates support for cache coherent shared memory for a large number of interconnected processing nodes. Starting from the specifications published by Winter [21] and by Farahbod at al. [10], we have written the AsmetaL specification for the protocol together with its safety properties. This model is available in the ASMETA repository. By means of the ASMETA simulator and the validator we were able to correct some defects in our specifications even before trying to prove the properties. A problem of vacuity detection also has been arisen. At the end, we were able to prove the three properties in less than 1 second for both the protocol versions with 2 nodes and 1 and 2 lines. A detailed comparison with [21] and [10] is however difficult since we were unable to run neither Asm2SMV

which is no longer maintained, nor the Coreasm to Spin plug-in which is not published yet. With respect to [21], our running times are much lower, but we ran the experiments on a faster machine. In terms of the BDD size of the resulting NuSMV model, we found that the specification with 1 line has similar size while our specification with 2 lines had a much smaller BDD size. Our experiments were much faster than that in [10], too, but only for one property we can actually compare our results with theirs, since they used the model checker Spin mainly to find faults in the original specification but we were unable to reproduce the same faults.

6 Conclusions

This work is part of our ongoing effort in developing a set of tools around ASMs for model validation and verification. We here describe how the ASMETA toolset has been enriched with model checking facilities to verify temporal properties of ASM models encoded in the AsmetaL language. By means of case studies of different complexity, we provide evidence of the importance of having simulation and model checking capabilities integrated within a unique environment. Indeed, the combined use of both tools can facilitate the verification process, since it may be sometimes useful to discover which system behavior is hidden behind a property to verify in order to better formulate it and easily prove it. As future plan, we intend to improve AsmetaSMV to handle turbo ASMs. Moreover, we plan to extend AsmetaSMV in order to allow the translation of counter examples produced by NuSMV to Avalla [8], the language we use to perform scenario driven validation of ASMs. The counter examples would constitute a set of wrong scenarios representing incorrect behaviors of the system, and they could be replayed later to check that corrected models do not exhibit those incorrect behaviors.

Acknowledgments. R. Farahbod sent us the CoreAsm specification of the flash coherence protocol, while K. Winter helped us in trying to make Asm2SMV [21] work on our computers.

References

1. Mastercard international inc.: Mondex, <http://www.mondex.com/>
2. The NuSMV website, <http://nusmv.itc.it/>
3. The ASMETA website (2006), <http://asmeta.sourceforge.net/>
4. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a model checker for AsmetaL models. tutorial. TR 120, DTI Dept., Univ. of Milan (2009)
5. Beckers, J., Klünder, D., Kowalewski, S., Schlich, B.: Direct support for model checking abstract state machines by utilizing simulation. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 112–124. Springer, Heidelberg (2008)
6. Börger, E.: The Abstract State Machines Method for High-Level System Design and Analysis. Technical report, BCS FACS Seminar Series Book (2007)

7. Börger, E., Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003)
8. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 71–84. Springer, Heidelberg (2008)
9. Castillo, G.D., Winter, K.: Model checking support for the ASM high-level language. In: Schwartzbach, M.I., Graf, S. (eds.) *TACAS 2000*. LNCS, vol. 1785, pp. 331–346. Springer, Heidelberg (2000)
10. Farahbod, R., Glässer, U., Ma, G.: Model checking coreasm specifications. In: Prinz, A. (ed.) *Proceedings of the ASM 2007, The 14th International ASM Workshop (2007)*
11. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to generate tests from ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) *ASM 2003*. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003)
12. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. *J. UCS* 14(12), 1949–1983 (2008)
13. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: The ASMETA case study. In: *International Conference on Software Engineering Advances, ICSEA*, pp. 373–378 (2008)
14. Gargantini, A., Riccobene, E., Scandurra, P.: Ten reasons to metamodel ASMs. In: Jean-Raymond, Glässer, U. (eds.) *Rigorous Methods for Software Construction and Analysis*. LNCS, vol. 5115, pp. 33–49. Springer, Heidelberg (2009)
15. Kardos, M.: An approach to model checking asml specifications. In: *Abstract State Machines*, pp. 289–304 (2005)
16. Kupferman, O., Vardi, M.: Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)* 4(2), 224–233 (2003)
17. McMillan, K.L.: *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell (1993)
18. Schellhorn, G., Banach, R.: A concept-driven construction of the mondex protocol using three refinements. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 57–70. Springer, Heidelberg (2008)
19. Spielmann, M.: Automatic verification of abstract state machines. In: Halbwachs, N., Peled, D.A. (eds.) *CAV 1999*. LNCS, vol. 1633, pp. 431–442. Springer, Heidelberg (1999)
20. Winter, K.: Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)* 3(5), 689–701 (1997)
21. Winter, K.: Towards a methodology for model checking ASM: Lessons learned from the FLASH case study. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) *ASM 2000*. LNCS, vol. 1912, pp. 341–360. Springer, Heidelberg (2000)

An Executable Semantics of the SystemC UML Profile*

Elvinia Riccobene¹ and Patrizia Scandurra²

¹ DTI Dept., Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

² DIIMM Dept., Università degli Studi di Bergamo, Italy
patrizia.scandurra@unibg.it

Abstract. The SystemC UML profile is a modeling language designed to lift features and abstractions of the SystemC/C++ class library to the UML level with the aim of improving the current industrial System-on-Chip design methodology. Its graphical syntax and static semantics were defined following the “profile” extension mechanism of the UML metamodel, while its behavioral semantics was given in natural language. This paper provides a precise and executable semantics of the *SystemC Process State Machines* that are an extension of the UML state machines and are part of the SystemC UML profile to model the reactive behavior of the SystemC processes. To this purpose, we used the meta-hooking approach of the ASM-based semantic framework, which allows the definition of the dynamic semantics of metamodel-based languages and of UML profiles.

1 Introduction

The SystemC UML profile [28,24] is a modeling language developed to improve the conventional industrial Systems-on-Chip (SoC) design methodology with a model-driven approach [25,26,27]. It is a consistent set of modeling constructs designed to lift both structural and behavioral features (including events and time features) of SystemC [32] to the UML [33] level. It was defined by exploiting the UML profile mechanism that requires the specification of UML extension elements (stereotypes and tagged values) and of new constraints as Object Constraint Language (OCL) [22] rules.

The profile, while provides a complete description of the modeling syntax and static semantics, suffers from the lack of a precise behavioral semantics that is given in natural language. Indeed, in the OMG framework used to define the profile, as well as in other metamodeling environments (like Eclipse/Ecore, GME/MetaGME, AM-MA/KM3, XMF-Mosaic/Xcore, etc.), the way to define the language *abstract syntax* in terms of a metamodel and its *static* semantics as OCL rules is well established, while no standard and rigorous support is given to provide the *dynamic* semantics that is usually expressed in natural language. This lack has negative consequences, as often remarked in the past since the first UML version. Moreover, defining a precise semantics of UML extensions is widely felt, especially now that UML is turning into a “family of languages” (see the OMG standardization activities of UML profiles in [33]).

* This work is supported in part by the PRIN project D-ASAP (Dependable Adaptable Software Architecture for Pervasive computing).

The definition of a means for specifying rigorously the semantics of UML profiles, as well as of metamodel-based languages, is therefore an open and crucial issue in the model-driven context.

In [11], a formal semantic framework based on the ASM (Abstract State Machine) formal method [2] is presented, which allows us to express a precise and executable semantics of metamodel-based languages using different techniques. We here adapt one of the techniques in [11], the *meta-hooking*, for UML profiles, and we show its application to the SystemC UML profile. This implies to provide a rigorous semantics of the *SystemC Process (SCP) state machines* formalism of the SystemC UML profile used to model the reactive behavior of SystemC processes.

This paper is organized as follows. Some background on the SystemC UML profile is given in Sect. 2. Sect. 3 presents the meta-hooking technique of the ASM-based semantic framework. Sect. 4 shows the application of the meta-hooking technique to the OMG metamodeling framework for the semantics specification of the SCP state machines. Some related work is presented in Sect. 5 while Sect. 6 concludes the paper.

2 The SystemC UML Profile

SystemC [32] is an open standard in the EDA (Electronic Design Automation) industry. Built as C++ library, SystemC is a language providing abstractions for the description and simulation of SoCs. Typically, the design of a system is specified as a hierarchical structure of modules and channels. A *module* is a container class able to encapsulate *structure* and *functionality* of hardware/software blocks, while a *channel* (primitive or hierarchical) serves as a container to encapsulate the *communication* functionality of blocks. Each module may contain *attributes* as simple data members, *ports* (proxy objects) for communication with the surrounding environment and *processes* for executing module's functionality and expressing concurrency in the system. Fig 1 shows a module example, `count_stim`, containing a thread process `stimgen`, two input ports `dout` and `clock`, and two output ports `load` and `din`, in the SystemC UML profile.

We here skip the details concerning the structural modeling constructs, as the focus is on the behavioral aspects of the profile. Some basic concepts underlying the SCP state machines are reported below as defined in the SystemC UML profile [28]. This formalism is to be considered a conservative extension of the UML *method* state machine¹ defined through the UML extension mechanism of “profiles” (i.e., stereotypes, tags, and constraints)[33].

SystemC Process State Machines: Processes are the basic unit of execution within SystemC and provide the mechanism for simulating concurrent behavior. Three kinds of processes are available: `sc_method`, `sc_thread` and `sc_ctypead`. Each kind of process has a slight different behavior, but in general (i) a process is declared within the scope of a class (a module or a hierarchical channel) as a stereotyped operation with no return type and no arguments (see, for example, Fig 1); (ii) all processes run

¹ A UML “method” state machine specifies the algorithm or procedure for a behavioral feature (such as a class’s operation).

concurrently; (iii) the process code is not *hierarchical*, i.e. no process can directly invoke another process (processes can cause other processes to execute only by notifying events); and (iv) a process is activated depending on its *static sensitivity* that is an initial list (possibly empty) of events that can dynamically change at run-time realizing the so called *dynamic sensitivity* mechanism. Finally, (v) all processes are usually activated at the beginning of the simulation, but a process can be explicitly not *initialized* – by means of a `dont_initialize` statement –, so it does not execute immediately when the simulation starts, but after a first occurrence of any of the events in its static sensitivity.

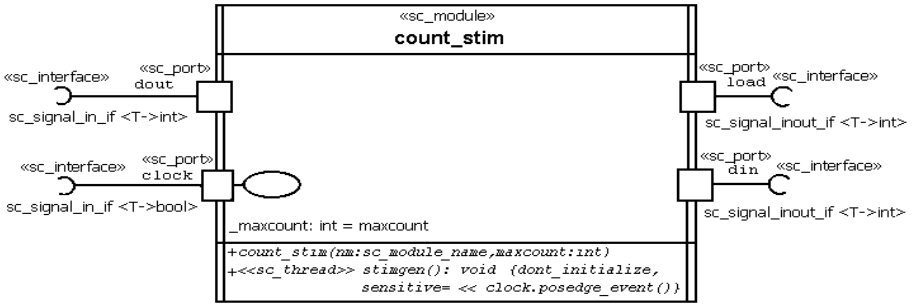


Fig. 1. The `count_stim` module

In this paper, we focus on the `sc_thread` process type. Fig. 2(A) shows the UML schema of a SC thread state machine. This diagram corresponds to a SystemC thread that: (i) has both a static (the list e_{1s}, \dots, e_{Ns}) and a dynamic sensitivity (the state WAITING FOR e^* with the stereotype `wait`), (ii) runs continuously (by an infinite while loop), and (iii) is not initialized (the state with the `dont_initialize` stereotype follows the top initial state). Activities `a1` and `a2` stand for structured blocks of sequential (or not) code without `wait` statements. The `wait`-state denotes a generic `wait(e*)` statement where the event e^* matches one of the cases described in Fig. 3. The pattern in Fig. 2(A) can be more complex in case of `wait` statements within the scope of nested control structures. In this case, as part of the SystemC UML profile, the control structures `while`, `if-else`, etc., need to be explicitly represented in terms of special stereotyped junction or choice pseudostates.

Fig. 2(B) shows the state machine for the `stimgen` thread of the module shown in Fig. 1. It is an instantiation of the pattern in Fig. 2(A). After initializing the `load` and `din` ports, the `stimgen` thread runs continuously: at each clock cycle (by the `wait` for the positive clock edge event of the static sensitivity list) it checks the received value from the `dout` port and may restart the counter in case it reaches the `_maxcount` attribute's value. Actions are specified using SystemC/C++ as action language.

The stereotype `sc_thread` labels both the operation indicating the thread process in the class of the container module (see, e.g., Fig. 1) and the state machine defined for the process (see Fig. 2(B)). The tag `sensitive` (see Fig. 1) is used to declare the static sensitivity list of the thread (if any) using the form `<< e1s << .. << eNs`, where

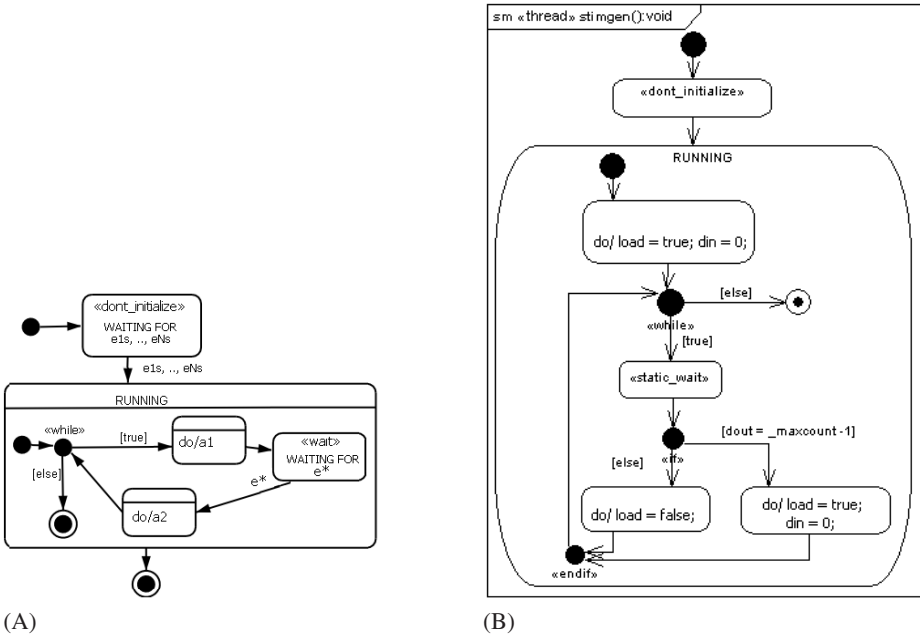


Fig. 2. A thread state machine pattern (A) and a (concrete) thread state machine (B)

e_{1s}, \dots, e_{Ns} are event types. The boolean tagged value `dont_initialize`, whose default value is `false`, represents the SystemC `dont_initialize` statement. The `dont_initialize` stereotype is also applied to a simple state (see Fig 2) and is used to capture at state machine level the behavioral semantics of the `dont_initialize` statement. A `dont_initialize` state has only one outgoing transition with possibly no explicit triggers; it is assumed that the static sensitivity list of the process are the implicit trigger event list of this transition.

The dynamic sensitivity of a thread is captured at behavioral level in the state machine associated to the thread by the use of the stereotypes `static_wait` and `wait`. These stereotypes are applied to simple states. They model the SystemC `wait()` and `wait(e*)` statements for resuming a waiting process depending on its static and dynamic sensitivity, respectively. A `static_wait` state has only one outgoing transition, the *static resuming transition*, with no explicit triggers since it is assumed that the events of the static sensitivity list of the process are the implicit triggers of this transition. The parameter e^* of a `wait(e*)` statement is the trigger of the outgoing transitions, the *dynamic resuming transitions*, of a `wait` state (see Fig 3).

3 ASM-Based Semantic Framework

The semantic framework presented in [11] is based on the ASM formal method and allows to link the abstract syntax (metamodel) of a language with its executable

SystemC	UML Notation
wait(e) wait for an event e	
wait(t) wait for t time units	
wait(e, t) wait for an event with timeout	
wait(e1&...&eN) wait for an AND-list of events	
wait(e1 ... eN) wait for an OR-list of events	
wait(t, e1&...&eN) wait for an AND-list of events with timeout	
wait(t, e1 ... eN) wait for an OR-list of events with timeout	
wait() wait for static sensitivity	

Fig. 3. Dynamic Sensitivity of a Thread

behavioral semantics expressed in terms of ASM transition rules. In the sequel, we recall from [11] some basic concepts.

A language metamodel A has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S : A \rightarrow S$ is provided [13] to give meaning to syntactic concepts of A in terms of the semantic domain elements. In the ASM-based semantic framework, the mapping M_S is defined in terms of the ASM metamodel, $AsmM$, and its semantic domain S_{AsmM} ². The semantics of a “terminal model”³ [15] conforming to A is therefore expressed in terms of an ASM model.

By assuming the semantic domain S_{AsmM} as the semantic domain S , the semantic mapping $M_S : A \rightarrow S_{AsmM}$ is defined as

$$M_S = M_{S_{AsmM}} \circ M$$

where $M_{S_{AsmM}} : AsmM \rightarrow S_{AsmM}$ is the semantic mapping of the ASM metamodel and associates a theory conforming to the S_{AsmM} logic with a model conforming to $AsmM$, and the function $M : A \rightarrow AsmM$ associates an ASM to a terminal model m conforming to A . Therefore, the problem of giving the metamodel semantics is reduced to define the function M between metamodels. The complexity of this approach depends on the complexity of building the function M .

² S_{AsmM} is the first-order logic extended with the logic for function updates and for transition rule constructors formally defined in [2].

³ A terminal model is a representation, that conforms to a reference metamodel, of a real world system (or portions of it).

Different ways of defining M were presented in [11], classified in *translational* and *weaving*, depending on the abstraction level of the metamodeling stack [15]. Going up through the metamodeling levels, these techniques allow increasing automation in defining model transformations, increasing reuse and decreasing dependency of the final ASM with respect to the terminal model. Among them, we here commit with the translational *meta-hooking* approach that works at meta-metamodel level, and allows us to exploit the definition of the function $\gamma : MOF \rightarrow AsmM$ (see below for details) defined in [11] and suitable for all languages whose metamodel is defined in terms of MOF. Here γ is adapted to handle also UML profiles, as the SystemC UML Profile.

Meta-hooking for MOF-based metamodels: This technique aims at automatically deriving (most of the part of) the signature of the resulting ASM from the source metamodel A and MOF. This resulting algebra is then endowed with ASM transition rules to capture the behavioral aspects of the underlying language. Finally, by navigating a specific terminal model m , the initial state is determined.

Formally, the function $M : A \rightarrow AsmM$ for a MOF-based metamodel A (such as UML or a UML profile) is defined as

$$M(m) = \iota(\omega(m))(\tau_A(\gamma(\omega(m))), m)$$

for all m conforming to A , where:

- $\gamma : MOF \rightarrow AsmM$ provides signature (domain and function definitions) of the final machine $M(m)$ from the metamodel $\omega(m)$ to which m conforms to,
- $\tau_A : AsmM \rightarrow AsmM$ provides the ASM transition rules capturing the behavioral aspects of A ,
- $\iota : MOF \rightarrow (AsmM \times A \rightarrow AsmM)$ is an HOT (High Order Transformation)⁴ and establishes, for a metamodel A , the transformation $\iota(A)$ that computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modeling elements in m .

Mappings γ and ι are *universal*, i.e. once defined for the MOF, they are applicable to all metamodels conforming to MOF, and therefore to the SystemC UML Profile.

4 Meta-hooking for the SystemC Process State Machines

We here exploit the meta-hooking technique of the ASM-based semantic framework to provide the operational semantics of the SCP state machines. To this purpose, as domain A of the function M , we do not need to consider the whole SystemC UML metamodel, but only its portion related to the abstract syntax for modeling state machines. Figures 4 and 5 shows a simplified⁵ portion of the UML metamodel (related to the state machines) together with the stereotypes definitions (only some elements) capturing specific features of the SCP state machines.

⁴ An HOT is a transformation taking as input or producing as output another transformation.

⁵ The effect of some OCL constraints of the SystemC UML profile is graphically emphasized by circles. They show that multiplicities have been restricted from many to exactly 1.

The semantics specification of the SCP state machines is captured by an ASM model obtained in three steps: (1) the γ mapping (see the MOFtoAsmM transformation rules provided in [11], Table 1) is applied to the portion of the UML metamodel related to the state machine formalism and to its extension through stereotypes to obtain the ASM signature; (2) the operational semantics of the SCP state machines is then defined by ASM transition rules as form of pseudo-code operating on the abstract data derived from step 1; finally, (3) the initial state of the ASM model for a terminal model (later referred SC-UML) conforming to the SystemC UML profile is provided by an HOT ι similar to the one defined in [11], Table 2.

Note that to fulfill step 1, the γ mapping provided in [11] is further extended here to handle the stereotypes of the UML profile mechanism. To this purpose, a *stereotype* is treated similarly to a class, and therefore mapped into a domain that is subset of the domain corresponding to the (extended) base class. *Tag definitions* of stereotypes are attributes of the stereotype class, and therefore they are mapped to ASM functions having as domain the ASM domain corresponding to the stereotype, and as codomain the ASM domain corresponding to the attribute type. Generalization relationships between stereotypes are mapped as for generalization relationships between classes.

The ASM model described here is an adaptation of the ASM model presented in [11] where an ASM semantics of the UML 1.x state machines is described. Although there are common parts, the model provided here takes into account the UML2 version and

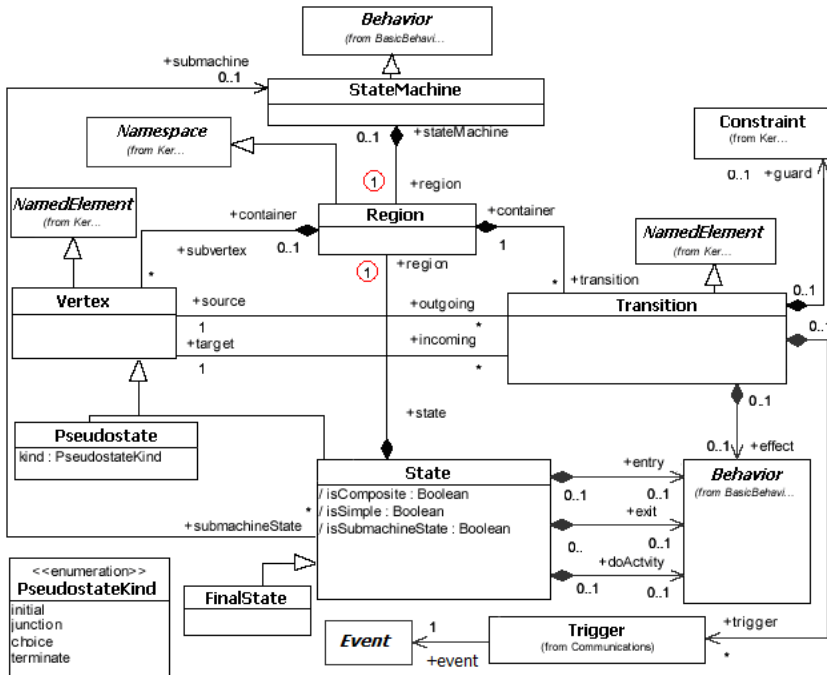


Fig. 4. SCP state machines metamodel (Part 1)

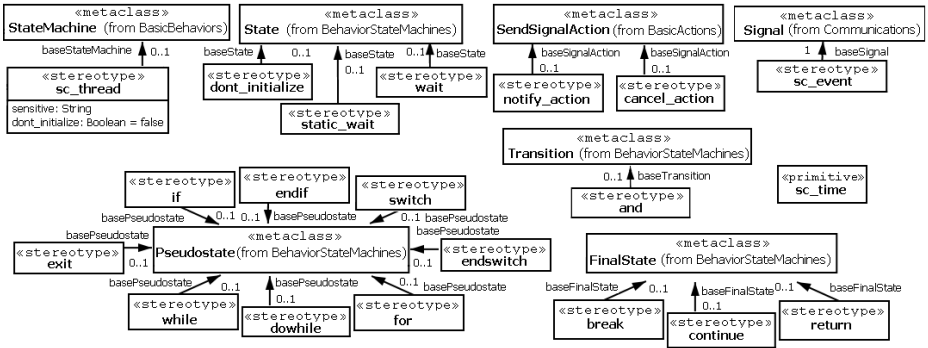


Fig. 5. SCP state machines metamodel (Part 2): some stereotypes

the restrictions and the specific behavioral features of the SystemC UML profile. Due to the lack of space, only a subset of the entire set of ASM transition rules is reported. The reader can find more details in the preliminary work [10] and in the implementation available at [16] using the ASMETA/AsmetaL language [8]. Moreover, the reader is assumed to be familiar with the semantics of the UML state machines.

4.1 ASM Signature

From the class diagram in Fig. 4 and the stereotypes in Fig. 5, a SCP state machine is a *sequential* state machine made up of just one Region, which in turn consists of (control) states and transitions belonging to the classes *Vertex* and *Transition*.

By applying γ to the SCP state machines metamodel, classes are mapped into ASM domains, generalization relationships are mapped into subset domain relations, and class attributes and associations are mapped into ASM functions suitable defined on the domains corresponding to the related classes. For example, the *State* class (see Fig. 4) is mapped in a subdomain of *Vertex*. Predicates *isSimple*, *isComposite*, and *isSubmachine* are defined on the domain *State* to distinguish among UML simple states, (sequential) composite states, and submachine states. In particular, simple states are of the form *state(name, container, incoming, outgoing, entry, exit, doActivity)* where the parameter *name* is the name of the state, *container* denotes the region containing the state, *incoming/outgoing* specify the transitions entering/departing from the state, *entry/exit* denote actions that are performed as soon as the state is entered/exited, *doActivity* denotes the internal behavior (if any) that must be executed as long as the state is active. All these parameters induced from the associations of the *State* class (or from the super classes *Vertex* and *NamedElement*) are encoded in terms of ASM functions according to the γ mapping rules in [11].

Stereotypes are mapped into domains, and their corresponding tags are mapped into functions, as well. OCL constraints of the SystemC UML profile, not reported here, state some restrictions on the stereotypes. This implies some constraints on the resulting

⁶ The ASMETA toolset <http://asmeta.sf.net/>

ASM model. For example, the `wait` stereotype is mapped into a simple state that has no *entry*, *exit*, and *doActivity* behavior⁷. The outgoing transitions of a *wait* state are dynamic resuming transitions of form $trans(container, source, target, trigger)$ where *container* denotes the region that contains the transition, *source/target* provide the source/target vertices of the transition, *trigger* is the label denoting the events (a time event or a signal event or an OR-list of signal events) which may enable the transition to fire. In case of an AND-transition – i.e. the dynamic resuming transition is stereotyped with `and` and labeled with a list of signal events –, the transition has AND-semantics: it may fire when all the events in the list have been notified, not necessarily all in the same delta-cycle⁸ or at the same time. To manage the history of the event occurrences of an AND-transition (see *Transition Selection rule* in Sect. 4.2), the controlled function $andHist(t, trans)$ is therefore introduced in the ASM signature and returns the list of events of an AND-transition t which have already been notified. Moreover, we distinguish wait-states using the predicate $isWait$ on *State*, and AND-transitions using the predicate $isAnd$ on *Transition*.

Control flow: Further signature elements not directly induced from the metamodel are added to represent the nesting structure of a state machine and its control flow. Suitable functions to encode and *navigate* nested states, like the functions $Up/DownChain(s_1, s_2)$ denoting ascending/descending sequences of nested states between states s_1 and s_2 , are defined similarly to the ones used for the same scope in the original ASM model in [11]. These functions can be formulated by composition of ASM functions derived from the metamodel.

In the sequel, elements of the domain `Sc_thread` are referred to as *threads*. A thread t moves through a SCP state machine diagram, $baseStateMachine(t)$, executing what is required for its *currently active state*. As effect of calling an operation op (that is not a SC process), during its lifetime a thread can temporarily moves from its base state machine to the state machine $method(op)$ associated to the invoked operation⁹, and then come back after completing the execution of the operation behavior. The state machine currently executed by t is given by

$$currStateMachine : Sc_thread \rightarrow StateMachine$$

initially set to $baseStateMachine(t)$, while the calling state machine is provided by

$$callMachine : Sc_thread \times StateMachine \rightarrow StateMachine$$

⁷ Some other OCL constraints state that concurrent (or orthogonal) composite states (i.e. composite states with more than one region) and other pseudostates (like `deepHistory`, `shallowHistory`, `entryPoint`, `join`, `fork`, and `exitPoint`) of the UML2 are not allowed in the SystemC UML profile. Moreover, *internal* transitions and *deferred* events are also not allowed.

⁸ A delta cycle is a very small step of time within the simulation, which does not increase the user-visible time.

⁹ The mechanism for determining the method (behavior) to invoke as effect of an operation call is unspecified in the UML. In the SystemC UML profile [28], state machines designed similarly to the SCP state machines are associated to operations as behaviors to invoke when the operations are called. We assume, therefore, that threads temporarily execute such a kind of state machine diagram.

In the current state machine of a thread, a state becomes active when it is entered as result of some fired transition, and becomes inactive when it is exited. All the composite states that either directly or transitively contain the active state are also active. The *current configuration* of active states w.r.t. the running state machine is given by

$$\text{currState} : \text{Sc_thread} \times \text{StateMachine} \rightarrow \mathcal{P}(\text{Vertex})$$

The function *deepest* : $\text{Sc_thread} \times \text{StateMachine} \rightarrow \text{Vertex}$ yields the last (innermost) active state reached by a thread running its current state machine.

Event handling: In the UML, the semantics of event occurrence processing is based on the *run-to-completion* assumption [33]. Since the event delivering and dispatching mechanisms are open in the UML, here they are explicitly modeled according to the discrete – *absolute* and *integer-valued* – time model of SystemC [32].

First, time is represented by an increasing monotonic function T . The domain `EventOccurr` represents the observable event occurrences (or event notifications) resulting from the execution of the processes. A function *type* : $\text{EventOccurr} \rightarrow \text{Event}$ returns the event type of a particular event occurrence. Event occurrences are collected in the global set *pendingEvents* and they are ordered by their time component $\text{time}(e)$ with respect to the current simulation time T_c .

Second, each thread t is endowed with a queue, *eventQueue*(t), of event occurrences. One event is processed at a time by each thread. In the context of a thread execution, an event is dispatched when it is taken from the head of the event queue. At this point, the event is considered consumed and referred to as the current event.

Third, as delivering (or resuming) mechanism, it is assumed that the threads' event queues are also updated by a *scheduler* modeled as a separate agent. This special agent has the responsibility to place the events, upon their occurrence, into the queues of the processes that are sensitive to them. Threads' event queues are therefore shared functions. The behavior of the scheduler agent is not formalized here (though we are recalling here the ASM functions adopted for the interaction with the scheduler). For an in depth description, we refer the reader to [20], where an ASM formalization of the SystemC 2.0 scheduler is given, and to its implementation in AsmetaL available at [16]. According to the scheduler formalization in [20], a shared function *status*(t) ranging over the enumeration $\{\text{READY}, \text{EXECUTING}, \text{SUSPENDED}\}$ is used to manage a thread life cycle. A thread is selected for execution by the scheduler, one after the other, from the set of *ready* processes. The set of all processes sensitive to an event type e is given by the function *processes*(e) : $\text{Event} \rightarrow \text{Sc_thread}$. Upon an event occurrence, the scheduler examines the process list of the event type to determine the processes (threads) to which deliver the event occurrence and turn them ready in case they were waiting for that event. An event occurrence can be explicitly required to be immediate in the current delta cycle, or for future time cycles.

In the SystemC UML profile, events can essentially be signal events (the `SignalEvent` class in the UML metamodel) or time events (the `TimeEvent` class). Signal events represent the receipt of asynchronous *sc_event* signals (as stereotyped), and are generated as a result of some *notify* actions (stereotyped `SendSignalAction`) executed by other processes (other threads or methods processes) either within the context module (or a hierarchical channel) or in the surrounding environment. Time events are

timeouts caused by the expiration of a time deadline always relative to the time of entry of the thread into a wait-state. *Completion events* (which originate from UML rather than SystemC) are directly handled by threads.

Finally, a function *dispatched(t)* yields for a thread *t* the head element of the thread's event queue to indicate the dequeued event to be processed¹⁰. At any moment, for a thread *t* the only transition *trans* that is eligible to fire when an event *e* occurs is the one departing from the deepest active state of *t*, with an associated guard (if any) evaluating to true ($eval(g,trans) = true$), and with *e* triggering *trans*. This is expressed by the following function:

$$enabled(e, t) = \begin{cases} trans & \text{if } triggering(trans, e, t) \\ undef & \text{otherwise} \end{cases}$$

where $triggering(trans, e, t)$ is a derived predicate defined as follows:

$$triggering(trans, e, t) \equiv source(trans) = deepest(t, currStateMachine(t)) \ \& \ eval(guard(trans), trans) \ \& \ \bigvee_i p_i(trans, e, t)$$

Each $p_i(trans, e, t)$ formalizes a different case of the semantics of dynamic resuming transitions (ranging, see Fig. 3, over timeout, events, and AND/OR lists of events), static resuming transitions, and completion transitions. In case *trans*, for example, is an AND-dynamic resuming transition, $p_i(trans, e, t)$ holds if and only if *e* is in $trigger(trans)$ and all the other events in $trigger(trans)$ have already been notified to *t*

$$isAnd(trans) \ \& \ (\exists e' \in trigger(t) : event(e') = type(e)) \ \& \ (\forall e'' \in trigger(trans), event(e'') \neq type(e) : event(e'') \in andHist(t, trans))$$

4.2 ASM Transition System

This section describes the ASM semantics of the *run to completion step* of the SystemC thread state machines. The behavior of a thread consists of the two rules *TransitionSelection* and *GenerateCompletionEvent* for simultaneously (i) selecting the machine transition to be executed next, and (ii) generate completion events. The next paragraph defines the exact meaning of “executing a state machine” by a parameterized macro rule *stateMachineExecution*.

In the *TransitionSelection* rule, a check is done in parallel to the machine execution for treating *dynamic resuming* transitions with an AND-semantics (the OR-semantics is the default): an AND-transition may fire when all the labeling triggers have been effectively triggered – not necessarily all in the same delta-cycle or at the same time –, and in this case the history of the occurrences of its AND-list of events is reset to empty. If a dispatched event does not trigger any transition in the current state of a thread, it is lost unless (the **else** branch) it must be collected in the history of an AND-transition.

rule *TransitionSelection(t)* =
if $status(t) = EXECUTING$
then let $e = dispatched(t)$, $trans = enabled(e, t)$, $s = deepest(t, currStateMachine(t))$

¹⁰ It should be noted that at this point the ASM model differs from the one in [11] since the mechanism here for selecting the event to consume is deterministic.

```

in if  $trans \neq undef$ 
  then par
     $stateMachineExecution(t,trans)$ 
    if  $isAnd(trans)$  then  $andHist(t,trans) := []$ 
  else if  $isAndWait(s,e)$  then  $andHist(t,trans) := add(e,andHist(t,trans))$ 

```

where $isAndWait(s,e) \equiv isWait(s) \ \& \ (\exists trans \in outgoing(s) \mid isAnd(trans) \ \& \ e \in trigger(trans))$

Completion events are generated by a thread when an active state satisfies the *completion condition* [33]. This is formalized similarly as in the ASM model in [1] by a rule *GenerateCompletionEvent* parameterized with t , with the only difference that the completion event generated is added to the head of the thread event queue.

The rule macros: This paragraph reports only a very small subset of the rule macros used in the top level rules. The subrule *stateMachineExecution* is described first. It formalizes the run-to-completion semantics which consists into sequentially executing: (a) the exit actions of the source state and of any enclosing state up to, but not including, the least common ancestor *LCA* (i.e. the innermost composite state that encloses both the source and the target state), innermost first (see macro *exitState*); (b) the action on the transition; (c) the entry actions of any enclosing state up to, but not including, the least common ancestor, outermost first (see macro *enterState*); finally, (d) the “nature” of the target state is checked and the corresponding operations are performed.

macro rule $stateMachineExecution(t,trans) =$

seq

```

   $exitState(source(trans),ToS,t)$ 
   $execute(effect(trans),t)$ 
   $enterState(FromS,target(trans),t)$ 
  case  $target(trans)$ 
     $isSimple: enterSimpleState(target(trans),t)$ 
     $isComposite, isSubmachine : enterCompositeState(target(trans),t)$ 
     $isWait: enterWaitState(target(trans),t)$ 
     $isStatic\_wait, isDont\_initialize: enterStaticWaitState(target(trans),t)$ 
     $isFinal, isIf, isEndif, isEndswitch: enterNextState(target(trans),t)$ 
     $isSwitch: enterSwitchState(target(trans))$ 
     $isWhile, isDowhile, isFor: enterLoopState(target(trans),t)$ 
     $isReturn: enterReturnState(target(trans),t)$ 
     $isBreak: enterBreakState(target(trans),t)$ 
     $isContinue: enterContinueState(target(trans),t)$ 
     $isExit: enterExitState(target(trans),t)$ 

```

endcase

where *ToS* is the direct sub-state of the *LCA* in the nested state chain from $source(trans)$ to *LCA*; while, *FromS* is the direct sub-state of the *LCA* in the nested state chain from *LCA* to $target(trans)$.

Macros *exitState* and *enterState* are formalized similarly as in [1]. The exits from nested states should be performed in an order that respects the hierarchical structure of the machine. Starting from the deepest state up to, but excluding, the source/target least common ancestor state, innermost first, a thread sequentially (i) executes the exit

actions (if any)¹¹, and (ii) removes those states from the thread's current state and, when states are exited, their enclosed final state (if any).

```

macro rule exitState( $s, v, t$ ) =
loop through  $S \in UpChain(s, v)$ 
seq
  if  $\neg isPseudoState(S)$  then execute(exit( $S, t$ ))
  currState( $t, currStateMachine(t)$ ) := remove( $S, currState(t, currStateMachine(t))$ )

```

Similarly, for entering nested states, any state enclosing the target one up to, but excluding, the least common ancestor will be entered in sequence, outermost first. Entering a state means that (a) the state is activated, i.e. inserted in $currState(t, currStateMachine(t))$, (b) its entry action (if any) is performed, and (c) the state internal activity (if any) is started.

```

macro rule enterState( $s, v, t$ ) =
loop through  $S \in DownChain(s, v)$ 
seq
  enterNextState( $S, t$ )
  if  $\neg isPseudoState(S)$ 
  then seq
    execute(entry( $S, t$ ))
    execute(doActivity( $S, t$ ))

```

where $enterNextState(s, t) \equiv currState(t, currStateMachine(t)) := insert(s, currState(t, currStateMachine(t)))$.

Macros for entering vertices depending on their specific nature are completely defined in [10]. We here report only that for entering a wait-state. When the target state of the triggered transition is a wait state, the thread is suspended as follows. The thread inserts itself in the process list of all events e appearing as triggers in the outgoing transitions of the wait-state, and changes its status to *suspended*. The thread will be turned *ready* by the scheduler when an event that the thread is waiting for will be notified. In case of timeout, i.e. an outgoing transition with a time event, the thread creates an event occurrence with time $timeout + T_c$ and adds it to the set of pending events.

```

macro rule enterWaitState( $s, t$ ) =
  if  $\exists e' \in trigger(outgoing(s)) : isTimeEvent(e')$ 
  then extend EventOccurr with  $e$ 
    time( $e$ ) :=  $T_c + eval(when(event(e')))$ 
    type( $e$ ) := event( $e'$ )
    pendingEvents := add(pendingEvents,  $e$ )
  forall  $e \in trigger(outgoing(s))$  do processes(event( $e$ )) := add(processes(event( $e$ )),  $t$ )
  status( $t$ ) := SUSPENDED

```

where $when(e)$ for a time event e is an expression specifying a relative instant in time.

4.3 ASM Initial State

The initial state is the result of the mapping $\iota(SC-UML)$, defined by the HOT ι applied to a terminal SystemC-UML model. It provides the initial values of domains and of

¹¹ Note there is no reason to stop the internal ongoing activities (if any) before exit, since the only outgoing transitions from a non stereotyped state are completion transitions.

(dynamic) controlled functions of the ASM signature necessary to execute each process state machine (like the `stimgen` thread machine shown in Fig. 2) appearing in the terminal model. For the lack of space, the result of this final step is not reported here.

5 Related Work

There are different ways currently used to specify the semantics of metamodel-based languages, and therefore of UML profiles. They mainly fall into the following categories.

(I) *Using natural languages* to describe language semantics informally.

(II) *Using the OCL* [22] and its extensions, see [4,8,7] to name a few, to specify static semantics through invariants and behavior through pre/post-conditions on operations; however, being side-effect free, the OCL does not allow the change of a model state, though it allows describing it.

(III) *Weaving behavior*. Recent works like Kermeta [19], xOCL (eXecutable OCL) [34], approaches in [29,31], to name a few, propose ways of providing executability natively into metamodeling frameworks. A minimal set of executable actions is usually defined to describe (create/delete object, slot update, conditional operators, loops, local variables declarations, call expressions, etc.) behavioral semantics of metamodels by attaching behavior to classes operations. Some approaches use imperative or objected-oriented (sub)languages, other use abstract pseudo-code. Furthermore, [9] provides an executable subset of standard UML (the Foundational UML Subset) to define the semantics of modeling languages such as the standard UML or extensions.

Although, these action languages aim to be pragmatic, extensible and modifiable, some of them suffer from the same shortcomings of traditional programming languages; indeed, a behavioral description written in one of such action languages has the same complexity of one (a program) written in a conventional programming language. Moreover, not all action semantics proposals are powerful enough to specify the model of computation (MoC) underlying the language being modeled and to provide such a specification with a clear formal semantics. As shown in [11,23], when we illustrate a similar technique, the *weaving technique*, of our ASM-based framework, the ASMs formalism itself can be also intended as an action language but with a concise and powerful set of action schema provided by different ASM rule constructors.

(IV) *Translational semantics*, consisting in defining a mapping from the language metamodel to the abstract syntax of another language that is supposed to be formally defined. In [3], metamodels are *anchored* to formal models of computation built upon AsmL, a language to encode ASM models. In [6], the semantics of the AMMA/ATL transformation language is specified in XASM, an open source ASM dialect. Similar approaches based on this *translational* technique are UML-B [30] using the Event-B formal method, those adopting Object-Z like [17,5], etc. Some previous works using ASMs to provide an executable and rigorous semantics of UML graphical sub-languages (statecharts, activity diagrams, etc.) [21,11,4,8] fall in this category. These approaches can be intended as exemplifications of other translational techniques of the ASM semantic framework, as better described in [11]. The technique used here is also translational, but it is more general as it works at the “meta” level leading more automation, and therefore less user effort, and more reusable mappings and specifications.

(V) *Semantic domain modelling*, where a metamodel for the “semantic domain” – i.e. to express also concepts of the run-time execution environment – is defined, and then OCL rules are used to map elements of the language metamodel into elements of the semantic domain metamodel. This approach was used for the *CMOF Abstract Semantics* [18] and for the OCL [22]. We postponed as future step the evaluation of the effectiveness of the joint-use of this technique with the ASM formal method, as it requires a certain effort in modeling, at metamodel level, also the semantic domain.

6 Conclusions

This paper presents an executable formal semantics for the SCP state machines of the SystemC UML profile by exploiting the meta-hooking approach of the ASM-based semantic framework in [11]. Executability allows *semantics prototyping* to examine particular behavioral features of the profile and to check if the provided extensions of the UML metamodel are *conservative*, i.e. if their semantics does not contradict the UML semantics, and fix explicitly the *semantics variation points* intentionally left in the UML as leeway for the definition of domain-specific UML profiles. The comparison in [10] between the ASM model for the UML state machines in [1] and the ASM model for the SCPs described here shows that SCPs are effectively a conservative extension.

Formal ASM models obtained from graphical SystemC-UML models can potentially drive practical SoC model analysis like simulation, architecture evaluation and design exploration [12].

References

1. Börger, E., Cavarra, A., Riccobene, E.: Modeling the Dynamics of UML State Machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 223–241. Springer, Heidelberg (2000)
2. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
3. Chen, K., Sztipanovits, J., Neema, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: ACM Conf. on Embedded Software, pp. 35–43 (2005)
4. Combemale, B., et al.: Towards a Formal Verification of Process Models’s properties - SimplePDL and TOCL case study. In: 9th Int. Conf. on Enterprise Information Systems (2007)
5. Mostafa, A.M., et al.: Toward a Formalization of UML2.0 Metamodel using Z Specifications. In: ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, vol. 1, pp. 694–701 (2007)
6. Di Ruscio, D., et al.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Tech. Rep. 06.02, LINA (2006)
7. Flake, S., Müller, W.: A UML Profile for Real-Time Constraints with the OCL. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 179–195. Springer, Heidelberg (2002)
8. Flake, S., Müller, W.: An ASM Definition of the Dynamic OCL 2.0 Semantics. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 226–240. Springer, Heidelberg (2004)
9. OMG. Semantics of a Foundational Subset for Executable UML Models, ptc/2008-11-03
10. Gargantini, A., Riccobene, E., Scandurra, P.: A precise and executable semantics of the SystemC UML profile by the meta-hooking approach. DTI T.R. 110, Univ. of Milan (2008)

11. Gargantini, A., Riccobene, E., Scandurra, P.: A semantic framework for metamodel-based languages. *J. of Automated Software Engineering* 16(3-4) (2009)
12. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven design and ASM-based validation of embedded systems. In: *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, July 2009, pp. 24–54 (2009)
13. Harel, D., Rumpe, B.: Meaningful modeling: What’s the semantics of “semantics”? *IEEE Computer* 37(10), 64–72 (2004)
14. Jürjens, J.: A UML statecharts semantics with message-passing. In: *Proc. of the 2002 ACM symposium on Applied computing*, pp. 1009–1013. ACM Press, New York (2002)
15. Kurtev, I., et al.: Model-based DSL frameworks. In: *21st ACM SIGPLAN conf. on Object-oriented programming systems, languages, and applications*, pp. 602–616 (2006)
16. https://asmeta.svn.sf.net/svnroot/asmeta/asm_examples/
17. Miao, H., Liu, L., Li, L.: Formalizing UML Models with Object-Z. In: George, C.W., Miao, H. (eds.) *ICFEM 2002*. LNCS, vol. 2495, pp. 523–534. Springer, Heidelberg (2002)
18. OMG. Meta Object Facility (MOF) 2.0, formal/2006-01-01 (2006)
19. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-Languages. In: Briand, L.C., Williams, C. (eds.) *MoDELS 2005*. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
20. Müller, W., Ruf, J., Rosenstiel, W.: An ASM based SystemC simulation semantics. *System C: Methodologies and Applications*, 97–126 (2003)
21. Ober, I.: More meaningful UML Models. In: *TOOLS - 37 Pacific 2000*. IEEE, Los Alamitos (2000)
22. OMG. Object Constraint Language (OCL), 2.0 formal/2006-05-01 (2006)
23. Riccobene, E., Scandurra, P.: Weaving executability into UML class models at PIM level. In: *Proc. of European Workshop on Behaviour Modelling in Model Driven Architecture (BM-MDA 2009)*, Enschede, The Netherlands, June 23, vol. 379, pp. 10–27. ACM, New York (2009)
24. Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A.: An Enhanced SystemC UML Profile for Modeling at Transaction-Level. In: Villar, E. (ed.) *Embedded Systems Specification and Design Languages* (2008)
25. Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A.: A SoC Design Methodology Based on a UML 2.0 Profile for SystemC. In: *Proc. of Design Automation and Test in Europe*, pp. 704–709. IEEE Computer Society, Los Alamitos (2005)
26. Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A.: A model-driven design environment for embedded systems. In: *Proc. of the 43rd Design Automation Conference*, pp. 915–918. ACM Press, New York (2006)
27. Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A.: A Model-driven co-design flow for Embedded Systems. In: *Advances in Design and Specification Languages for Embedded Systems, Best of FDL 2006* (2007)
28. Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A., Lavazza, L., Mantellini, L.: SystemC/C-based model-driven design for embedded systems. *ACM TECS* 8(4) (2009)
29. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) *ECMDA-FA*. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
30. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.* 15(1), 92–122 (2006)
31. Soden, M., Eichler, H.: Towards a model execution framework for Eclipse. In: *Proc. of the 1st Workshop on Behavior Modeling in Model-Driven Architecture*. ACM, New York (2009)
32. *SystemC Language Reference Manual*. IEEE Std 1666 (2006)
33. OMG. The Unified Modeling Language (UML), v2.2. (2009), <http://www.uml.org>
34. The Xactium XMF Mosaic (2007), <http://www.xactium.com/>

Specifying Self-configurable Component-Based Systems with FracToy

Alban Tiberghien, Philippe Merle, and Lionel Seinturier

INRIA Lille - Nord Europe
University of Lille 1 - LIFL CNRS UMR 8022
Villeneuve d'Ascq, France
`firstname.lastname@inria.fr`

Abstract. One of the key research challenges in autonomic computing is to define rigorous mathematical models for specifying, analyzing, and verifying high-level self-* policies. This paper presents the FracToy formal methodology to specify self-configurable component-based systems, and particularly both their component-based architectural description and their self-configuration policies. This rigorous methodology is based on the first-order relational logic, and is implemented with the Alloy formal specification language. The paper presents the different steps of the FracToy methodology and illustrates them on a self-configurable component-based example.

Keywords: Alloy, Autonomic Computing, Component-based Systems, Formal Specification Self-Configuration.

1 Introduction

Autonomic computing gathers *systems that can manage themselves given high-level objectives from administrators* [12]. The idea is to design software which can provide efficient and continuous services to users without any human intervention. Self-configurability is a key property of any autonomous system, and means the capability of such a system to configure itself according to high-level policies automatically. For instance, software components [16] and connectors can be added or removed to/from a running software system according to evolutions of runtime conditions. These dynamic modifications of running software architectures can be described by high-level self-configuration policies. Here, one of the key research challenges is to define rigorous mathematical models for specifying, analyzing, and verifying such autonomous systems. Such a model must allow to detect errors and inconsistencies of high-level policies early at design time instead of during execution of targeted autonomous systems.

To tackle this problem, this paper presents the FracToy formal methodology to specify, analyse, and verify self-configurable component-based systems. This rigorous methodology is based on the first-order relational logic, and is implemented with the Alloy formal specification language [10]. This methodology is iterative and divided into two main steps. The first step consists in

specifying the component model used to build applications. This step is itself divided into three sub-steps. This first sub-step consists in defining the formal syntax of the component model, both its core concepts and the relations between these concepts. Secondly, this component model is constrained in order to define its static semantics, i.e., the set of constraints that any application must satisfy. Thirdly, it is necessary to specify the dynamic semantics of the component model, i.e., the set of operations allowing to update the architecture of running applications. Here, this dynamic semantics must be defined in a way allowing self-configurability of applications. Then, the second step of the FracToy methodology is to specify self-configurable component-based applications. Their components are defined by extending the core concepts of the component model and their self-configuration policies are defined as first-order logic constraints. Furthermore, the FracToy methodology allows to highlight and verify properties like the consistency of both the static semantics and self-configuration policies, and the commutativity of dynamic operations.

This paper is organized as follows. Section 2 presents the FracToy methodology and its different steps. Section 3 illustrates the methodology on a self-configurable component-based “Room” example. Section 4 discusses related works. Finally, Section 5 concludes and draws perspectives of FracToy.

2 The FracToy Framework

FracToy is a framework that introduces a methodology, based on Alloy [9], for the formal description of self-configurable component-based systems.

2.1 Alloy in a Nutshell

The Alloy formal specification language fits with the first-order relational logic [10]. The manipulated concepts are *sets* (Alloy signatures) that can be brought together using *relations* (Alloy signature fields). Alloy models are described with these two concepts and are constrained using facts or predicates. A fact is an expression that the whole model must always satisfy. A predicate is a parametrizable constraint which is applied only when invoked. As facts, predicates can be applied on the whole systems but also just on a specific signature. Furthermore, the language provides a model analyser. The Alloy Analyser can be used as a model finder (invoked with the Alloy `run` command) that instantiate all the models that satisfy the Alloy specification. It can also be used as a counter-example finder (invoked with the Alloy `check` command) in order to counter-example models that don't satisfy assertion (defined with the Alloy `assert` keyword). The combined use of the model finder and of counter-example finder allows fast iterative debugging, during the design process.

2.2 The FracToy Methodology

The FracToy methodology proposes a use of Alloy for specifying, verifying and analysing self-configurable component-based systems. This rigorous and iterative methodology is divided into the two following steps and illustrated by Figure 1.

1. **Specification of the component model** composed of three sub-steps.

(a) **The formal syntax:** This step consists in defining each core concept of the component model and the relations between these concepts. Each concept is an Alloy abstract signature. Alloy signature fields define how and what concepts can be bound to a given concept. At this step, the model is not constrained but basic restrictions are nevertheless specified using the `one`, `lone` and `set` keywords in order to define the cardinality of the relations.

(b) **The formal static semantics:** The static semantics of the component model is the set of constraints restricting the model. These constraints can be facts establishing what is possible to model with the component model. They can also be predicates in order to define finer-grain constraints that just concern certain concepts.

Consistency checking: Once the static semantics is specified, it is possible to run a consistency test in order to verify that the constrained component model is instantiable. If tests don't pass, a correction/refinement loop can be performed on this step or on the previous step.

(c) **The formal dynamic semantics:** Operations that dynamically update the running system must be specified in a way to preserve the self-configurable nature of the architecture. It is important to clearly identify the different states that the systems can reach. By fixing the pre-conditions and post-conditions, these operations define what is preserved during the changes of state of the system.

Properties checking: These checks ensure that the dynamic of the system is well specified. For example, they ensure that add/remove operations are commutative. Indeed, all operations of the system have its inverse operations and the couple of operations must be commutative in order to have the certainty that it is possible to roll back in a stable state after applying an operation on the system.

2. **Specification of the self-configurable system:** Each component of the self-configurable architecture is a signature extending a concept of the component model. In the context of component-based architecture specification, the declaration part of the signature is dedicated to the declaration of services, references and/or sub-components. The Alloy `one`, `lone` and `set` keywords are used to specify the cardinality of these relations. The constraint part of the signature is dedicated to the definition of the assembly. In this part, constraints are used to map the previously declared fields to the concepts of the component model. Additional constraints can be added in order to limit the use of components in the case of the component model is not enough restrictive.

Self-configuration policies definition: Self-configuration policies are directly defined in the constraint part of the component signature. Indeed, in our approach, self-configuration is managed by components themselves.

Consistency/Properties checking: Here, it is possible to check the consistency on the full specified architecture and to verify that the self-configuration policies are efficiently applied and conform to the requirement.

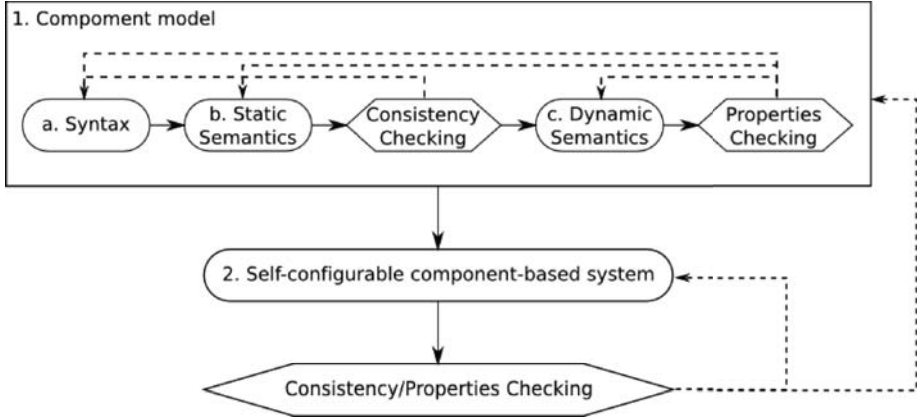


Fig. 1. The FracToy methodology

3 FracToy in Action

Following the methodology presented in Section 2, this section provides the specification of a self-configurable component-based system, the “Room” use case, presented in Section 3.1. First, the component model is specified in Section 3.2 and, then, the “Room” self-configurable architecture is specified in Section 3.3. Verification and analysis are performed in Section 3.4.

3.1 The “Room” Scenario

The scenario describes the case where a mobile user enters a room and wants to keep in touch with news and services provided by the room. The user’s mobile device can receive news from the room and once s/he has obtained the expected information, s/he can visualise them on a screen or print them, according to the features available on her/his mobile device.

More precisely, there is a news provider that broadcasts news in the room. The room provides two kinds of output devices: screen and printer. The room is aware of the presence of all mobile devices. When a new mobile device (e.g. PDA, smartphones, etc.) enters the room, it is automatically connected to the news provider and to the screen and/or the printer devices according to the type of output devices it supports. For example, a PDA can print and display whereas a smartphone can only print because of power and energy restrictions. Finally, several mobile devices can be in the same room at the same time.

A component-based architecture description: The *Room* scenario can be reified as a self-configurable component-based systems. The room and all devices are components. Each component has a variable number of input and output ports (communication points) respectively called services and references. The *NewsProvider* component has no service and its number of references (of type *News*) is not statically defined and can evolve according to the number of *MobileDevice* components contained in the *Room* component. Each *MobileDevice* component has one *News*-typed service and the number of references and their types (either *DisplayableNews* or *PrintableNews*) are specific to each type of *mobileDevice*. According to the informal definition, the *PDA* component has a reference of type *DisplayableNews* and a reference of type *PrintableNews* whereas the *Smartphone* component has only a reference of type *PrintableNews*. Self-configuration is performed when a *MobileDevice* component is added to the *Room* component. In this case, all bindings are automatically established between the *MobileDevice* components and other components.

3.2 Specification of the Component Model

Informal specification: Our use case is not based on an existing component model, the presented component model remains consistent with the Szyperski component definition given in [16] in the way that “a component is a unit of composition with contractually specified interfaces and context dependencies only”. The elementary entity of our model is *Component*. As this component model is hierarchical, a component can be either *Composite*, i.e. a component that can contain sub-components, or *Primitive*, i.e. a component implemented in a programming language. *Port* represents typed communication access points to a component. A port is either a *Service* (providing functionality) or a *Reference* (requiring functionality). Finally, it is possible to bind a reference to a service in order to explain communication channels between components. As our work takes place in a context of dynamic environments, this component model has to deal with this concern. That is why it is important to notice that when we use the term “component” or “port” it must be understood “a state of a component” or “a state of a port”. Indeed an instance of a component models a certain state of the component. Each state is identified by an *Id*. Components have a *cid* and ports have a *pid*. If two different component instances have the same id, that means that we are semantically dealing with the same component but in different states.

Figure 2 represents the diagram of the key-concepts of the component model and their relations.

The formal syntax: We first declare the signature named *Component* (line 3). The fields *services* (line 5) and *references* (line 6) allow to respectively put in relation a component to its set of *Services* and *References*. Each component has a field *cid* (line 4) which represents the identity of the component (line 1). Two signatures specialize (“extends” in Alloy) the concept of component. The *Primitive* signature (line 8) just allows to directly manipulate this concept and to

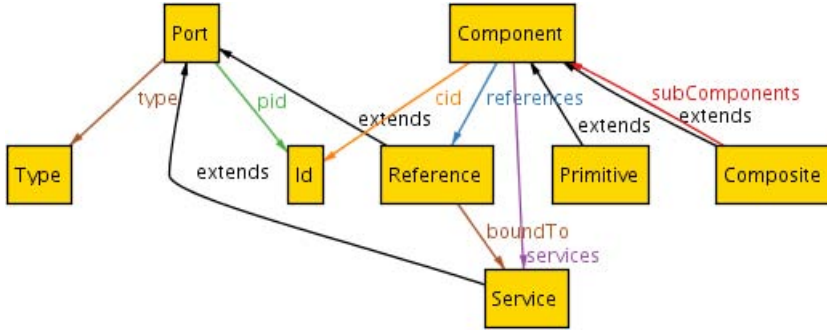


Fig. 2. Diagram of the component model generated by the Alloy Analyser

have a type for this kind of component. It is the same principle for the *Composite* signature (line 11) whereas it is possible, with this set, to associate (line 12) a component to other components (semantically its sub-components).

```

1  sig Id {}
2
3  abstract sig Component {
4    cid : one Id,
5    services : set Service,
6    references : set Reference
7  }
8  abstract sig Primitive extends Component{}
9
10
11 abstract sig Composite extends Component{
12   subComponents : set Component
13 }

```

In this component model, the *Port* signature (line 16) is a typed (line 18) communication access point of a component. In the same way as component, ports have an identity *pid* (line 17). *Service* (line 20) and *Reference* signatures (line 22) correspond to the functionality that a component provides and requires, respectively. The *boundTo* field (line 23) allows to bind a reference to a service. A reference can be bound to zero or one service and as a consequence a reference can exist even if it is not bound (specified with the Alloy *lone* keyword).

```

14 abstract sig Type {}
15
16 abstract sig Port {
17   pid : one Id,
18   type : one Type
19 }
20 sig Service extends Port {}
21
22 sig Reference extends Port {
23   boundTo : lone Service
24 }

```

The formal static semantics: In addition of the formal syntax, the static semantics of the component model is defined as a set of constraints in order to avoid certain use cases. The fact *AllPortsAffectedToOneComponent* (line 25) forces that all ports of a system are owned by one and only one component i.e. can be shared by two distinct component instances only if they have the same identity. The fact *AllBindingsInTheSameComposite* (line 32) ensures that all references of a sub-component are bound to a service of a sub-component of the same composite. The fact *NoBindingBetweenUncompatibleTypes* (line 39) just

forbids that a binding is established if the types of the reference and the service are not the same. The fact *CompositeNotContainItself* (line 42) avoids that a composite contains itself in its sub-components. The *bind* predicate (line 45) declares a binding between a reference and a service. This statement chooses a reference in the set of references and binds it to the service (line 46).

```

25 fact AllPortsAffectedToOneComponent {
26   all p : Port {
27     all c, c' : Component {
28       (p in c.(services+references) and p in c'.(services+references)) implies c.cid = c'.cid
29     }
30   }
31 }
32 fact AllBindingsInTheSameComposite{
33   all c : Composite {
34     all ref : c.subComponents.references {
35       ref.boundTo in c.subComponents.services
36     }
37   }
38 }
39 fact NoBindingBetweenUncompatibleTypes {
40   all r : Reference, s : Service | r.type != s.type implies r.boundTo != s
41 }
42 fact CompositeNotContainItself {
43   all c : Composite | c not in c.subComponents
44 }
45 pred Composite.bind[references : set Reference, service : one Service] {
46   one ref : references {
47     ref.boundTo = service
48   }
49 }

```

A test of consistency can be performed on the formal specification of this component model. This test consists in asking to the analyser to instantiate a model in an arbitrary (but coherent) scope. Here, *ComponentModelConsistency* test can be run, i.e, the analyser is able to instantiate a model that satisfy all the defined constraints. In other words, this core of concepts is consistent and can be a sure basis for more complicated architectures.

ComponentModelConsistency : **run** {} **for** 20

The formal dynamic semantics: The last part of the specification of the component model is its dynamic semantics. In the context of our example, the dynamic semantics of the addition and the removal of a component in a composite has been formally specified. The two predicates *addComponent* (line 50) and *removeComponent* (line 57) are semantically commutative and are built following the same logic. In order to modelize the dynamicity of an addition (removal resp.), a predicate formalizes the change of state due to the operation execution. The two first parameters of these predicates, *c1* and *sc1*, symbolize the state of the system before the operation execution, and the two last parameters *c2* and *sc2*, symbolize the state of the system after the operation execution. A semantics for these actions is to formalize that the resulting state of a component addition (removal resp.) is the start state plus (minus resp.) the component to add (remove resp.) and there is nothing more nothing less element in the architecture. This semantics is too strong in our case of self-configurable component-based

system. Indeed, according to our *Room* example, when a *MobileDevice* component is added in the *Room* composite, the self-configuration policies are applied and as a consequence bindings are created between components and, thus, there is more than the new *MobileDevice* component in the *Room* composite. That is why it is important to notice that these operations don't ensure the strict equality of the system state (modulo the addition/removal of the component) but are based on the notion of state equivalence. Indeed both operations ensure the preservation of at least all that were present in the initial state of the system but it is not forbidden that the final state contains more elements.

Based on this logic, the *addComponent* predicate constrains the component *sc1* not to be in the sub-components of the composite *c1* (line 51). The final composite *c2* is constrained to be equivalent to the initial composite *c1* (line 52) and the final component *sc2* to be equivalent to the initial added component *sc1* (line 53). Finally, the component *sc2* must be in the sub-components of the composite *c2* (line 54). It is exactly the opposite for the *removeComponent* predicate.

```

50  pred addComponent[c1 : Composite, sc1 : Component, c2 : Composite, sc2 : Component] {
51    sc1 not in c1.subComponents
52    compositeEquiv[c1, c2]
53    componentEquiv[sc1, sc2]
54    sc2 in c2.subComponents
55  }
56
57  pred removeComponent[c1 : Composite, sc1 : Component, c2 : Composite, sc2 : Component] {
58    sc1 in c1.subComponents
59    compositeEquiv[c2, c1]
60    componentEquiv[sc2, sc1]
61    sc2 not in c2.subComponents
62  }

```

The relation of equivalence used for the formalization of the addition and the removal of a component in a composite is specified through the three following predicates. Two components are equivalent (line 63) if they have the same identity (line 64) and if their services and references are equivalent (lines 65 and 66). Two port sets are equivalent (line 68) if all ports of the first set (line 69) have an equivalent port in the second set (line 70). Two ports are equivalent if they have the same identity (line 71) and the same type (line 72). Finally, two composites are equivalent (line 76) if they are equivalent components (line 77) and if all sub-components of the first composite have its equivalent in the second composite (lines 78-80). This formalization allows to support self-configuration policies as shown in Section [3.3](#).

```

63  pred componentEquiv(c1 : Component, c2 : Component) {
64    c1.cid = c2.cid
65    portEquiv[c1.references, c2.references]
66    portEquiv[c1.services, c2.services]
67  }
68  pred portEquiv(portSet1 : set Port, portSet2 : set Port) {
69    all p1 : portSet1 {
70      one p2 : portSet2 {
71        p1.pid = p2.pid
72        p1.type = p2.type
73      }
74    }
75  }

```

```

76 pred compositeEquiv(c1 : Composite , c2 : Composite){
77   componentEquiv[c1, c2]
78   all sc1 : c1.subComponents {
79     one sc2 : c2.subComponents {
80       componentEquiv[sc1,sc2]
81     }
82   }
83 }

```

An important property can be checked thanks to the Alloy analyser on the dynamic addition and removal of a component in a composite. Semantically the *addComponent* and *removeComponent* are two commutable operations. The *AddRemoveCommutable* assertion checks that adding a component in a composite then removing it keep the system in the same state. In other words, this assertion tests that the two predicates are commutable.

```

assert AddRemoveCommutable {
  all c1, c2 : Composite, sc1, sc2 : Component {
    addComponent[c1, sc1, c2, sc2] implies removeComponent[c2, sc2, c1, sc1]
  }
}
check AddRemoveCommutable for 10 expect 0

```

3.3 Specification of the Self-configurable *Room* System

In a general way, the “Room” example is specified by extending the component model. Three singleton types, i.e. , *News*, *DisplayableNews*, and *PrintableNews*, are first defined. They respectively correspond to the type of each service and reference port (singletons are obtained thanks to the Alloy **one** keyword).

```

84 one sig News, DisplayableNews, PrintableNews extends Type {}

```

Specification of the primitive components: *NewsProviders* is a primitive component (line 85). It provides no service (line 88) but requires a set of references named *r* (line 86). All these references are of type *News* (line 89) and it can not require other references than *r* (line 90).

```

85 sig NewsProvider extends Primitive {
86   r : set Reference
87 } {
88   no services
89   r.type = News
90   references = r
91 }

```

Printer and *Screen* are two other primitive components (lines 92 and 100 resp.). Both require no reference (lines 95 and 103 resp.) but they provide a service named *s* (lines 93 and 101 resp.). This service is of type *PrintableNews* for the *Printer* primitive (line 96) and of type *DisplayableNews* for the *Screen* one (line 96). They can not provide other services than *s* (lines 97 and 105 resp.).

MobileDevice is an abstract primitive component (line 99) for modeling any mobile device.


```

92 sig Printer extends Primitive {           100 sig Screen extends Primitive {
93   s : one Service                         101   s : one Service
94 } {                                        102 } {
95   no references                           103   no references
96   s.type = PrintableNews                 104   s.type = DisplayableNews
97   services = s                           105   services = s
98 }                                        106 }
99 abstract sig MobileDevice extends Primitive {}

```

Specification of the *Room* composite: After having defined the different primitive components of the architecture, the *Room* composite can be specified (line 107). As this composite is autonomous, it doesn't declare neither services (line 113) nor references (line 114). It contains at least three primitives declare as a relation between the *Room* and the primitive sets (lines 108-110). Here the relation name represents the name of the sub-component. The Alloy **one** keyword means that there can be only one *NewsProvider*, one *Printer*, and one *Screen*. The *mobileDevices* field declares a pool of *MobileDevice*. Indeed, as the *Room* composite is open to different incoming/outcoming mobile devices, we have modelised this by the use of a set of *MobileDevice* (line 111). The constrain in line 111 specifies that these components are effectively declared as sub-component of the composite and that it can not have other kind of components in a *Room*.

In our methodology, the self-configuration policies are expressed as a constraint. These policies are declared in the signature of the composite that manages the self-configuration. Thus, the self-configuration policy of this use case specifies that, for all mobile devices contained in a room (line 118), all services of this mobile device (line 119) and of type *News* is bound from one reference of the *NewProvider* component (line 120). Regarding the mobile device references, there are two cases. If the reference is of type *DisplayableNews*, this reference is bound to the service provided by the *Screen* component (line 123). If the reference is of type *PrintableableNews*, this reference is bound to the service provided by the *Printer* component (line 124).

```

107 sig Room extends Composite {
108   newsProvider : one NewsProvider,
109   printer : one Printer,
110   screen : one Screen,
111   mobileDevices : set MobileDevice
112 } {
113   no services
114   no references
115   subComponents = newsProvider + printer + screen + mobileDevices
116
117   //SELF-CONFIGURATION POLICY
118   all md : mobileDevices {
119     all serv : md.@services {
120       serv.type = News implies bind[newsProvider.r, serv]
121     }
122     all ref : md.@references {
123       ref.type = DisplayableNews implies bind[ref, screen.s]
124       else ref.type = PrintableNews implies bind[ref, printer.s]
125     }
126   }
127 }

```

The specification of specific mobile devices: *PDA* and *SmartPhone* are both *MobileDevice* components. Both provide only one service *s* of type *News* (lines 129, 133, 136 and lines 140, 143, 146). The difference is done by the reference that these mobile devices require. Both require one reference of type *PrintableNews* (lines 130, 134 and lines 141, 144) but, in addition, the PDA requires one reference of type *DisplayableNews* (lines 131, 135).

```

128 sig PDA extends MobileDevice {           138 }
129   s : one Service,                       139 sig SmartPhone extends MobileDevice {
130   r1 : one Reference,                    140   s : one Service,
131   r2 : one Reference                    141   r : one Reference
132 } {                                       142 } {
133   s.type = News                          143   s.type = News
134   r1.type = PrintableNews                144   r.type = PrintableNews
135   r2.type = DisplayableNews              145   services = s
136   services = s                           146   references = r
137   references = r1 + r2                   147 }

```

The whole self-configuration specification is completed and a more realistic test of consistency can be performed. The *SelfConfigurableArchitectureConsistency* tries to instantiate a model conform to the “Room” use case when a PDA is present in the room.

```

SelfConfigurableArchitectureConsistency: run {
  one myRoom : Room, pda : PDA | pda in myRoom.mobileDevices
}
for exactly 1 Composite, exactly 4 Primitive, exactly 6 Port, exactly 3 Type, exactly 11 Id

```

3.4 Analysis of the *Room* Architecture

Static properties checking: The *AllReferencesAreBound* assertion (line 1) specifies that a mobile device contained in a room (line 3) implies that all its references are bound to a service provided either by a printer or a screen (line 4). This assertion is verified on all the instantiable model in a large scope (line 7). The analyser doesn’t find any counter-example and that is why it assures that when a mobile device is added to the room all the expected bindings are well established. This assertion shows that the self-configuration policy specification produces the expected result.

```

1 assert AllReferencesAreBound {
2   all room : Room, md : MobileDevice {
3     room.component[md]
4     implies all ref : md.references | ref.boundToIn room.(printer+screen).services
5   }
6 }
7 check AllReferencesAreBound for 10 expect 0

```

Dynamic properties checking: A more interesting use of the Alloy Analyser is to find non-explicit dynamic properties. The following assertion specifies that a *MobileDevice* primitive dynamically added in a *Room* composite implies that this primitive is also in the *mobileDevices* set of the *Room* composite. The analyser doesn’t find any counter-example and it proves that an explicit constraint on the component model implies an implicit constraint on the self-configurable architecture. Indeed the *addComponent* predicate formalizes the adding of a component in a composite by preserving the state of the composite. The following

satisfied assertion proves that if this predicate is applied on a *Room* composite and a *MobileDevice* primitive it implicitly implies that the *MobileDevice* primitive is also contained in the *mobileDevices* set of the *Room* composite. Even if this fact result from the conjunction of all constraints of the whole system, we want to highlight the fact that this constraint has never been expressed and that is a consequence of other constraints.

```

assert AddComponentImpliesMobileDeviceInRoom {
  all room1, room2 : Room, md1, md2 : MobileDevice {
    addComponent[room1, md1, room2, md2]
    implies md2 in room2.mobileDevices
  }
}
check AddComponentImpliesMobileDeviceInRoom for 10 expect 0

```

4 Related Work

In [3], Bradbury et al. highlight that formal methods are used to provide formal specification languages for designing dynamic software architectures. Works presented in [1], [5] and [6] are also based on logic-based formalisms but they aim at providing formal specification languages where our work provides rigorous and formal methodology to specify, verify and analyse self-configurable component-based systems on top of the use of a formal specification language.

In the domain of CBSE, Architectural Description Language (ADL) have been proposed in order to describe the configuration and the assembling of component-based systems [14]. Generally, the semantics of the underlying component model and of the description language are not clear and are hard-coded in their compiler/interpreter. Nevertheless, two works aim to describe dynamic architectures. The Plastik framework [8] provides a unique formalism (extending Acme/Armani ADL) to specify dynamic architecture (implemented with the OpenCOM component model [4]). Armani (now full part of Acme) allows to set invariants on architectures and some additional statements allows to imperatively describe the architectural reconfigurations Wright [2] is an ADL based on formal method, i.e., the Communicating Sequential Processes (CSP) process algebra and allows to formalize the dynamic behaviour of architectural connections. FracToy approach explicitly focuses on the description of component-based systems and allows to describe and reason on the architectural evolution of the system. The use of Alloy provides an unified, declarative, and constraint-based way of description.

Among Alloy community, Alloy has been already used in CBSE. In [7], Darwin ADL has been formalized with Alloy. This work presents a formalization of the Darwin component model and specifies an architecture built on to top of this model. In this work, constraints are only to express static invariants on the architecture. In [13], a way to formally express and verify properties of Acme architectural styles. Acme styles are mapped to Alloy in order to use the Alloy Analyser to check consistency and properties on these styles. In this work, the dynamic nature of software is not considered. Other works focus on the way to modelize existing component models using Alloy. It is the case for COM in [11]

and Fractal in [15]. These works aim to formally specify component models that are originally specified in natural language. Thereby, they can highlight properties on the model that are ambiguous in the textual specification. The FracToy approach is not dedicated to a specific component model and allows, in addition, to specify, verify, and analyse both the component model and the self-configurable architecture built on top of these component model.

5 Conclusion and Future Work

In this paper, we have presented FracToy, a rigorous and formal methodology for specifying, verifying and analysing self-configurable component-based systems. This methodology is divided into two main steps: specify the component model and specify the self-configurable architecture.

The FracToy methodology was applied to design the *Room* self-configurable component-based system, both the underlying component model and the self-configurable component-based system. This example has shown how to efficiently use the Alloy analyser in order to exhibit static/dynamic and not necessary explicit properties on the architecture. The Alloy formal specification language proves that it fits to the specification of such systems. Indeed the underlying theory of Alloy, i.e., the set theory, is closed to the component-based programming and its analyser allows fast analysis, debugging, and visualizing. Moreover, this approach provides a unique paradigm for specifying, verifying and analysing systems. In addition, the first-order relational logic approach allows to design self-configurable systems in a declarative and constraint-based way without considering syntactic and technical concerns. Thus, specifications describe what the system should be, not how the system should do it. The system is described according to the different states that it can reach instead of describing the sequence of operations to execute to reach a certain state.

Nevertheless, the FracToy approach is limited by a built-in limitation of Alloy. Indeed, as other model finder, all Alloy model instantiations has to be performed in a defined scope. As a consequence, highlighted properties are fully true only in this scope. In addition, by writing the *Room* use case in Alloy, we have identified some recurring syntactic patterns and that specification auto-generation can be expected. That is why, on the short term, we plan to add syntactic sugar on top of the FracToy description to fill this gap.

References

1. Aguirre, N., Maibaum, T.: A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In: ASE 2002: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Washington, DC, USA, p. 271. IEEE Computer Society, Los Alamitos (2002)
2. Allen, R.J.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon University (May 1997)

3. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: WOSS 2004: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 28–33. ACM, New York (2004)
4. Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems* 26, 1–42 (2008)
5. de Paula, V.C.C.: ZCL: A Formal Framework for Specifying Dynamic Software Architectures. PhD thesis, Federal University of Pernambuco (1999)
6. Endler, M., Wei, J.: Programming generic dynamic reconfigurations for distributed applications. In: Proceedings of the International Workshop on Configurable Distributed Systems, pp. 68–79. IEE (1992)
7. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: WOSS 2002: Proceedings of the first workshop on Self-healing systems, pp. 33–38. ACM, New York (2002)
8. Gomes, A.T.A., Batista, T.V., Joolia, A., Coulson, G.: Architecting Dynamic Reconfiguration in Dependable Systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems IV*. LNCS, vol. 4615, pp. 237–261. Springer, Heidelberg (2007)
9. Jackson, D.: Alloy: a Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11(2), 256–290 (2002)
10. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
11. Jackson, D., Sullivan, K.: COM Revisited: Tool-Assisted Modelling of an Architectural Framework. In: SIGSOFT 2000/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 149–158 (2000)
12. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *Computer* 36, 41–50 (2003)
13. Kim, J.S., Garlan, D.: Analyzing Architectural Styles with Alloy. In: ROSATEA 2006: Proceedings of the ISSA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 70–80. ACM, New York (2006)
14. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26, 70–93 (1997)
15. Merle, P., Stefani, J.-B.: A formal specification of the Fractal component model in Alloy. Technical Report RR-6721, INRIA (November 2008)
16. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Number 0-201-74572-0 (2002)

Trace Specifications in Alloy

Jeremy L. Jacob

University of York
Jeremy.Jacob@cs.york.ac.uk

Abstract. Safety properties of a system may be specified by constraining the sequences of interactions of the system with its environment. This paper shows how to encode specifications in such a style using Alloy.

1 Introduction

Alloy^[1] is a light-weight modelling formalism whose underlying mathematics is the relational calculus; its tool is a model checker which can both refute invalid claims and find examples that satisfy consistent first-order predicates. Jackson shows how to use Alloy to model systems in a state-based style that follows Z (and, in some respects, VDM), but where operations are not ‘first-class’^[2]. He also sketches how operations can be made first class.

The purpose of this paper is to explain a modelling style that omits state to leave nothing *but* the events. Essentially, it allows specifications in the style of Hoare, as predicates describing the allowed sequences of events (or *traces*) of a system^[3]. Traces allow a complete description of the safety properties of a system; they do not allow us to discuss non-determinism and termination properties (the *failures* model is needed for those) or liveness properties. Safety properties are described pointwise: a system satisfies a specification if every one of its traces satisfies the property, and so refinement is modelled by implication. (Certain properties, such as confidentiality, are only describable in terms of the entire set of traces of the system^[4]; such properties are outside the scope of this paper.)

A sequence of all the events in a system’s history is (if we ignore clock time) the most detailed information about the state of a non-terminated system to which we have access. A trace may contain more information than we need to construct the current state (or states, in the presence of non-determinism), but it certainly contains enough. Thus the trace of events so far is a universal form for specification states.

Traces can be pragmatically useful, too. I have found errors in a state-based specification by considering a trace-based version, because no history is ever thrown away in a trace model. The example involved an operation that adds

¹ See <http://alloy.mit.edu/>.

pairs to a relation (let us say, a ticket and a time of issue); the specification was supposed to have the property that at most one ticket should be added at any given time. The original description of the ‘delete-ticket’ operation removed all information from the database associated with the ticket. However, this deleted too much information, and a later ticket could be recorded as issued at the same time as a ticket that had been deleted. Consideration of the trace-based specification made this error obvious, because all the information about previously entered tickets was in the trace.

The major disadvantage of the style is that interesting traces are relatively long. In one example below, at least 9 events are necessary to get all possibilities for one cycle of a protocol communicating over a one-place buffer (and a second cycle may have started but not finished). Checks of interesting properties are expensive to analyse. One aim of this paper is to encourage tool developers and maintainers to improve the analysis of ordered signatures.

The encoding of traces presented here is different to that given by Bolton [1]; her encoding is useful for stating and proving higher level properties of the refinement ordering. The encoding in this paper is oriented towards specification.

2 Encoding

The encoding strategy is to use a signature of `Events`, and to encode individual events as sub-signatures of `Event`. Elements of the sub-signatures represent occurrences of the event. The Alloy `ordering` module is used to arrange the event occurrences in the analysis scope into a trace. (The `ordering` module defines a total order over a signature, together with some utility functions and relations such as `first`, `next` and `last`.) The Alloy analyzer can then find all traces, of some fixed length [2], that satisfy a given property.

In order to generate ‘real’ examples we ensure that the properties are *prefix-closed*, that is if they apply to a trace then they also apply to every initial sub-trace. In symbols, in the context of `sig Event{}` and `open util/ordering[Event]`, the predicate `pred p[t : set Event]{...}` is prefix-closed if and only if:

$$\text{all } e : \text{Event} \mid \text{let } \text{pfx} = \text{prevs}[e] \mid \text{p}[\text{pfx} + e] \text{ implies } \text{p}[\text{pfx}]$$

As prefix-closure is a second-order predicate it cannot be programmed (using the encoding of this paper) in Alloy, and consequently we have to program it explicitly. Trace specifications need not be prefix-closed in practice, as a correct system’s evolution guarantees prefix-closure; we need the property to restrict the model checker to ‘real’ instances.

The encoding is best explained by examples. The first (Sect. 3) uses Hoare’s examples of vending machines, while the second (Sect. 4), contains the more complex example of the triple-redundancy protocol.

² Because of the way that the `ordering` utility is encoded the set of `Events` must be non-empty.

```

2  module vending_machines
3
4  open util/ordering[Event]
5  fun fst:Event{ordering/first}
6  fun nxt:Event->Event{ordering/next}
7  fun upto[e:Event]:set Event{prevs[e]+e}
8
9  abstract sig Event{}
10
11 sig Coin extends Event{}
12
13 pred no_vendor_loss[product : set (Event-Coin)]{
14     all e:Event | let pfx=upto[e] | #(product&pfx)<=#(Coin&pfx)
15 }
16
17 pred max_customer_loss[product : set (Event-Coin), max_loss : Int]{
18     all e:Event | let pfx=upto[e] | #(Coin&pfx)-max_loss<=#(product&pfx)
19 }

```

Listing 1. Simple trace specification: start of module

3 Example: Vending Machines

3.1 Module Heading

Listing 1 shows the start of the module. It declares a signature to represent events in the system (Line 9), ensures that they are totally ordered in a way displayable by the visualiser (Lines 4-6), and defines a function to calculate prefixes (Line 7).

Modelling starts on Line 11, where a signature defines the events that represent payment (Coin); atoms of this signature represent instances of the event. Two utility predicates capture properties a designer may wish to assert of all behaviours of a vending machine system, with respect to Coin. The first is that the vendor never makes a loss (no_vendor_loss, Lines 13-15): the number of products delivered never exceeds the number of coins paid. The second (max_customer_loss, Lines 17-19) states that the customer may make a loss when buying products, but no larger than some value max_loss. Note that both of these predicates are written to be prefix-closed.

3.2 Very Simple Vending Machines

The next part of the module (Listing 2) is inspired by the ‘chocolate vending machine’ specification of Hoare [3, §1.10].

We add a signature to model delivery of a chocolate bar (Choc, Line 21), and a predicate to model a machine that satisfies the vendor of chocolate bars, but may only satisfy the customer up to some possible maximum loss (choc_vm,


```

21 sig Choc extends Event{}
22
23 pred choc_vm[max_loss : Int]{
24   no_vendor_loss[Choc] and max_customer_loss[Choc, max_loss]
25 }
26
27 choc_vm0: run{choc_vm[0]} for 8 but 0 Drink_Event expect 0
28 choc_vm1: run{choc_vm[1]} for 8 but 0 Drink_Event expect 1
29 choc_vm2: run{choc_vm[2]} for 8 but 0 Drink_Event expect 1
30 choc_vm2a: run{choc_vm[2] and #Choc<#Coin} for 8 but 0 Drink_Event expect 1
31 vm1_is_vm2: check{
32   choc_vm[1] implies choc_vm[2]
33 } for 6 but 0 Drink_Event expect 0
34
35 pred alternate_coin_choc{
36   fst in Coin
37   nxt in Coin->Choc + Choc->Coin
38 }
39
40 choc_alt_is_vm1: check{
41   alternate_coin_choc iff choc_vm[1]
42 } for 6 but 0 Drink_Event expect 0

```

Listing 2. Simple trace specification (cont.): simple vending machines. The signature `Drink_Event` is defined later; here it is effectively ignored.

Lines 23–25). This predicate, when instantiated, defines a class of acceptable behaviours for a vending machine.

Five commands are given in Lines 27–33. The first three generate example traces for chocolate vending machines in which the customer may make zero, one or two coins loss, and in the fourth case we restrict attention to examples of `choc_vm[2]` in which the vendor makes a profit over-all, but not necessarily in any prefix. (The scopes generate sequences of six events; it also forbids atoms of the signature `Drink_Event`, a signature which is introduced in Section 3.3.) The first command, `choc_vm0`, finds no instance because the ordering utility insists that there be at least one element in the ordering; it would be nicer if ordering allowed the empty ordering over an empty sequence as there is exactly one consistent trace that satisfies no loss to either party: the empty trace. The fifth command, `vm1_is_vm2`, is a refinement check: that every observable behaviour of `choc_vm[1]` is a possible observable behaviour of `choc_vm[2]`.

Lines 35–38 define a property, `alternate_coin_choc`, that says payments and deliveries strictly alternate, that the first event must be a payment, and that there are no other types of event. The command in Lines 40–42 checks that (within the scope) payment followed by strict alternation of delivery and payment is an identical restriction on behaviour to that of no vendor loss combined with a maximum loss to the customer of one coin. Thus, in this style, with one command we can compare the entire behaviour of two systems.

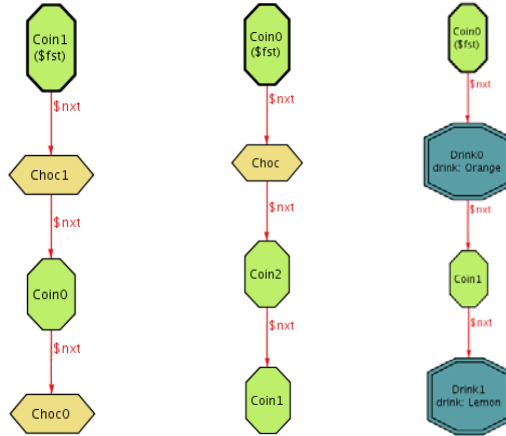


Fig. 1. Three traces of length 4. Two are from `choc_vm[2]`: a non-overlapping pair of transactions is illustrated on the left, while one transaction followed by the start of two other transactions is illustrated in the middle. On the right is a counterexample to the proposition that `drink_random_b` allows only behaviours allowed by `drink_random_a`.

The vending machines described so far are very simple. Example traces are given in Figure 1. In the next section we describe vending machines with more complex behaviours.

3.3 More Complex Vending Machines

In this subsection we describe a family of vending machines that can deliver products from two classes, and also allow the user to switch between products. This example is also derived from Hoare [3, §1.1.4]. The code can be found in Listing 3.

The two products are drinks of orange and lemon, declared on Line 44; these signatures represent values and so are not extensions of `Event`. The events are: selecting which beverage the machine will deliver next (`Select`), and delivering it (`Drink`). These events are declared in Lines 45–46; note that they have a value field to record which beverage is involved. A different modelling strategy for the events is to use four separate signatures, but that is less convenient.

The property, `default`, in Lines 48–51 defines what it means for a selection event to affect later deliveries, and also describes what happens before a selection is made. If the user has made a selection, then the product selected is that delivered, otherwise any product from the set of defaults is delivered.

On Lines 53–57 a behaviour, `drink_orange_1`, is defined in which the vendor makes no loss on drinks, the customer may make a loss of one coin and the customer may select between orange and lemon, with the default being orange.

The behaviour `drink_orange_1` allows a customer to insert a coin and then the flexibility to make any number of selections before taking a drink. We may

```

44 enum Beverage{Orange, Lemon}
45 abstract sig Drink_Event extends Event{drink : Beverage }
46 sig Select, Drink extends Drink_Event{}
47
48 pred default[d : set Beverage]{
49   all e : Drink | let s=max[Select&prevs[e] |
50     e.drink in (some s implies s.drink else d)
51 }
52
53 pred drink_orange_1{
54   no_vendor_loss[Drink]
55   max_customer_loss[Drink, 1]
56   default[Orange]
57 }
58
59 pred no_choice_within_transactions{
60   all e : Select | let pfx=prevs[e] | #(Coin&pfx)=#(Drink&pfx)
61 }
62
63 pred drink_orange_2{
64   no_vendor_loss[Drink]
65   max_customer_loss[Drink, 2]
66   default[Orange]
67   no_choice_within_transactions
68 }
69
70 pred drink_random_a{
71   no_vendor_loss[Drink]
72   max_customer_loss[Drink, 1]
73   default[Orange] or default[Lemon]
74 }
75
76 pred drink_random_b{
77   no_vendor_loss[Drink]
78   max_customer_loss[Drink, 1]
79   default[Orange+Lemon]
80 }
81
82 rand_a_refines_rand_b: check {
83   drink_random_a implies drink_random_b
84 } for 6 but 0 Choc expect 0
85
86 rand_b_refines_rand_a: check{
87   drink_random_b implies drink_random_a
88 } for 4 but 0 Choc expect 1

```

Listing 3. Simple trace specification (cont.): less simple vending machines

wish to restrict selection to between transactions. This property is captured by the predicate `no_choice_within_transactions` (Lines 59–61); a transaction is complete when every insertion of a coin has a matching delivery of a drink: this predicate restricts choice events to exactly these situations. The behaviour described on Lines 63–68 is an example system that has this property.

The two systems on Lines 70–80 illustrate two different types of choice. In the first system, `drink_random_a`, the system decides on start-up whether the default is orange or lemon; however in `drink_random_b` a choice is made at each delivery before the first selection event. The check on Lines 82–84 tests whether the first system refines the second (it does). The check on Lines 86–88 shows that the converse does not hold; a counter example is given in Figure 11.

4 Example: Triple-Redundancy Protocol

A more complex class of examples are those composed of concurrent subsystems. Among the simplest of these are the simple protocols: we describe one in which the transmitter repeats each input bit three times over the network, and the receiver reads bits in groups of three from the network, passing on the majority value. The network is represented by a one-place buffer, and a one-place corrupting buffer. Also, we describe an unbounded buffer and check that the sender-network-receiver system refines a buffer, for a network implemented as a one-place buffer and as a corrupting buffer.

4.1 Model

Module heading: Listing 12 contains the header.

As for the vending machines, we declare a carrier signature `Event`, but this time every event refers to some value (either T or F). Observable classes of events are communications on the five synchronisation channels `In`, `Out`, `Mid`, `MidI` and `MidO`. `In` and `Out` are used for communications between the environment and the protocol; the other three channels will be used to join together components of the protocols.

Useful functions calculate the initial sub-sequence of some subclass of events `upto`, but excluding any event; the `prefix`, which is similar but includes the event; and the ordering relation restricted to a class of events (`subseq`). (Note that these are a different set of utilities to those defined in Listing 11.) The fourth function (`show_match`) is a utility to persuade the visualiser to show how inputs and outputs in different sub-systems match; it is not a part of the modelling.

Buffers: Listing 13 contains the specification of two buffers.

A buffer (`buffer`) has the property that if an n th output exists, then the n th input is earlier and has the same value. Note that this places no constraints on the capacity of the buffer.

A one-place buffer (`unary_buffer`) is a buffer that strictly alternates inputs and outputs (compare with `alternate_coin_choc` above, Listing 12). Its definition includes the constraint that the capacity is a single datum.

```

1  module trp
2
3  open util/ordering[Event]
4  fun nxt:Event→Event{{a:Event, b:next[a]}}
5
6  fun upto[xs:set Event, x:Event]:set xs{prevs[x]&xs}
7  fun prefix[xs:set Event, x:Event]:set xs{upto[xs,x]+x&xs}
8  fun subseq[xs:set Event]:xs→xs{{disj a,b:xs | lt[a,b] and no nexts[a]&upto[xs,b]}}
9  pred show_match[disj ins, outs : set Event]{
10     some c : ins→outs | c={i : ins, o : outs | #(upto[ins, i])=#(upto[outs, o])}
11 }
12
13 abstract sig Event{value : Value}
14
15 enum Value{T, F}
16
17 sig In, Out, Mid, Midl, MidO extends Event{}

```

Listing 4. The triple-redundancy protocol: start of module

```

18
19 pred buffer[disj ins, outs : set Event]{
20     all e : outs |
21     one {i : ins&prevs[e] | #(upto[ins,i])=#(upto[outs,e]) and i.value=e.value}
22 }
23
24 buffer: run{
25     buffer[In,Out] and show_match[In, Out]
26 } for 16 but 0 Mid, 0 Midl, 0 MidO, 3 int expect 1
27
28 pred unary_buffer[disj ins, outs : set Event]{
29     buffer[ins, outs]
30     subseq[ins+outs] in ins→outs + outs→ins
31 }
32
33 unary_buffer: run {
34     unary_buffer[In, Out] and show_match[In, Out]
35 } for 16 but 0 Mid, 0 Midl, 0 MidO, 3 int expect 1
36
37 unary_buffer_is_buffer: check {
38     unary_buffer[In, Out] implies buffer[In, Out]
39 } for 11 but 0 Midl, 0 MidO, 3 int expect 0

```

Listing 5. The triple-redundancy protocol: buffers

The first two commands show how the utility `show_match` can be used to make the visualisation more helpful. The third allows us to have some confidence that a one-place buffer is a buffer (although this is a consequence of the way `unary_buffer` is defined).

Transmitter: Listing 6 contains the specification of the transmitter.

An output of some value is allowed only if that value has been input in the previous three interactions (`prev3`), and no other input has occurred in that triple.

An input is allowed either at the start of the protocol or when there has not been an input for three events.

```

40 pred transmitter[disj ins, outs : set Event]{
41   let next=subseq[ins+outs] |
42   all e : ins+outs |
43     let prev3=next.e+next.(next.e)+next.(next.(next.e)) {
44       e in outs implies (one prev3&ins and (prev3&ins).value=e.value)
45       e in ins implies (no prev3 or prev3 in outs)
46     }
47 }
48
49 transmitter: run {
50   transmitter[In, Mid]
51 } for 16 Event, 0 Out, 0 MidI, 0 MidO, 3 int expect 1

```

Listing 6. The triple-redundancy protocol: transmitter

Receiver: Listing 7 contains the specification of the receiver.

```

52 pred receiver[disj ins, outs : set Event]{
53   let next = subseq[ins+outs] |
54   all e : ins+outs |
55     let prev3=next.e+next.(next.e)+next.(next.(next.e)), pvi=#prev3&ins {
56       e in outs implies {
57         pvi = 3
58         e.value = (#(prev3&value.T)>1 => T else F)
59       }
60       e in ins implies pvi < 3
61     }
62 }
63
64 receiver: run {receiver[Mid, Out]} for 16 Event, 0 In, 0 MidI, 0 MidO, 3 int expect 1

```

Listing 7. The triple-redundancy protocol: receiver

An output is allowed if it follows exactly three inputs, and its value is the majority value of those inputs.

An input is allowed if there are fewer than three immediately preceding inputs.

4.2 The Transmitter/Receiver Pair Is a Protocol

A transmitter/receiver pair is defined to be a protocol exactly when the pipeline that is the transmitter feeding the receiver refines a buffer. This is easily checked (in some finite scope): see Listing 8.

```

66 trp_traces: run{
67     transmitter[In, Mid]
68     receiver[Mid, Out]
69     show_match[In, Out]
70 } for 16 but 0 MidI, 0 MidO, 3 int expect 1
71
72 trp_correct: check{
73     transmitter[In, Mid] and receiver[Mid, Out] implies buffer[In, Out]
74 } for 8 but 0 MidI, 0 MidO, 3 int expect 0

```

Listing 8. The triple-redundancy protocol: correctness of protocol

The first command generates examples of the system (using the `show_match` utility to enhance the output of the visualiser). Note how communication between the two sub-systems arises from synchronisation on the events in the class `Mid`. The second command checks the correctness of the pair as a protocol.

4.3 The Triple-Redundancy Protocol Connected via a Good Network

In Listing 9 we see commands to generate examples of two systems that differ in their networks. The first is a one-place buffer and the second an unbounded buffer. Again, notice the use of `show_match` to enhance visualisations.

Commands to check the correctness of these systems are easily written, but they are not presented here.

```

75 trp_1_traces: run{
76     transmitter[In, MidI]
77     unary_buffer[MidI, MidO]
78     receiver[MidO, Out]
79     show_match[In, Out]
80     show_match[MidI, MidO]
81 } for 16 Event, 0 Mid, 3 int expect 1
82
83 trp_oo_traces: run{
84     transmitter[In, MidI]
85     buffer[MidI, MidO]
86     receiver[MidO, Out]
87     show_match[In, Out]
88     show_match[MidI, MidO]
89 } for 16 but 0 Mid, 3 int expect 1

```

Listing 9. The triple-redundancy protocol: good networks

4.4 The Triple-Redundancy Protocol Connected via a Bad Network

Last of all we construct a network that corrupts, but not too often, and see if the protocol mitigates the corruption. A corrupting network is defined and used in Listing 10.

The sub-system `corrupter` describes a buffer, except that an output may deliver a corrupt value whenever neither of the previous two outputs are corrupt.

Predicate `show_corruption` is another utility predicate, in this case to visualise corruptions.

The command `trp_corrupter_traces` lets us investigate examples of the behaviour of the triple-redundancy protocol in the presence of (mild) corruption. The final check gives confidence that the triple-redundancy protocol is useful,

```

90 pred corrupter[disj ins,outs : set Event]{
91   all e : outs | let pos=#(upto[outs, e]) {
92     some i : ins&prevs[e] | #(upto[ins, i])=pos
93     ((val[outs, pos-1]!=val[ins, pos-1] or val[outs, pos-2]!=val[ins, pos-2])
94     implies
95     val[outs, pos]=val[ins, pos])
96   }
97   subseq[ins+outs] in ins->outs + outs->ins
98 }
99
100 pred show_corruption[disj ins, outs : set Event]{
101   some b : ins->outs |
102   b={i : ins, o : outs | #(upto[ins, i])=#(upto[outs, o]) and i.value!=o.value}
103 }
104
105 corrupter: run {
106   corrupter[In, Out]
107   show_match[In,Out]
108   show_corruption[In, Out]
109 } for 16 but 0 Mid, 0 MidI, 0 MidO, 3 int expect 1
110
111 trp_corrupter_traces: run{
112   transmitter[In, MidI]
113   corrupter[MidI, MidO]
114   receiver[MidO, Out]
115   show_match[In, Out]
116   show_match[MidI, MidO]
117   show_corruption[MidI, MidO]
118 } for 16 but 0 Mid, 3 int expect 1
119
120 trp_useful: check{
121   transmitter[In, MidI] and corrupter[MidI, MidO] and receiver[MidO, Out]
122   implies
123   buffer[In, Out]
124 } for 9 but 0 Mid, 3 int expect 0

```

Listing 10. The triple-redundancy protocol: a bad network

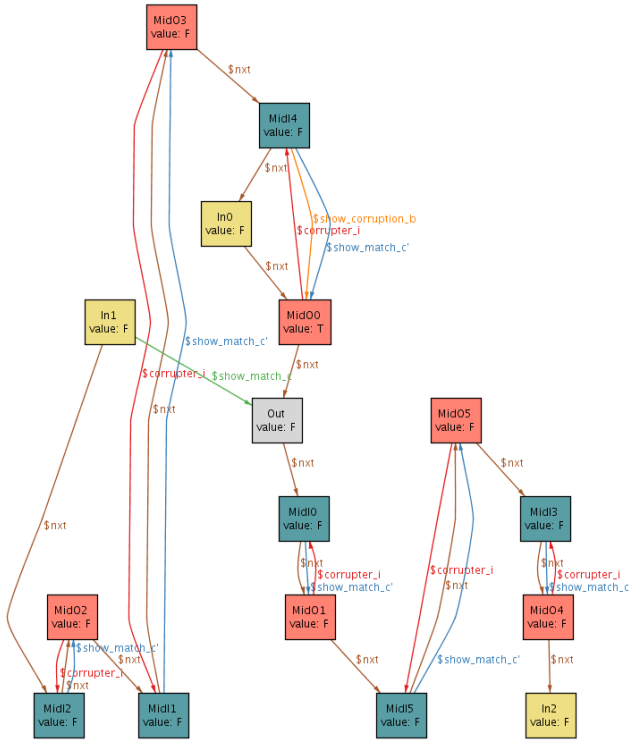


Fig. 2. A trace of the triple-redundancy protocol over a corrupting network (using ‘Magic layout’). The trace starts at ‘In1’ and follows the $\$nxt$ arrows to end at ‘In2’. Note that the third bit of the first cycle is corrupted by the network (indicated by the ‘ $\$show_corruption_b$ ’ arrow), but that the eventual output, Out is correct; no other bits in the first cycle nor any in the second are corrupted.

as well as correct, by showing that the whole system corrects for a degree of corruption in the network.

Figure 2 shows a trace obtained from running the protocol with a corrupting network.

5 Conclusions

Properties of traces —with traces encoded as a total ordering of the events in scope— are a good way to think about safety properties of a variety of systems. This style is flexible, allows simple statements of refinement and shows examples in a useful way through the Alloy analyzer.

The full gamut of safety properties can be encoded using *failures*. These allow maximal degrees of non-determinism to be specified, and to rule out unwanted deadlocks. A failure is a pair of a trace and a set (see, for example, Hoare for details [3] and Bolton’s encoding [1]). It is not clear whether the gains

of such an encoding are worth the effort for specification, as the loading on the model checker would mean only very small examples could be checked. Other model checkers (such as FDR2 from Formal Systems (Europe) Ltd., <http://www.fsel.com/>) are optimised for such checking (although their language is that of processes rather than individual observations).

Acknowledgements. Thanks to the anonymous referees, to Michael Banks and to members of the 3rd year Formal Specification of Systems class for commenting on previous versions of this paper.

References

1. Bolton, C.: Using Alloy to Automatically Verify the Soundness of the Simulation Rules for Reasoning about State-Based and Event-Based Models (2002), <http://alloy.mit.edu/community/files/simulationDiscussion.ps>
2. Jackson, D.: Software Abstractions: Logic, language and analysis. MIT Press, Cambridge (2006)
3. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice Hall, Hemel Hempstead (1985)
4. Jacob, J.L.: Basic theorems about security. *Journal of Computer Security* 1, 385–411 (1992)

An Imperative Extension to Alloy

Joseph P. Near and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{jnear,dnj}@csail.mit.edu

Abstract. We extend the Alloy language with the standard imperative constructs; we show the mix of declarative and imperative constructs to be useful in modeling dynamic systems. We present a translation from our extended language to the existing first-order logic of the Alloy Analyzer, allowing for efficient analysis of models.

1 Introduction

We present an extension to the Alloy language [1] for the specification of dynamic systems. The typical approach to modeling dynamic systems, and the one taken by Z [2], VDM [3], and DynAlloy [4,5], is to model state changes using pre- and post-conditions on each transition. Both the existing idioms for modeling dynamic systems in Alloy and our approach support this technique; we add the standard imperative constructs: assignment, sequential composition, guards, and loops. We give these operators the expected, *operational*, semantics.

Moreover, our language extension allows for the separation of the static and dynamic elements of a model. Our extension allows dynamic operations to be added to a static model: it makes updates to mutable state explicit and separates imperative operations from static properties. This separation of concerns is important to the design of a system, and is not well-supported by the Alloy idioms currently in use.

The use of imperative operators in specifications simplifies the process of implementation. Using our language extension, modelers have the option of refining a specification (in the style of Morgan [6]) until the modeler can easily translate it into an imperative implementation. Each refinement step is automatically checked by the Alloy Analyzer to ensure that no errors have been made.

These advantages come at no loss of expressive power. We place no restrictions on the existing language, and allow actions to be defined declaratively, using pre- and post-conditions; our framework and composition operators also apply to these declarative actions.

The contributions of this paper are:

- an extension to the Alloy language consisting of the standard imperative operators (Section 3);
- a set of examples showing how the extension may be used to model dynamic systems concisely (Section 4);

- a translation from the action language of the extension to the first-order logic supported by the Alloy Analyzer, allowing for the efficient analysis of models written using the extension (Section 5).

2 Alloy and Dynamic Systems

Alloy [1] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. As such, few features are provided beyond first-order logic and transitive closure, making the semantics of the language easily expressible, understandable, and extendable.

The Alloy Analyzer supports fully automatic analysis of Alloy models. While this analysis is bounded and thus not capable of producing proofs, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [7]—which claims that most inconsistent models have counterexamples within small bounds—means that modelers may have high confidence in the results. This sacrifice of completeness in favor of automation is in line with the *lightweight formal methods* philosophy [8].

Alloy’s universe is made up of uninterpreted atoms; *signatures* define the sets into which these atoms are partitioned. For example [1], the following signatures define sets of names and addresses:

```
sig Name {}
sig Addr {}
```

Similarly, this signature defines an address book with a field “addr” mapping names to addresses:

```
sig Book { addr: Name → lone Addr }
```

Operations that modify the state of an address book may be defined as *predicates* over pre- and post-states:

```
pred add [b, b': Book, n: Name, a: Addr] {
  b'.addr = b.addr + n→a }
```

We can use Alloy’s “**check**” command to check that that the “add” operation correctly updates the address book.

```
check {
  all b, b': Book, n: Name, a: Addr |
  add[b, b', n, a] ⇒ n.b'.addr = a }
```

The pre- and post-state idiom is well-known, both in the context of declarative specification and in functional programming. It is the basis of the idioms for modeling dynamic systems in Alloy and of the monadic theory of state used in functional languages such as Haskell. While this technique often produces concise, readable models, it is not adept at expressing certain types of imperative control flow. The following excerpt, for example, is taken from a previously published Alloy model of a flash filesystem [9], and uses a common trace-based idiom:

```

some stateSeq: StateSeq |
  stateSeqConds[stateSeq, numBlocksToProgram + 1] &&
  all trscSeq_idx : stateSeq.butlast.inds |
    programBlock[stateSeq, trscSeq_idx, cfsys, inode, startBlockIdx]

```

The specification for the flash memory requires blocks to be written in sequence. In the traditional approach, this would be expressed using multiple operations, one for each write; a sequence of such operations would then be shown to refine an abstract write that occurs in a single step. This approach can be tedious and unnatural, however, as it becomes necessary to encode the control flow explicitly in the state, using preconditions to constrain the ordering. Consequently, many modelers prefer to describe such a behavior using a single operation whose execution involves multiple steps. This notion has no standard formulation in Alloy; here, the modeler has introduced a special signature, “StateSeq”, to model a sequence of states, which happens to be used only inside this operation.

The Haskell community, having encountered precisely the same situation, introduced special syntax for expressing sequential operations. One way to view this paper is as an attempt to provide the same facilities to Alloy modelers. Using our language extension, the excerpt above can be written as follows:

```

Cnt.idx := 0 ;
loop {
  programBlock[Cnt.idx, cfsys, inode, startBlockIdx];
  Cnt.idx := Cnt.idx + 1
} && after Cnt.idx = numBlocksToProgram

```

An operational language extension, with operational semantics, can thus make some models easier to write. The basic operations—state update, conditionals, loops, and so on—can be proved correct. All models written in the extension use the same mechanism for expressing dynamic operations, making models easier to read. Imperative operators can make sequential operations more concise. And models written using a standard operational mechanism can be optimized for efficient analysis.

3 Language Extension

A small extension to the Alloy language, summarized in this section, supports the modeling of dynamic systems.

3.1 Dynamic Fields

Immutable fields are declared in the traditional way:

```

sig Addr {}
sig Name {}

```

Mutable fields, whose values may vary with time, are defined using the “**dynamic**” keyword:

```

one sig Book { addr: dynamic (Name →lone Addr) }

```

3.2 Named Actions

Named actions can be defined at the top level, and can be invoked from within other actions. Adding an entry to the address book, for example, can be written as a named action that adds the appropriate tuple:

```
action add[n:Name, a:Addr] {
  Book.addr := Book.addr + (n → a) }
```

The deletion operation, on the other hand, removes all tuples containing a given name from the book:

```
action del[n:Name] {
  Book.addr := Book.addr - (n → Addr) }
```

3.3 Action Language

Our action language includes operators for imperative programming: field update, sequential composition, and loops. Pre- and post-conditions employ boolean-valued formulas (written φ) with the existing syntax and semantics of Alloy.

$Act ::= o_1.f_1, \dots, o_n.f_n := e_1, \dots, e_n$	(field updates)
$Act ; Act$	(sequential composition)
loop { Act }	(loop)
$action[a_1, \dots, a_n]$	(action invocation)
before φ after φ	(pre- and post-conditions)
some $v : \tau$ Act	(existential quantification)
$Act \Rightarrow Act$ $Act \wedge Act$ $Act \vee Act$	

A *field update* action changes the state of *exactly* those mutable fields mentioned, *simultaneously*. The action

```
b.addr := b.addr + (n → a)
```

for example, adds the mapping $n \rightarrow a$ to the address book b , while

```
a.addr, b.addr := b.addr, a.addr
```

swaps the entries of address books “a” and “b”.

Sequential composition composes two actions, executing one before the other:

```
add[n,a]; del[n,a]
```

performs the “add” operation and then the “del” operation.

A loop executes its body repeatedly, nondeterministically choosing when to terminate. The standard conditional loop may be obtained through the use of a post-condition; the action

```
loop { dec[Cnt.idx] } && after Cnt.idx = 0
```

for example, runs the “dec” action until “Cnt.idx” reaches zero. Because they are nondeterministic, execution of these loops generally requires backtracking.

We view actions as relations between initial and final states. This view of actions allows for the lifting of the standard logical connectives and existential

quantification into our action language, and for the mixing of declarative constraints with actions. The “**before**” and “**after**” actions, for example, introduce declarative pre- and post-conditions; these act as filters on other actions when combined using the logical connectives. The action

`add[n,a] ⇒ after n.Book.addr = a`

for example, has executions that either end with the correct mappings in the address book or are not executions of “add.”

3.4 Temporal Quantifiers

Actions have as free variables their beginning and ending states. Temporal quantifiers bind these variables: “**sometimes**,” existentially; and “**always**,” universally.

$$\begin{aligned} \varphi ::= & \langle \text{Alloy Formula} \rangle \\ & | \text{ **sometimes** } | \textit{Act} \\ & | \text{ **always** } | \textit{Act} \end{aligned}$$

Given our view of actions as relations, a “**sometimes**” formula holds if and only if the action in its body relates *some* initial and final states; an “**always**” formula holds if and only if it relates *all* states. To visualize the result of adding the mapping $n \rightarrow a$ to the address book, for example, one executes the Alloy command:

`run { sometimes | add[n,a] }`

One can also check that “add” adds the mapping in all cases:

`check { always | add[n,a] ⇒ after n→a in Book.addr }`

4 Examples

4.1 River Crossing

River crossing problems are a classic form of logic puzzle involving a number of items that must be transported across a river. Some items cannot be left alone with others: in our problem, the fox cannot be left with the chicken, or the chicken with the grain. A correct solution moves all items to the far side of the river without violating these constraints. We begin by defining an abstract signature for objects, each of which eats a set of other objects and has a dynamic location. The objects of the puzzle are then defined as singleton subsets of the set of objects. Similarly, an abstract signature defines the set of locations, and two singleton sets partition it into the near and far sides of the river.

```
abstract sig Object { eats: set Object,
                    location: dynamic Location }
one sig Farmer, Fox, Chicken, Grain extends Object {}
abstract sig Location {}
one sig Near, Far extends Location {}
```

We define the “eats” relation to reflect the puzzle by constraining it to contain exactly the two appropriate tuples.

```
fact eating { eats = (Fox →Chicken) + (Chicken →Grain) }
```

The “cross” action picks an object o for the farmer to carry across the river, a new location fl for the farmer, and a (possibly new) location ol for o , and moves the farmer and the object.

```
action cross {
  some o: Object – Farmer, fl: Location – Farmer.location, ol: Location |
  (Farmer.location := fl, o.location := ol) && -- move the object and farmer;
  after (all o: Object |
    o.location = Farmer.location || -- all objects end up with
    o'.location = o.location || -- the farmer, or not with
    (all o': (Object – o) |
      o'.location = o.location ⇒ o !in o'.eats)) }
```

To obtain a solution, we find an execution that begins with all objects on the near side, calls “cross” repeatedly, and ends with all objects on the far side.

```
pred solvePuzzle {
  sometimes |
  before (all o: Object | o.location = Near) && -- objects start on near side,
  loop {
    cross [] -- cross runs repeatedly, and
  } && after (all o: Object | o.location = Far) } -- objects end on far side.
```

The “cross” action relies on the ability to mix declarative and imperative constructs: it chooses an object and a destination nondeterministically and then formulates the requirement that no object be eaten as a postcondition. In obtaining a solution, we have applied another imperative construct—**loop**—illustrating our ability to declaratively construct abstract actions and then compose them imperatively.

4.2 Filesystem

As an example of the addition of dynamic operations to a static model, we present a simple filesystem. We begin with signatures for filenames and paths. File paths are represented by linked lists of directories terminated by filenames.

```
sig Name {}
abstract sig Path {}
sig NonEmptyPath extends Path { first: Name, rest: Path }
sig EmptyPath extends Path {}
```

Next, we define the filesystem: an inode is either a directory node or a file node; a directory node maps names of files and directories to other inodes, and a file node contains some mutable data. The root node is a directory.

```
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
one sig RootNode extends DirNode {}
```



```
sig FileNode extends INode { data: dynamic Data }
sig Data { }
```

We now define operations over this static filesystem, beginning with navigation. We use a global MVar to hold the destination path, the current inode, and the data to be written to or read from the destination. One navigation step involves moving one step down the list representing the destination path and following the appropriate pointer to the corresponding inode.

```
one sig MVar { path: dynamic Path,
  current: dynamic INode, mdata: dynamic Data }
```

```
action navigate {
  MVar.path := MVar.path.rest;
  MVar.current := (MVar.path.first).(MVar.current.files)}
  -- follow the path one step and then
  -- update "current" to point to the
  -- corresponding inode
```

Reading from a file involves calling “navigate” until the destination inode has been reached and then reading its data into “MVar.” Writing, similarly, involves navigation followed by a write.

```
action read {
  loop {
    navigate []
  } && after MVar.current in FileNode;
  MVar.mdata := MVar.current.data }
  -- call navigate repeatedly
  -- until we have reached the file inode
  -- then read its data into MVar
```

```
action write {
  loop {
    navigate []
  } && after MVar.current in FileNode;
  let file = MVar.current |
  file .data := MVar.mdata }
  -- call navigate repeatedly
  -- until we have reached the file inode
  -- take the data from MVar
  -- and write it to the file inode
```

We would like a write to the filesystem followed by a read to yield the written data. We can verify this property by writing arbitrary data to an arbitrary file, reading it back, and checking that the result is the original data. We use a global “Temp” to hold the original data.

```
one sig Temp { tdata: dynamic Data }
```

```
assert readMatchesPriorWrite {
  always |
  before (MVar.current = RootNode &&
    no f: FileNode | f.data = MVar.mdata) &&
  write [];
  Temp.tdata := MVar.mdata;
  read [] =>
  after Temp.tdata = MVar.mdata }
  -- if we begin at the root node,
  -- and no file contains
  -- MVar.mdata
  -- and we write MVar.mdata,
  -- store the original data,
  -- and read back the data
  -- then they're the same
```

This model illustrates the ability to build up multi-step actions using loops and sequential composition, and to verify properties of those actions.

4.3 Insertion Sort

Following Morgan [6], we present insertion sort as a refinement from a declarative specification to a deterministic, imperative implementation. We begin by defining mutable sequences of naturals and a declarative sortedness predicate.

```
sig Sequence { elts: dynamic seq Natural }
pred sorted[elts: seq Natural] { -- each element is less than the next
  all i: elts.inds - elts.lastIdx | let i' = i + 1 | i.elts <= i'.elts }
```

Using this predicate, we can define a declarative sorting operation.

```
action declarativeSort [s: Sequence] {
  some s': Sequence |
  before (sorted[s'.elts] && s.elts = s'.elts) &&
  s.elts := s'.elts }
```

To bring this model closer to executable code, we define insertion sort as a series of swaps of elements of a sequence. We begin with a global counter and a declarative predicate to find the index of a sequence's smallest element, leaving the imperative definition of this predicate for later.

```
one sig Cnt { cur: dynamic Int }
pred minIdx [s: seq Natural, c, i: Int] { -- i is the index greater than c whose
  i >= c && no i': s.inds | i' >= c && i'.s < i.s } -- value in s is smallest
```

Next, we define the insertion step, in which the first element in the sequence is swapped, using relational override (++), with the smallest one.

```
action insertionStep [s: Sequence] {
  some i: s.elts.inds | -- nondeterministically pick an index
  (before minIdx[s.elts, Cnt.cur, i]) && -- whose element is smallest
  Cnt.cur := Cnt.cur + 1, -- and swap it with the first element
  s.elts := s.elts ++((Cnt.cur)→i.(s.elts)) ++(i→Cnt.cur.(s.elts)) }
```

The sorting action simply sets the counter to zero and runs the insertion step to the end of the sequence.

```
action insertionSort [s: Sequence] {
  Cnt.cur := 0;
  loop {
    insertionStep[s]
  } && after Cnt.cur = s.elts.lastIdx }
```

Next, we show that the sort is correct by verifying that an arbitrary sequence is sorted when the sort completes.

```
assert sortWorks {
  all s: Sequence |
  always | insertionSort[s] ⇒ after sorted[s.elts] }
```

We now return to the problem of finding the minimum unsorted element in the sequence. We begin with a bit of global state to hold the current index in the search and the value and index of the minimal element found so far.

one sig Temp {idx: **dynamic** Int, min: **dynamic** Natural, minIdx: **dynamic** Int}

Next, we define an action to iterate over the subsequence *s*, checking each element against the minimal one found so far.

```
action findMin[s: Sequence] {
  Temp.idx := Temp.idx + 1;           -- increment the current index
  -- if the current value is less than the previous minimum, remember it
  (before Temp.idx.(s.elts) < Temp.min =>
    (Temp.min := Temp.idx.(s.elts), Temp.minIdx := Temp.idx)) &&
  (before Temp.idx.(s.elts) >= Temp.min => skip) } -- else nothing
```

Finally, we redefine `insertionStep` to use our new action.

```
action insertionStep [s: Sequence] { -- start at the current index,
  Temp.idx := Cnt.cur, Temp.min := Cnt.cur.(s.elts), Temp.minIdx := Cnt.cur;
  loop { -- run findMin over the suffix of the sequence,
    findMin[s]
  } && after Temp.idx = s.elts.lastIdx;
  (Cnt.cur := Cnt.cur + 1, -- and swap minimum element with the current one
   s.elts := s.elts ++((Temp.minIdx)→Cnt.cur.(s.elts))
   ++(Cnt.cur→Temp.minIdx.(s.elts))) }
```

Since our change was only incremental, we can show that the new sort refines the old one by verifying that repeating `findMin` yields the same element as our declarative `minIdx`.

```
assert findMinWorks {
  all s: Sequence | -- for all sequences ...
  always |
    (before (Temp.idx = Cnt.cur &&
            Temp.min = Cnt.cur.(s.elts) &&
            Temp.minIdx = Cnt.cur) &&
    loop { -- running findMin over the suffix of the sequence ...
      findMin[s]
    } && after Temp.idx = s.elts.lastIdx => -- finds the same element
    after minIdx[s.elts, Cnt.cur, Temp.minIdx] } -- as minIdx
```

Thus we can use the automated analysis our language extension affords us to support the stepwise refinement of a specification to executable, imperative code: our final version of `insertionSort` could easily be translated into an imperative programming language. Moreover, we have kept the analysis of our refinements tractable by performing it in a modular fashion, refining declarative specifications one at a time and analyzing the implementation of each separately.

5 Translation to Alloy

We now present the translation (Figure [II](#)) of our action language and associated operators into the first-order logic supported by the Alloy Analyzer.

$$\begin{aligned}
\llbracket o.f := e \rrbracket(t, t') &\hat{=} o.f.t' = e[.]t \wedge \\
&\quad \forall f' : (fields - f) \mid o.f'.t = o.f'.t' \wedge \\
&\quad \forall o' : (sigs - o), f' : fields \mid o'.f'.t = o'.f'.t' \wedge \\
&\quad t' = t.next \wedge t'.pc = fresh\ pc \\
\llbracket c_1 ; c_2 \rrbracket(t, t') &\hat{=} \exists t_1 : Time \mid \llbracket c_1 \rrbracket(t, t_1) \wedge \\
&\quad \llbracket c_2 \rrbracket(t_1, t') \\
\llbracket \text{loop } \{c\} \rrbracket(t, t') &\hat{=} \\
&\quad \exists begin, end : t.*next - t'.^next \mid \\
&\quad \llbracket c \rrbracket(t, begin) \wedge \llbracket c \rrbracket(end, t') \wedge \\
&\quad \forall mid, mid' : t.*next - end.^next \mid \\
&\quad \llbracket c \rrbracket(mid, mid') \Rightarrow \exists mid'' : mid'.^next \mid \\
&\quad \llbracket c \rrbracket(mid', mid'') \\
\llbracket act[a_1, \dots, a_n] \rrbracket(t, t') &\hat{=} act[a_1, \dots, a_n, t, t'] \\
\llbracket \text{before } \varphi \rrbracket(t, t') &\hat{=} \varphi[.]t \\
\llbracket \text{after } \varphi \rrbracket(t, t') &\hat{=} \varphi[.]t' \\
\llbracket \text{some } v : \tau \mid c \rrbracket C &\hat{=} \exists v : \tau \mid \llbracket c \rrbracket C \\
\llbracket c_1 \wedge c_2 \rrbracket C &\hat{=} \llbracket c_1 \rrbracket C \wedge \llbracket c_2 \rrbracket C \\
\llbracket c_1 \vee c_2 \rrbracket C &\hat{=} \llbracket c_1 \rrbracket C \vee \llbracket c_2 \rrbracket C \\
\llbracket c_1 \Rightarrow c_2 \rrbracket C &\hat{=} \llbracket c_1 \rrbracket C \Rightarrow \llbracket c_2 \rrbracket C \\
\llbracket \text{action name}[a_1, \dots, a_n] \{ Act \} \rrbracket &\hat{=} \\
\text{pred name}[a_1, \dots, a_n, t, t'] \{ \llbracket Act \rrbracket(t, t') \} &
\end{aligned}$$

$$\begin{aligned}
\llbracket \text{sometimes} \mid Act \rrbracket &\hat{=} \exists t, t' : Time \mid \llbracket Act \rrbracket(t, t') \\
\llbracket \text{always} \mid Act \rrbracket &\hat{=} \forall t, t' : Time \mid \llbracket Act \rrbracket(t, t')
\end{aligned}$$

Fig. 1. Rules for Translating the Action Language to Alloy

<pre> one sig Book { addr: dynamic (Name→lone Addr)} action add[n:Name, a:Addr] { Book.addr := Book.addr + (n→a)} assert addAdds { all n: Name, a: Addr always add[n,a] ⇒ after n.Book.addr = a} </pre>	<pre> one sig Book { addr: Name→lone Addr→Time} pred add[n:Name, a:Addr, t, t':Time] { t' = t.next && t'.pc = pc0 && all o:Book-Book o.addr.t = o.addr.t' && Book.addr.t' = Book.addr.t + (n→a) } assert addAdds { all n: Name, a: Addr all t, t': Time add[n, a, t, t'] ⇒ n.Book.addr.t' = a} </pre>
--	---

Fig. 2. Address Book Example (Left) and its Translation (Right)

5.1 Dynamic Idiom

Our translation uses two idioms that are common in the Alloy community for modelling dynamic systems. The first involves the addition of a “Time” column to each relation that represents local mutable state; the second involves the creation of a global execution trace using a total ordering on “Time” atoms.

Our translation adds a “Time” column to each **dynamic** field, and actions become predicates representing transitions from one time step to the next. We do not, however, enforce a global total ordering on time steps; instead, time steps are only partially ordered, allowing many traces to exist simultaneously.

In avoiding the single global trace, we gain the ability to compare executions, to run executions from within executions, and to run concurrent executions. The global trace does have performance and visualization benefits, however; fortunately, it is not difficult to infer that a particular analysis requires only a single trace, and then to enforce a total ordering on time steps. Our implementation performs this optimization, improving the performance and visualizability of many analyses considerably.

5.2 Translation

To translate our action language into a declarative specification following the trace-based idiom, we add a “Time” column to dynamic fields and thread a pair of variables through the action execution to represent the starting and ending time steps of that execution. We define a partial ordering on times using a field named “next:”

```
sig Time { next: lone Time }
```

We write the translation of action c into first-order logic in a *translation context* as $\llbracket c \rrbracket(t, t')$ (or $\llbracket c \rrbracket C$ when the parts of C are not needed separately) where the context contains start and end time steps t and t' . We also assume a global set *sigs* representing signatures with dynamic fields, and a global set of dynamic relations *fields*. We write $e[\cdot]t$ to denote the replacement of every reference to a dynamic relation $f \in \text{fields}$ in e by the relational join $f.t$; this operation represents the evaluation of e at time t . We give the complete translation in Figure 1, and an example translation in Figure 2.

Assignment simulates the process of updating an implicit store. The first generated conjunct updates the field $o.f$ with the value of e at time t . The second and third represent the *frame condition* that the transition updates only f at o : the second ensures that the other fields of o do not change, while the third ensures the same for objects other than o . The fourth conjunct specifies that an update takes exactly one time step, and the fifth constrains the final time step’s program counter.

Sequential composition is accomplished by existentially quantifying the time step connecting its two actions; loops are defined in terms of sequential composition. Action invocation passes the current time interval to the called action.

Named action definitions are translated into Alloy predicates with two extra arguments: the action’s starting and ending times. The action representing the

body is translated in the context of those times. A definition of an action is translated to a standard Alloy predicate, with the before and after times made explicit.

The translation of a “**sometimes**” formula existentially quantifies the beginning and ending states related by the result of translating the action in the body of the formula, while an “**always**” formula universally quantifies these states.

5.3 Semantic Implications

Our translation gives the language’s imperative constructs the same relational semantics given by Nelson [10] to Dijkstra’s original language of guarded commands [11]; these semantics also correspond to the standard operational semantics [12]. In addition, the relational semantics implies the existence of a corresponding semantics in terms of the weakest liberal precondition (namely, the *wlp*-semantics of Dijkstra’s guarded commands, also given by Nelson [10]). Our translation does not, however, correspond to a semantics in terms of weakest preconditions. The use of *wp*-semantics allows termination to be expressed; our language can only express partial correctness properties.

The property that an abstract action of only one step is refined by another action is directly expressible. The same property for actions of more than one step, however, is not expressible due to the known problem of unbounded universal quantifiers in Alloy [13].

6 Related Work

Our approach to modeling dynamic systems is similar to Carroll Morgan’s [14], the primary difference being that Morgan defines a programming language and then adds specification statements, while we begin with a specification language and extend it with commands. Like Morgan’s language, however, our command language supports the practice of refinement-based program development [6]. Our language is also similar to Butler Lampson’s system specification language Spec [15], which also provides both declarative and imperative constructs. The B Method [16] also provides the same imperative constructs that we present here, and gives them the same semantics. Abstract State Machines [17] represent another operational specification technique, but ASMs lack the declarative features of Alloy. None of these approaches currently support the Alloy Analyzer’s style of analysis.

Other traditional methods (such as Z [2] and VDM [3]) for specifying dynamic systems and analyzing those specifications center around the definitions of single-step operations, and do not offer a command language. Z does provide sequential composition, but no looping construct.

DynAlloy [45] has a very similar motivation to our work. It likewise extends Alloy, and offers operational constructs, but based on dynamic logic rather than relational commands. Unlike our extension, however, DynAlloy extends the semantics of Alloy, and translations are not intended to be human-readable.

Alchemy [18] also defines state transitions declaratively, but has the goal of compiling Alloy specifications into imperative implementations. Since it uses an idiom-based approach to state transitions, this work has prompted an exploration [19] of the properties that a declarative specification must have in order to correctly define a transition system. The specifications generated by our translation satisfy the necessary conditions by construction.

Some similar executable languages also exist: Crocopat [20] and RelView [21] both allow the definition and execution of relational programs. While these tools can execute commands over very large relations, they cannot perform the kind of exhaustive analysis that the Alloy Analyzer supports.

7 Conclusions and Future Work

We have extended the Alloy language with imperative operators. Our examples are indicative of our experience using the extension: dynamic models can be built statically and the dynamic elements added after verification of the static model. Moreover, the addition of sequential composition and looping constructs make models of dynamic systems more concise and easier to read.

We have also experimented with refinement-oriented development from specifications to implementations. The similarity of our language extension to the programming language used by Morgan [6], in addition to our ability to perform automated analysis on each refinement step, makes this strategy very attractive.

Finally, the move towards a mix of imperative and declarative constructs blurs the line between models and implementations. We have presented example models that can be translated easily into imperative implementations; given the simplicity of this translation, we plan to explore the possibility of automating it. Nondeterministic execution strategies like Prolog's backtracking search combined with a clever translation of Alloy's relational logic may allow for the execution of a large fragment of our extended Alloy language, making possible the a more direct execution that may perform well enough to be a practical implementation.

Acknowledgements

We are grateful to Jonathan Edwards, Eunsuk Kang, and Derek Rayside for their lively discussions and helpful comments. This research was funded in part by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software), and 0707612 (CRI: CRD – Development of Alloy Tools, Technology and Materials).

References

1. Jackson, D.: Software Abstractions: logic, language, and analysis. The MIT Press, Cambridge (2006)
2. Spivey, J.: The Z notation: a reference manual (1992)

3. Jones, C.: Systematic software development using VDM. Prentice Hall, New York (1990)
4. Frias, M., Galeotti, J., Pombo, C., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Proceedings of 27th International Conference on Software Engineering, 2005. ICSE 2005, pp. 442–450 (2005)
5. Frias, M., Pombo, C., Galeotti, J., Aguirre, N.: Efficient Analysis of DynAlloy Specifications (2007)
6. Morgan, C.: Programming from specifications (1990)
7. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the small scope hypothesis. In: Popl 2002: Proceedings of The 29th ACM Symposium on The Principles of Programming Languages (2002)
8. Jackson, D., Wing, J.: Lightweight formal methods. In: Saiedian, H. (ed.) Roundtable contribution to: An invitation to formal methods, vol. 29, pp. 16–30. IEEE Computer, Los Alamitos (1996)
9. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 294–308. Springer, Heidelberg (2008)
10. Nelson, G.: A generalization of Dijkstra’s calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11(4), 517–561 (1989)
11. Dijkstra, E.: A discipline of programming. Prentice Hall PTR, Upper Saddle River (1997)
12. Pierce, B.: Types and programming languages. The MIT Press, Cambridge (2002)
13. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. *ACM SIGSOFT Software Engineering Notes* 30(5), 216 (2005)
14. Morgan, C.: The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10(3), 403–419 (1988)
15. Lampson, B.: 6.826 class notes (2009), <http://web.mit.edu/6.826/www/notes/>
16. Abrial, J.: The B-book: assigning programs to meanings. Cambridge Univ. Pr., Cambridge (1996)
17. Börger, E., Stärk, R.: Abstract state machines: a method for high-level system design and analysis. Springer, Heidelberg (2003)
18. Krishnamurthi, S., Fisler, K., Dougherty, D., Yoo, D.: Alchemy: transmuting base alloy specifications into implementations. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 158–169. ACM, New York (2008)
19. Giannakopoulos, T., Dougherty, D., Fisler, K., Krishnamurthi, S.: Towards an Operational Semantics for Alloy. In: Proceedings of the 16th International Symposium on Formal Methods (2009) (to appear)
20. Beyer, D.: Relational programming with CrocoPat. In: Proceedings of the 28th International Conference on Software engineering, pp. 807–810. ACM, New York (2006)
21. Behnke, R., Berghammer, R., Meyer, E., Schneider, P.: RELVIEW-A system for calculating with relations and relational programming. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 318–321. Springer, Heidelberg (1998)

Towards Formalizing Network Architectural Descriptions^{*}

Joud Khoury, Chaouki T. Abdallah, and Gregory L. Heileman

ECE Department, MSC01 1100,
1 University of New Mexico, Albuquerque NM 87131
{jkhoury, chaouki, heileman}@ece.unm.edu

Abstract. Despite the rich literature on network architecture and communication system design, the current practice of describing architectures remains informal and idiosyncratic. Such practice has evolved based on idiomatic terminology and hence, it is failing to provide a formal framework for representing and for reasoning about network architectures. This state of affairs has led to the overloading of architectural terms, and to the emergence of a large body of network architecture proposals with no clear indication of their cross similarities, their compatibility points, their unique properties, and their architectural performance and soundness. Formalizing network architectural descriptions is therefore a timely contribution, and this paper presents a first step in that direction. The paper builds upon architectural style modeling concepts from the software engineering field, and applies them to the network architecture space. Our approach is presented through a case study detailing a formal model for a common class of network architectures. The model uses a simple declarative language based on relations and first-order logic.

1 Introduction

Despite the rich literature on network architecture and communication system design, the current practice of describing architectures remains informal and idiosyncratic. This was caused by the evolution of a semantically rich terminology that has been adopted by network architects over time. The terminology, despite being informal, reveals a lot of architectural information and has so far enabled efficient communication between architects. This scenario is very similar to the evolution of software architecture modeling in the context of software engineering [18]. This state of affairs has however, led to the overloading of architectural terms, and to the emergence of a large body of network architecture proposals with no clear understanding of their cross similarities, compatibility points, their unique properties, and architectural performance and soundness.

Several models for communication systems have been recently proposed, some of which are focused on particular communication aspects such as binding [21] or routing [11]. Others [13] are more general, and concern themselves with multiple communication aspects such as forwarding, naming, addressing. It is important to notice

^{*} The work presented in this paper is partially funded by the National Science Foundation NSF under the Future Internet Design (FIND) Grant CNS-0626380.

however, that the formal modeling and representation of network architectures is fundamentally different from that of communication systems. In fact, while the communication structure is necessary for defining and representing a network architecture, it is not sufficient. In addition to the communication structure, information and computation structures are building blocks that need to be properly understood within modern network architectures. Communication systems tend to share the same set of elements and are generally concerned with switching properties of networks and their communication and control primitives. On the other hand, network architectural descriptions are concerned with high-level architectural abstractions, their interactions, their structural and behavioral properties, and the constraints and invariants that define each architecture.

Towards formalizing network architectural descriptions, we utilize concepts relevant to architectural *style* modeling. Architecture *style* (or *pattern*) is a term commonly used in the software engineering field [18]. An architectural style is a family of network architectures that share a common representation vocabulary. Hence, while architectural *instances* specializing a particular style may vary in their particulars, their overall structure remains the same and obey the general style constraints. There are significant advantages associated with architectural style design. Those include a better overall system understandability by defining a precise common design vocabulary, the availability of design re-use among all instances of a class, architectural interoperability, and specialized analysis of a class of architectures by constraining the design space [15]. This paper presents a design methodology for formally describing and reasoning about network architectures and architectural styles. Our work is influenced by the work in [15]. The methodology is demonstrated by detailing a formal model for the FARA [8] family of network architectures. Our work provides a framework for network architects to formally group various architectures into a set of styles based on their common structural and behavioral characteristics, enabling researchers to better represent, analyze, reason about, and infer their important properties.

The rest of the paper is organized as follows: Section 2 presents the necessary background related to architectural styles and to the language *Alloy*. *Alloy* is a simple declarative language based on relations and first-order predicate logic and is the language that we shall use throughout the discussion for formal modeling and verification. Section 3 details our approach through a case study of the FARA [8] class of network architectures. We then present related work in section 4 before concluding in section 5.

2 Background

2.1 Architectural Styles: What and Why?

Software architectures are usually viewed as a set of interconnected elements that define the structure of a system. The elements are mainly components (computational and storage elements) and connectors (interactions among the components) [1]. An architectural *style* represents a family of architectures that share a common structural organization. Despite the different representations of a style [3, 5, 6, 18], it is typically composed of

¹ For example, in a client-server architecture description, one might model the client and server elements as components and an RPC communication protocol between them as a connector.

component/connector types, and a collection of constraints on how the types are combined. Associated with a style are a design vocabulary, an underlying computational model, and invariants [18].

Styles may be treated as stand-alone structures and may be related through inheritance, or composition. *Inheritance*, an extremely attractive property for describing architectural styles is the ability of a sub-style to extend one or more super-styles inheriting their structural properties, vocabulary, and constraints/invariants. *Composition* is another form relating multiple styles. The composed style is an aggregation of the vocabulary, structure, and constraints of the its constituent styles. Generally, the composed style introduces a new structure to relate the constituent styles together.

The advantages of *modeling* architectural styles are several (check [15]). First, given the abstraction level of an architectural style, it is generally hard to verify properties pertaining to the style or even to implement the style itself. A compact model then allows the verification of a style’s structural and behavioral properties over constrained instance sets without having to actually implement the style. This is an important step when applied prior to the actual instantiation of a complete architecture from the style. In other words, a formal model helps the transition from abstract style design to actual instantiations. Additionally, claims of compatible network architectures, whether those pertaining to general architectures, or to scoped architectures (such as naming, addressing, or routing) may then be logically verified. Finally, a formal model helps to classify the literature into related styles and architectures, and to succinctly illuminate the relations between them, whatever forms those may end up taking.

Modeling the structural properties of software architectural styles has generally been associated with the component/connector abstractions, and has utilized architectural description languages (ADLs) [3,5,15,6] for formal description. We believe that traditional component/connector abstractions associated with style modeling do not provide sufficient abstractions for network architects to work with². Therefore, we simply borrow the notion of “architectural style” without constraining ourselves to the component, connector, port, and role abstractions. Additionally, we choose to use the Alloy modeling language [12] rather than ADLs based on Alloy’s simplicity, its expressive power and ability to describe structural and behavioral aspects of an architectural style, and its ability to model desired specification properties that fit our needs (invariants, inheritance, and composition). Despite Alloy’s scalability concerns, we have found it useful to formally describe network architectures/styles because of the presumably small scope of abstractions involved in describing network architectural styles.

2.2 Alloy

Architectural design revolves around exploring the right abstractions, which are simple ideas expressed in some primitive form. Designing those abstractions requires a formal specification language that is intuitive, expressive, and at the same time avoids the intricacies of coding. Alloy [12] is one such language that we use to write our formalization of the FARA style [8] (to be detailed shortly). Alloy is a declarative language based on

² The component/connector abstractions might be sufficient when modeling communication systems, as may be deduced from the axiomatic model in [13].

relations and first-order predicate logic. A brief overview of Alloy's logic, language, and analysis follows. A complete reference is located elsewhere [12].

The Logic - At the core of Alloy is a relational logic that combines relational algebra with first-order predicate logic. Structures are composed of *atoms* and *relations*. Atoms represent typed, immutable structures that are uninterpreted and can be related through relations. A relation is a set of tuples each being an atom and can have arbitrary arity. Relations are combined with *operators* to form *expressions*. Some of the most common operators in Alloy are tabulated in Table 1.

Table 1. Operators in Alloy

Set operators	Relational operators	Logical operators
+ for union	\rightarrow for product	! for negation
- for difference	. for join	&& for conjunction
& for intersection	\sim for transpose	for disjunction
<i>in</i> for subset	$\hat{}$ for transitive closure	\Rightarrow for implication
= for equality	* for reflexive-transitive closure	, for alternative
		\Leftrightarrow for bi-implication

Constraints are formed of expressions and logical operators. Quantified constraints take the form $Q x : e|F$, where F is a constraint over x , e is an expression bounding x , and Q is a quantifier that can take values **all** (universal), **some** (existential), **no** (no values), and **lone** (at most one value). For example, **no** $x : e|F$ is true when no x in e satisfies F . When *Let* is used as in *Let* $a = b|F$, every occurrence of a in F is replaced by b . Declarations in Alloy take the form *relation-name* : *expression*, where expression is the bounding expression for the declared relation. For example, $r : Am \rightarrow nB$, where m and n are multiplicities, is a declaration saying that relation r is constrained to map each element of set A to n elements of set B , and each element of set B to m elements of set A .

The Language - In addition to the logic, Alloy provides some language constructs to help organize a *model*. A model in Alloy may consist of signatures (**sig**), facts (**fact**), functions (**fun**), predicates (**pred**), and assertions (**assert**).

Signature: A signature, declared with **sig**, introduces a basic type along with a collection of fields, their types and restrictions over their values. A signature can **extend** another signature inheriting its fields and constraints. An **abstract** signature has no elements except those belonging to its extensions. For example, if we write:

```
abstract sig A {
  f: set B
}{--constraints go here}
abstract sig B {}
sig A1 extends A {}
sig A2 extends A {}
```

one sig C{} --'one' means sig constrained to one element

we have declared three elements A , $A1$, and $A2$. Since $A1$ and $A2$ extend A , it follows that A **in** $A1 + A2$. Additionally, because A is abstract, it follows that $A = A1 + A2$ and $A1$ and $A2$ are disjoint sets that partition A . A declares a field f of type set B . This is

saying that for each element A , $A.f$ is a set of type B , i.e., the relation f is mapping from elements in A to sets of elements in B .

Facts, Predicates, Functions, and Assertions: A **fact** is simply a constraint that is assumed always to hold, and hence needs not be explicitly invoked. Facts usually describe global model constraints. The facts and the signature constraints thus constitute a complete set of structural constraints over the model. A function, declared with **fun**, is a named reusable expression that can be invoked within the model. A function takes zero or more arguments and returns either a true/false or a relational value. A predicate, declared with **pred**, is a named reusable constraint that can be invoked. A predicate takes zero or more arguments. An assertion, declared with **assert**, is a named constraint that is intended to follow from the model's facts. Assertions take no arguments and are usually checked by the Alloy Analyzer as discussed next.

The Analysis - The Alloy Analyzer (AA) [11] is an automated tool for analyzing models written in Alloy. Two kinds of analysis are enabled by AA, based on *commands*. The first is *simulation* (using **run** command) whereby the validity of a predicate or function is verified by showing a snapshot of the system for which the predicate is valid. The second analysis technique is *checking* (using **check** command), whereby an assertion is tested and AA tries to find a counterexample. This requires a finite *scope*, bounding the number of atom instances within the universe, within which AA looks for solutions. Given the undecidability of predicate logic, a finite scope is necessary to bound the space within which AA searches. Finding an instance to a predicate or a counterexample to an assertion guarantees the consistency of the constraint. However, failure to find such instance simply makes it inconsistent *within the scope*. The intuition is that subtle design bugs are likely to be detected even in small scopes.

3 Case Study

To motivate the usefulness of formal architectural modeling, and the expressiveness of the Alloy language, we represent the FARA [8] family of network architectures (or the FARA architectural style) using a formal model. Briefly, FARA [8] is an abstract network model in which the current Internet architecture is generalized and remodeled to enable clean separation of endpoint names from network addresses. Modeling FARA is an illustrative exercise in *architectural abstraction*, whereby a basis set of structural and behavioral components, assumptions, and constraints (invariants) that pertain to a desired class of architectures are extracted at the first stage of design to describe the general architectural model. Instantiations of the general model may then specialize it, obeying the general design assumptions and invariants. The authors of FARA had to implement a prototype of a FARA instantiation, M-FARA [8], in order to validate FARA's usefulness, and self-consistency. One of the goals of this section is to show how a formal model can be expressive and efficient in validating architectural design decisions, hoping to replace "validation through implementation" by "validation through formal modeling". Aside from providing a conceptual framework for reasoning about a class of architectures, a formal model of an architectural style transcends into a formal framework over which essential architectural design decisions can be modeled and verified.

Listing 1

```

abstract sig AID{}

abstract sig Entity{
  associations:Entity->Time,
  state:associations->one AID,
}{
no (this & associations.univ)
all t:Time, aid:AID |
  lone (state.aid).t
#state = #associations
}
abstract sig RIString {}

```

Listing 2

```

abstract sig FD{}
abstract sig Packet{
  dstFD: FD,
  replyFD: FD
}
abstract sig DPkt extends Packet{
  srcAID: AID,
  dstAID: AID
}
abstract sig SPkt extends Packet{
  ri: RIString
}

```

3.1 FARA Model

We hereby lay out a formal description of FARA's basic structural and behavioral components (static and dynamic properties) along with the constraints attached to the components and to the overall architectural style. The description accounts for dynamic behavior by explicitly including logical time steps to model evolution over time³.

Structural aspects: A formal definition of the *entity* and the *association* is given in Listing 1. An **Entity** is an abstract element that can have multiple concurrent **associations**. An association is a relation between two entities over time. Each entity maintains local immutable **state** per association, the association ID (**AID**). A particular association has exactly one AID, and AIDs are reusable over time. Several constraints are attached to the entity definition: the first constraint eliminates associations that connect an entity to itself. The second constraint is one of FARA's key assumptions, and it states that no two associations of an entity can have the same AID at any given time. The third structural consistency constraint forces each association to have state. An entity does not define a universal name since FARA does not require a global namespace⁴.

Listing 2 defines the Forwarding Directive (FD) and the packet abstractions. The **FD** encapsulates enough topological information to allow the substrate to deliver a packet to its intended destination. A generic packet, **Packet**, says nothing about the identity of the entities, and must indicate a destination forwarding directive (**dstFD**) that will be used by the communication substrate (to be defined shortly) to deliver the packet to a destination entity. A packet might also include a reply FD (**replyFD**) which the destination entity utilizes on the reverse path. FARA distinguishes between a packet that belongs to an association, a **DPkt**, and a setup packet, **SPkt**, that bootstraps an association. **DPkt** must specify the association state at both ends of an association, **srcAID** and **dstAID**, allowing the destination entity to correctly demultiplex the packet to its

³ Note however that analyzing the static properties of the architecture, simply requires dealing with a snapshot of the system at some timestep t , i.e., constraining the analysis scope of the *Time* signature to 1 instance.

⁴ Our approach to modeling an association as part of the entity's signature versus modeling it as a separate semantic element renders the dynamic constraints simpler and clearer.

association. SPkt includes a rendezvous information string, **ri** of type **RIString**, and does not include association state since the association is being bootstrapped.

Listing 3 defines the communication substrate component, **CommSubstrate**, representing a single global medium (the underlying operating systems and network) that is able to deliver packets on behalf of associations. The substrate assumes a basic connectionless delivery, **delivery**, without making any assumptions about the delivery function itself. A particular FARA instance, as we shall see later, provides the respective addressing, routing and forwarding mechanisms required for successful packet delivery. Supplied with an FD, the substrate delivers a packet all the way to its destination entity. The point-to-point assumption in FARA is modeled as part of the CommSubstrate constraints specifying that an FD can lead to a single entity at any time. So far, the model defines entities and associations independently of the mechanisms employed by the substrate for packet delivery. This acknowledges FARA's "red line" logical separation, whereby entities and associations operate above the line while the communication substrate operates below the line. Additionally, as a key assumption of FARA, no global address space is defined, in order to support a multitude of forwarding mechanisms.

Global style constraints, or simply invariants, are specified in Listing 4. The first consistency invariant constrains the association to be symmetric. Hence, entity A has an association with entity B if and only if the latter has an association with entity A. The second constraint eliminates dangling association states.

Having formally described the style, we may now proceed to validate some of its properties, specified as predicates and checked through the AA. For example, to check whether an entity might have overlapping state for distinct associations at some time, we define and run the predicate in Listing 5. AA does not find any instance of overlapping state within the simulated scope (7 Entity, Packet, FD, etc.; 15 AID; and 20 Time instances). This guarantees the correctness of the above claim only within the specified finite scope, and not in general. However, if inconsistent models can indeed be found, it is likely to find those within the specified scope.

Functional aspects: This section shows how functional aspects are formally specified at a high level of abstraction, leaving the details for architectural instances to specify.

The first function specified in FARA deals with the creation of associations. To model the system's dynamic behavior as a response to establishing and tearing down associations, we use Alloy traces to capture state transitions over time. Initially, at time t_0 , there are no associations. As presented in Listing 6, we consider two events that may change the system's state, the establishment or the tearing down of an association. The time instants **t1** and **t2** describe the state of the system before and after an operation is performed, respectively.

Listing 3

```
abstract one sig CommSubstrate{
  delivery: FD-> Entity -> Time
}{
  all t:Time | delivery.t in
    FD -> one Entity
}
```

Listing 4

```
fact Invariants{
  all t:Time | associations.t
    = ~ (associations.t)
    Time.(Entity.(Entity.state))
    = AID
}
```


Listing 5

```

pred showOverlapState {
  all t:Time |
  some disj e1,e2,e3:Entity |let
  w12=getAssociation[e1,e2,t],
  w13=getAssociation[e1,e3,t]
  |e1.w12=e1.w13 and some w12
}
run showOverlapState for 7
  but 15 AID, 20 Time

```

```

--Returns entity AIDs on both
--sides of the association
fun getAssociation
  [fst,snd:Entity,t:Time]
  :Entity->AID
{
  fst -> t.(snd.(fst.state))+
  snd -> t.(fst.(snd.state))
}

```

Listing 6

```

pred init[t:Time]{
  no associations.t
}
pred establishAssociation
[t1,t2:Time,fst,snd: Entity]{
  --Preconditions
  ---association does not exist
  let aset = {fst->snd+snd->fst}
  |no (aset & associations.t1)
  --Postconditions
  --no association change
  let aset={fst->snd+snd->fst} |
  {
  noAssociationStateChange[t1,t2]
  associations.t2 =
    associations.t1 + aset
  }
}

```

```

pred teardownAssociation
[t1,t2:Time,fst,snd: Entity]{
  --association exists
  let aset={fst->snd+snd->fst} |
  some (aset & associations.t1)
  --remove it
  let aset={fst->snd+snd->fst} |
  associations.t2 =
    associations.t1 - aset
}
--associations @t1 valid @t2
pred noAssociationStateChange
[t1,t2: Time] {
  all e1,e2:Entity |
  getAssociation[e1,e2,t1]
  in getAssociation[e1,e2,t2]
}

```

Given the possible state transitions of the system, we can form those into an execution trace by modeling the latter as a fact (Listing 7). Assertions may then be checked against the trace. An invalid assertion will demonstrate a trace showing how the assertion was violated. The Alloy analyzer may be used to show some execution trace of the system. For example, running the **showSomeState** assertion using AA, we obtain a counterexample showing a sample trace which, when projected over time, clearly demonstrates the state change resulting from creating or tearing down associations.

M-FARA: an Instantiation: M-FARA [8] is an instantiation of FARA that specifies its own addressing, forwarding, and FD management mechanisms. M-FARA is not a complete architecture, but it is specific enough to explore two points in the FARA design space: 1) location/identity separation, and 2) mobility. This section models M-FARA, particularly its addressing and forwarding mechanisms, using Alloy to demonstrate style specialization. First, a new module for M-FARA is created importing the FARA module just defined. Several new addressing and topological abstractions are introduced by the M-FARA module, as shown in Listing 8. M-FARA assumes multiple addressing realms, **Domains**, each having a **space** of unique addresses. A **subFD**

Listing 7

```

fact Traces {
  init [TO/first[]]
  all t:Time-TO/last[] |
    let t' = TO/next[t] |
      some disj e1,e2:Entity|
        establishAssociation[t,t',e1,e2]
        or teardownAssociation[t,t',e1,e2]
}

assert showSomeState{
  no e:Entity |
    #e.associations >=1
}
check showSomeState for
  4 but 7 AID, 7 Time,
  0 RIString, 0 Packet

```

represents a set of addresses that determine a local path within a domain. A domain has a static address space, **space**, and a dynamic forwarding mechanism, **forwarding**. The latter delivers a packet that is destined to some subFD to the entity that is bound to the respective subFD. Moreover, the topology assumed in M-FARA consists of a

Listing 8

```

sig subFD{}
abstract sig Domain {
  space: set subFD,
  forwarding:space->Entity->Time
}{
  --point2point forwarding
  all t:Time | forwarding.t in
    subFD -> lone Entity
}
--*No global address space*--
one sig MF_CommSubstrate
  extends CommSubstrate{
    domains: set Domain,
}

one sig Core extends Domain {}
sig PrivDomain extends Domain{
  upspace: some subFD,
  downspace: set subFD
}{
  upspace in space
  downspace in space
  no (upspace & downspace)
  -- up forwarding is implicit
  no ((forwarding.Time).Entity)
  & upspace
}

```

two-level domain hierarchy with a single distinguished central “**Core**” domain to which the private domains, **PrivDomains**, connect (Listing 8). The extended communication substrate, **MF_CommSubstrate**, may thus be viewed as the set of all domains including the core. Part of a private domain’s space, **upspace**, is used to reach the “core” domain. Similarly, part of the “core” domain’s space, **downspace**, is used by the core to reach the private domains. In this model, it is implicitly assumed that the forwarding function of every domain delivers subFDs belonging to upspace to the core. On the other hand, forwarding from the core down to the domain is explicitly specified in the domain’s forwarding function (hence subFDs belonging to downspace originate at the “core”).

Listing 9 defines the complete end-to-end FD in M-FARA, **MF_FD**. It consists of a tuple (*FDup*, *FDdown*) which the substrate can use to forward a packet from the source up to the “core” (**up**), and then from the “core” down to the destination entity (**down**). Regarding the entity abstraction, **MF_Entity**, M-FARA extends the entity definition with the local subFD to which the entity is bound, **fd**down and on which it is reachable.

Listing 9

```

sig MF_FD extends FD {
  up: lone subFD,
  down: one subFD
}
sig MF_Entity extends Entity{
  --canonical route
  fddown: subFD -> Time,
}

```

M-FARA does not specify whether an entity may be multi-homed (simultaneously bound to multiple domains) or not and our model does not restrict that either.

Some general structural constraints apply to the model and are expressed in Listing 10. No dangling subFDs or domains are allowed. Additionally, a subFD can belong to a single domain’s address space. Finally, the forwarding operation is local to the domain, i.e., an entry in the domain’s forwarding table means that the entity is bound to the domain. Modeling mobility in M-FARA is another interesting exercise, which we do not address in this paper. This task requires extending the FARA dynamical behavior, which so far includes establishing and tearing down associations, with a new mobility operation.

Abstract style properties: We have so far modeled an architectural style, FARA, and a particular instantiation of the style, M-FARA. The FARA style advertises a global theme of separating the entity from the communication substrate, and a set of style goals and properties. Despite the fact that the style leaves much of the functional details unspecified such as addressing/forwarding mechanisms, it is still essential for the style architect to model *super-properties*. A *super-property* is a *property of the style that is expressed in terms of abstract unspecified functionality*. In other words, the architect needs to confirm that any instantiation of the style that specifies the missing functionality will respect the super-properties are respected. In object-oriented programming, such design methodology is known as polymorphism. This section demonstrates a process for modeling style super-properties and checking those against the instantiation, by referring back to the FARA style and the M-FARA instantiation models.

As a first step, the style model includes the super-properties as facts, predicates, or assertions expressed in terms of unspecified functionality. The snippet in Listing 11 augments the previous FARA model with two new invariants (super-properties), expressed in Alloy as facts. The first fact is a “below the line” property. It states that delivery, which we have previously defined as part of the **CommSubstrate** in FARA, must be supported by the substrate’s addressing and forwarding mechanisms. In other

Listing 10

```

fact Invariants{
  --no dangling subFDs
  Domain.space = subFD
  --no Dangling Domains
  MF_CommSubstrate.domains
    = Domain
  --space is private
  all sf: subFD | lone space.sf
  --Forwarding local to domain
  all t:Time, d:Domain |
    let fwd = d.forwarding.t
    | all sfd:subFD,e:MF_Entity
    | {sfd ->e in fwd
      => sfd->t in e.fddown}
}

```

Listing 11

```

--Step1: super-property 1
fact {
  all t:Time | let
    delv=CommSubstrate.delivery.t
    | all fd:FD, e:Entity
    | {fd->e in delv =>
      this/isDeliverable[fd,e,t]}
}
--super-property 2
fact {
  all t:Time, e:Entity |
  let ea = e.associations.t
  | some fd:FD |
  this/ise2eDeliverable[e,ea,fd,t]
}
--*To be specified by Instance
pred isDeliverable
[fd:FD, e:Entity, t:Time]{}
pred ise2eDeliverable
[src,dst:Entity,dstfd:FD,t:Time]
{}

```

Listing 12

```

--Step 2
--Replicate facts from FARA
...
--*overriden function
pred isDeliverable
[dst:FD,e:Entity,t:Time]{
  let d_sfd=dst.down,
  d_dom = (getDomain[d_sfd])
  |d_sfd in d_dom.downspace
  and (d_sfd->e in
    d_dom.forwarding.t) }
--*overriden function
pred ise2eDeliverable
[src,dst:Entity,dfd:FD,t:Time]
{ some dfd.up and
  dfd.up in
  (getEntAttachments[src,t].univ)
  .upspace
  this/isDeliverable[dfd,dst,t]
}

```

words, if the substrate is able to deliver a message to an entity based on some destination FD, then the substrate’s forwarding mechanism must be able to deliver to that entity, hence satisfying **isDeliverable**. Again, note that **isDeliverable** is left unspecified by the style (in step 1), and is to be implemented by an instantiating architecture based on the forwarding mechanisms employed. The second fact is an end-to-end property (“above the line”) stating that an association exists and is valid only if packets are able to flow over the association from source to destination. In other words, there must exist some FD that satisfies **ise2eDeliverable**.

As a second step, the style instantiation extends the style model implementing the unspecified functionality. Super-properties are then enforced and checked against the instantiation to verify that the desired style goals are satisfied by all instantiations. To illustrate this step, the M-FARA model is augmented with the Alloy snippet in Listing 12, overriding the abstract functionality, **isDeliverable** and **ise2eDeliverable** ⁵. In M-FARA, **isDeliverable** or deliverability implies that: 1) some packet may be forwarded from the “core” down to destination’s domain, i.e., the *FDdown* part of the destination FD should belong to the **downspace** of the entity’s current domain, and 2) the domain’s forwarding function delivers to the entity given *FDdown*. End-to-end deliverability, in turn, requires two valid paths: one from the source entity’s domain up to the core, and another from the “core” down to the destination entity.

In the same fashion that facts about the style were replicated in the instantiation above, assertions and predicates may also be replicated. It is straightforward to add

⁵ In Alloy, the super-properties have to be replicated to the M-FARA model since Alloy does not directly support inheritance of a style or “module”.

assertions that verify the facts introduced above. For example, assertions dealing with mobility may easily be implemented.

Composition: Having already demonstrated inheritance and polymorphism in style modeling, we proceed to define and briefly overview (due to lack of space) *composition* as a means for composing separately defined modules or styles and checking for their compatibility. Let $S_i, i=1..n$, $n > 1$ be two or more styles, and let $P_i, i = 1..n$, be the global consistency constraints defined by S_i . The new composed style is denoted by $S = C(S_1, \dots, S_n)$ and contains the merged constraint set $\bigcup_{i=1}^n P_i$. S_i s are compatible styles *iff* the new consistency constraint $P = \&\&_{i=1}^n P_i$ is satisfied by S .

As an example, assume that a global-hierarchical addressing style, GHAR, is defined in which address spaces or domains are composed hierarchically (for example through customer-provider or peering relationships) with a distinguished core. The FARA style may then be composed with GHAR into a new style, say FARA-GH. An entity in FARA-GH extends the FARA entity and defines a global address field that is inherently hierarchical. Interestingly, the new FARA-GH architecture resembles the NIRA [20] routing architecture with the added conceptual clarity and design space partitioning.

4 Related Work

There are two broad areas of related work: network architecture and communication system modeling, and software system modeling.

The Internet architecture has been thoroughly studied over the past decade. The design principles of the DARPA Internet are clearly outlined in Clark's seminal paper [7] and other architectural design papers [9][10]. A methodology for designing and assessing evolvable network architectures based on *invariants* (or *fixed points*) is proposed in [4] which calls for considering invariants at an early design phase. Our formalization model inherently accounts for invariants as a part of the complete architectural description, and hence provides the architect with a clearer formal framework to work with invariants. As to communication system modeling, Karsten et al. [13] have proposed a general axiomatic basis to consistently model communication primitives such as forwarding, naming, and addressing for better expressing architectural invariants and formally proving properties about node reachability within any communication system. Our work is concerned with modeling general architectural descriptions rather than switching properties of networks. In [21], the author utilizes the Alloy modeling language to formally model identifier binding schemes which enables informed architectural design decisions for better supporting networking services.

In terms of modeling of software architectures, a lot of work has focused on formally describing those using Architecture Description Languages (ADL) [3][5][6]. Some of the common ADLs are the Acme ADL with the underlying first-order logic [15], extended WRIGHT [5], process ADL with the underlying process algebra [6], and π -ADL with the underlying π -calculus [17]. The Acme model in [15] utilizes Alloy and is a very relevant work to ours. Style inheritance and composition as well as verification of structural properties and compatibility checking are concepts demonstrated by the authors; however, their current model falls short of capturing the behavioral aspects

of the architectural style. Alternatively, the model in [6] explicitly involves topology specification (i.e. component/connector instances and their interconnections) as part of the architectural style description, which we believe is not an efficient approach considering the level of abstraction at hand. Finally, Alloy has been utilized within several modeling case studies that as described on the Alloy website [1]. We mention some of those that pertain to networking and that were useful for this work. Khurshid [14] used Alloy for modeling and correcting the architecture of the Intentional Naming System (INS). Jackson [11] has used it to model the Chord peer-to-peer lookup protocol. Some recent work by Narain [16] utilizes Alloy’s model finding techniques to find network configurations that satisfy a set of input requirements expressed with predicate logic.

5 Discussion, Future Work, and Conclusion

As previously stated, we have refrained from using the component/connector/interface abstractions for modeling network architectural styles. By surveying the network architecture literature, we have noticed that architects have different approaches to modeling abstractions. It is our belief that constraining them to component/connector/interface abstractions limits the expressiveness of the model and hence the innovation. It is additionally hard to anticipate whether and what modeling abstractions for networks will emerge in the future. The language we have utilized, Alloy, is generic and flexible enough to allow the architect to represent whatever abstractions she finds suitable. Despite the scalability concerns associated with constrained instances in Alloy, which does not represent a major limitation to us considering the high level of abstraction being modeled (and hence the presumable small instance sets required), the problem is currently being addressed in the literature (such as in [19]).

While this work has presented a first step towards formalizing network architectures and architectural styles, several research challenges remain to be solved and we address those as part of our current and future research. First, there needs to be a consensus regarding the most imminent styles that span the network architecture design space. Modern and future network architectures, as has been recently acknowledged [2], are being equipped with more intelligence, generally introducing information and computation structures that are manifested through increased in-network processing and storage. Extracting a complete, and disjoint set of network architectural styles may potentially frame the architectural problem and provide a formal framework for classifying, relating, and reasoning about architectures. Towards this end, we believe that a taxonomy of network architectures is a timely and essential contribution and represents a significant part of our current work.

To conclude, this paper has presented a methodology towards formally describing and modeling network architectures and architectural styles. Style inheritance, polymorphism, and composition were demonstrated on the FARA class of network architectures using the Alloy modeling language. Our work helps network architects and researchers, whereby architects are able to formally represent and group various architectural patterns into styles, while researchers are provided with a means to better understand, analyze, and reason about network architectures.

References

1. The alloy analyzer, <http://alloy.mit.edu/>
2. NSF Nets FIND initiative, <http://www.nets-find.net/>
3. Abowd, G.D., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.* 4(4), 319–364 (1995)
4. Ahlgren, B., Brunner, M., Eggert, L., Hancock, R., Schmid, S.: Invariants: a new design methodology for network architectures. In: *FDNA 2004: ACM Workshop on Future Directions in Network Architecture*, pp. 65–70. ACM Press, New York (2004)
5. Allen, R., Garlan, D.: A case study in architectural modelling: The aegis system. In: *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA, p. 6. IEEE Computer Society, Los Alamitos (1996)
6. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.* 11(4), 386–426 (2002)
7. Clark, D.: The design philosophy of the darpa internet protocols. In: *Proceedings of SIGCOMM 1988*, pp. 106–114. ACM Press, New York (1988)
8. Clark, D., Braden, R., Falk, A., Pingali, V.: Fara: reorganizing the addressing architecture. In: *FDNA 2003: ACM Workshop on Future Directions in Network Architecture*, pp. 313–321. ACM Press, New York (2003)
9. Clark, D.D., Sollins, K., Wroclawski, J., Faber, T.: Addressing reality: an architectural response to real-world demands on the evolving internet. In: *FDNA 2003: ACM Workshop on Future Directions in Network Architecture*, pp. 247–257. ACM Press, New York (2003)
10. Clark, D.D., Wroclawski, J., Sollins, K.R., Braden, R.: Tussle in cyberspace: defining tomorrow's internet. *IEEE/ACM Trans. Netw.* 13(3), 462–475 (2005)
11. Griffin, T.G., Sobrinho, J.L.: Metarouting. In: *Proceedings of SIGCOMM 2005*, pp. 1–12. ACM Press, New York (2005)
12. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge (2006)
13. Karsten, M., Keshav, S., Prasad, S., Beg, M.: An axiomatic basis for communication. In: *Proceedings of SIGCOMM 2007*, pp. 217–228. ACM Press, New York (2007)
14. Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: *ASE*, pp. 13–22 (2000)
15. Kim, J.S., Garlan, D.: Analyzing architectural styles with alloy. In: *ROSATEA 2006: Proceedings of the ISSSTA 2006 workshop on Role of software architecture for testing and analysis*, pp. 70–80. ACM Press, New York (2006)
16. Narain, S.: Network configuration management via model finding. In: *LISA 2005: Proceedings of the 19th conference on Large Installation System Administration Conference*, Berkeley, CA, USA, p. 15. USENIX Association (2005)
17. Oquendo, F.: A model-driven formal method for architecture-centric software engineering. *SIGSOFT Softw. Eng. Notes* 31(3), 1–13 (2006)
18. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs (1996)
19. Torlak, E., Jackson, D.: Kodkod: A relational model finder, pp. 632–647 (2007)
20. Yang, X.: Nira: a new internet routing architecture. In: *FDNA 2003: ACM Workshop on Future Directions in Network Architecture*, pp. 301–312. ACM Press, New York (2003)
21. Zave, P.: Compositional binding in network domains. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) *FM 2006*. LNCS, vol. 4085, pp. 332–347. Springer, Heidelberg (2006)

Lightweight Modeling of Java Virtual Machine Security Constraints

Mark C. Reynolds

Boston University
Department of Computer Science
markreyn@cs.bu.edu

Abstract. The Java programming language has been widely described as secure by design. Nevertheless, a number of serious security vulnerabilities have been discovered in Java, particularly in the component known as the Bytecode Verifier. This paper describes a method for representing Java security constraints using the Alloy modeling language. It further describes a system for performing a security analysis on any block of Java bytecodes by converting the bytes into relation initializers in Alloy. Any counterexamples found by the Alloy analyzer correspond directly to insecure code. Analysis of a real world malicious applet is given to demonstrate the efficacy of the approach. This type of analysis represents a significant departure from standard malware detection methods based on signatures or anomaly detection.

Keywords: Alloy, JVM, lightweight modeling, Java security.

1 Introduction

This paper will describe an analysis tool for verifying security constraints within Java bytecodes. This investigation was motivated by the continued appearance of malicious Java code that violates the security constraints imposed by the Java compiler, the Java Bytecode Verifier and the Java runtime. The analysis approach is based on the lightweight modeling language Alloy [1], [2]. This paper will describe the security verification approach taken by the Java Virtual Machine (JVM), and briefly enumerate some of the ways that it has been circumvented. A review of the top level goals of this work will then be presented, followed by a description of the design of the analysis tool and its implementation. Results will then be presented in detail using a real world example of malicious code: the BlackBox applet. Finally, a path toward future work will be described. The analysis tool has, in fact, proven to be a powerful approach to analyzing JVM security constraints. The approach of applying lightweight modeling as a means to check JVM security constraints appears to be a novel approach.

1.1 Background

The Java programming language has been touted as “secure by design” since its inception. However, attacks against Java security have been promulgated from

the earliest days of Java. Felten discovered several weaknesses in the Java security model almost immediately, and his work on Java [3] contains an extensive list of early exploits. The development of Java malware has continued unabated up to the present. The Common Vulnerabilities and Exposures project [4] lists numerous Java bugs that can lead to privilege escalation, sensitive data exfiltration, denial of service and other malicious outcomes. Of particular note is the BlackBox malicious Java applet [5], [6]. This applet exploits a number of Java security weaknesses, and was widely deployed, infecting thousands of machines. The BlackBox applet not only breaks out of the supposedly inescapable sandbox that the Java applet runtime imposes, it also manages to escalate its privilege to the highest possible level. The BlackBox applet can be easily customized to download any program to the infected machine and then run it. This applet is thus not only an exploit in itself; it is also a delivery vehicle for an arbitrary malicious payload. The BlackBox applet will be analyzed using the methodology presented in this paper.

In order to understand how these security failures come about, it is first necessary to briefly review the Java security model. Java security is enforced in three ways. The Java compiler has a large number of rules that it enforces in order to ensure that the syntax and semantics of the Java language are satisfied, but also to prohibit certain actions that are known to be associated with malicious code. For example, the Java compiler will refuse to compile any program that contains a method that makes use of an uninitialized variable. The output of the Java compiler is a binary file known as a classfile. In order for a Java application or applet to use the methods provided by a class, it must load the classfile that contains that class into the Java execution environment. Loading is accomplished by a Java classloader. Whenever a class is loaded the Java Bytecode Verifier is invoked. The Bytecode Verifier checks that the contents of the classfile conform to the classfile format and also verifies a large number of security constraints before it will allow the classloader to succeed. Finally, the Java runtime performs array bounds checking, runtime type conversion checking and a number of other tests.

Almost all Java exploits to date have used weaknesses in the Bytecode Verifier. The Bytecode Verifier's rules are described in great detail in the JVM specification [7]. The Bytecode Verifier uses a constraint based approach in performing its analysis. For example, it checks that all local variables are written before being read, that each instruction receives precisely the set of operands that it is expecting, that the stack has the same depth at each program point regardless of execution path used to reach that program point, and many other constraints.

Our approach uses Alloy to perform constraint analysis on Java bytecodes. It attempts to emulate the constraint checking that is ostensibly being performed by the Bytecode Verifier. In Alloy it is very easy to express constraints in terms of formulas involving relations, and therefore it has proven to be a rich environment for checking Java security constraints. Previous efforts have been made to apply formal methods to Java bytecodes [8], [9], [10], but these efforts have used a more heavyweight model checking approach that attempts to prove soundness,

as opposed to Alloy’s lightweight constraint based approach that converts assertions into Boolean formulas and then searches for satisfaction assignments or the existence of counterexamples.

1.2 Goals

This work described in this paper has three goals: (1) to provide an extensible framework for modeling security constraints imposed by the JVM’s Bytecode Verifier; (2) to provide a concrete model for meaningful, high value security constraints, and (3) to demonstrate that the analysis tool does check them correctly.

It would be straightforward to use Alloy to create a model for a specific block of Java bytecode. While this might serve as the demonstration of the applicability of Alloy to security analysis of the JVM, this would have little value in analyzing compliance with the JVM security constraints as a whole. Therefore, it is desirable to have an extensible model. In this context “extensible” means that the model must have the ability to be applied to any block of JVM code and to perform analysis on that code against a specified set of constraints. In the **Design** section it will be shown how this goal was realized.

Several of the security constraints imposed by the JVM have already been mentioned. In general, most constraints are independent of one another, although there are some functional overlaps, as will be demonstrated below. In order to prove the soundness of the basic concept, it was deemed prudent to select a realistic subset of the total set of JVM security constraints and begin with a simple model that would encompass that reduced subset of constraints. With the extensibility goal in mind, a general framework for code analysis was created such that adding additional constraints would involve only incremental modifications, and not a complete restructuring of the model code. The current implementation concretely models a small, but critical, set of security constraints. The work to date strongly suggests that the current implementation can be readily adapted to additional constraints.

2 Design

Alloy is a lightweight modeling language that uses first order logic. Alloy is capable of analyzing assertions for satisfiability and also for the existence of counterexamples. A key observation is that the security constraints imposed by the JVM can be modeled as invariants, and thus can be analyzed by the Alloy Analyzer. Alloy is not a proof system, so the failure to find a counterexample to a constraint is not a proof that that constraint is always satisfied, only that the constraint is satisfied within the search space specified. If a counterexample is found, however, that does indicate that the invariant has been violated, and the Alloy Analyzer conveniently provides a graphical representation of that counterexample.

The initial design problem was to find an “implementation” of the Alloy model that would capture the invariants of interest abstractly, independent of any actual JVM code, but would then permit the model to be run against any concrete

realization of such JVM code. Initial experimentation with Alloy suggested two possible approaches: automatically generate Alloy functions, facts or predicates based on the JVM code to be analyzed, or automatically generate Alloy statements that initialize relations based on the JVM code to be analyzed. In order to realize a classical code/data separation, it was decided to use the latter approach. Thus, the Alloy model would be realized as a template containing a fixed set of relations, functions, facts, predicates and assertions. This model would then be supplemented by relation initializers that would be derived from particular JVM code. In this approach, the template portion of the Alloy model would be completely independent of any choice of Java bytecodes, while the initializers would depend only weakly on the detailed implementation of the template. Specifically, the initializers being generated would only depend on the set of relations being initialized, and not on any specific way in which the constraints were realized in the model template. This decoupling between the “data” portion of the model and the “code” portion of the model is the means by which the stated extensibility goal has been achieved.

Further requirements analysis revealed that these two top level components, the model template and the initializers, could be further refined into four subcomponents: (1) the relation definitions; (2) the relation initializers; (3) the execution engine; and (4) the constraint assertions. The relation definitions, execution engine and constraint assertions are all part of the Alloy model template. The relation definitions are Alloy definitions of the top level signatures, which contain relations, as well as the definitions of the relations themselves. These relation definitions capture the static properties of individual JVM instructions, as well as capturing the JVM state as the execution engine executes. All other components of the Alloy model are logically dependent on the relation definitions.

The relation initializers are the initial values of the Alloy relations. They are generated from specific JVM code, and vary from one invocation of the model to the next. An initial design decision was made to capture JVM code at the method level. This, of course, is a trade off between performance and granularity. It is certainly possible to model multiple methods within a single model. However, the time that Alloy takes to analyze a particular model is strongly dependent on the number of (program execution) states, which, in turn is strongly dependent on the size of the relation initializers. As will be seen below, the actual Alloy model template is quite suited to analyzing code blocks within a method, and could be extended to handle multiple methods. Relation initializers need to be generated from specific Java methods. Therefore, there needs to be an automatic way of converting the Java bytecodes in a method into these relation initializers. To this end, a Java classfile parser was created to perform this conversion. The parser takes a Java classfile as input and produces an Alloy model fragment as output. When the model fragment is combined with the Alloy template, a complete Alloy model is produced, as is shown in Figure [II](#).

The relation definitions and their initializers form a static representation of a set of properties of the Java method being analyzed. In order to observe dynamic behavior, this static representation needed to be extended with model

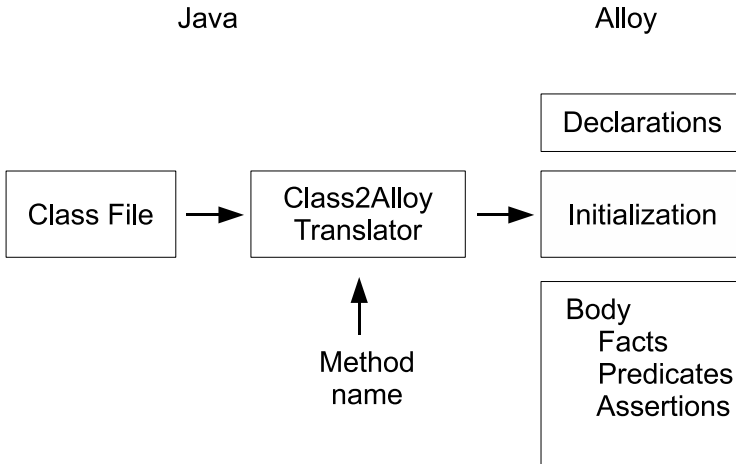


Fig. 1. Constructing a complete Alloy model using the classfile parser

actions that would mimic the execution of the JVM itself, at least to the extent that the JVM’s Bytecode Verifier synthetically executes method code in order to perform its constraint checking. Thus, an execution engine was needed. This execution engine would represent the flow of execution through the medium of stateful relations. Alloy’s “ordering” utility is used for representing this state. Execution could not be unbounded, of course, since Alloy only performs analysis over a finite set of states. It would have been possible to simply let Alloy “fall off the end” of execution, which is to say to allow the analyzer to perform an exhaustive analysis of all possible states in the state space. For both performance and structural reasons this was deemed to be an unacceptable solution. Therefore, the execution engine was designed such that certain JVM instructions are designated as terminal instructions. (Any type of *return* instruction would be terminal, for example.) The execution engine was then implemented to recognize this condition and act on it in such a way as to create no further unique states. Of course, this models the actual execution of the JVM itself. Certain instructions within a method are, in fact, terminal, in that they cause the method to be exited. One obvious question is the manner in which iterative constructs are handled by the execution engine. Would it provide better model fidelity to have the execution engine attempt to exactly mimic runtime execution, or would this lead to unacceptable performance penalties? In fact, the execution engine does not attempt to perform any branch prediction analysis in the model. The precise way in which this was handled, and its implications, will be explained in the **Implementation** section below.

Finally, the model must provide for a way in which each JVM security constraint is actually checked by Alloy. Formulating the security constraints as Alloy assertions proved to be straightforward once the model had been constructed to accurately reflect the static and dynamic properties of the method code.

3 Implementation

The implementation of the JVM security constraints analyzer will be described in three subsections. In the first subsection, the three components of the model template, namely the relation definitions, the execution engine, and the security constraint assertions, will be described. In the second subsection, the implementation of the Class2Alloy classfile parser which is used to generate the relation initializers will be discussed. In the third subsection a concrete example will be dissected, including a description of the parser invocation and subsequent model analysis. The example in question is a reduced form of the BlackBox applet.

3.1 Model Template

The model template employs two top level signatures, an *Instruction* signature and a *State* signature. The *Instruction* signature is made abstract in order that each of the individual instructions that make up a method can be defined as concrete, atomic extensions of this abstract signature. Intuitively, this is reasonable because the properties (relations) of instructions vary from instruction to instruction, but are still static for any particular instruction. For example, the length of a given instruction in bytes is fixed for all time once the instruction is specified, but obviously varies between instructions. The *State* signature is derived from Alloy's ordering utility, which predefines certain relations such as *first*, *next* and *last*. The *State* signature is dynamic, and the values of its relations are updated by the execution engine as it executes during analysis. The Alloy definition of these two signatures is shown below.

```
abstract sig Instruction {
  map:      Int,           // offset of this instruction in bytes
  term:    lone Int,      // is this a terminal instruction?
  r:       set Int,       // local variables read by this instruction
  w:       set Int,       // local variables written by this instruction
  ubt:     lone Int,      // unconditional branch targets
  cbt:     lone Int,      // conditional branch targets
  smod:    Int,           // bytes pushed/popped onto the stack
  len:     Int }         // byte length of this instruction

sig State {
  prog:      Instruction,
  readers:  set Int,
  writers:  set Int,
  depth:    Int }
```

An Alloy model is defined by its relations, so a careful description of each of the relations shown above will serve to illuminate the rest of the implementation. In the *Instruction* signature the *map* relation defines the byte offset of the instruction from the beginning of the method (or other block of code) being analyzed; it is an integer. The *term* relation is a set of integers that is either empty, or contains a single value. If the set is nonempty and contains the value 1, then

the instruction is a terminal instruction: it causes the execution engine to cease creating new states. The *r* and *w* relations model the sets of local variables read or written by the instruction, respectively. It is quite possible for an instruction to access more than one local variable, so these relations must be modeled as sets of integers. (The JVM itself also describes local variables in terms of integers.) The *ubt* relation names a possible unconditional branch target for the instruction. Most instructions do not have such a target, so the value of this relation is usually the empty set. An instruction can have at most one such target. If such a target exists, it is specified as a byte offset from the beginning of the method or code block, which is identical to the manner in which it is encoded in a classfile. The *cbt* relation names a possible conditional branch target. Conditional branch targets occur with conditional instructions. An unconditional branch target represents a transfer of control that must be executed, while a conditional branch target represents one that might be executed. Note that in the JVM it is possible for a conditional branch instruction to have multiple targets, but for simplicity this is not currently modeled. The *smod* relation models the number of bytes that the instruction modifies on the method stack. This can be a positive integer (item(s) are pushed onto the stack), a negative integer (item(s) are popped off the stack) or zero. Finally, the *len* relation models the length of the instruction in bytes. Note that *len* and *map* contain redundant information, in that it should always be the case that $next.map = current.map + current.len$. This redundancy was introduced deliberately as an additional way of validating the internal consistency of the model, as will be described shortly. The output of the translator acting on a simple method is shown below.

```

one sig startup, iload_1_1, bipush_2, if_icmpge_3, iload_1_4,
    iconst_5_5, imul_6, istore_2_7, goto_8, iload_1_9,
    istore_2_10, iload_2_11, ireturn_12 extends Instruction {}

fact maps { map = startup->(-1) + iload_1_1->0 + bipush_2->1 +
    if_icmpge_3->3 + iload_1_4->6 + iconst_5_5->7 + imul_6->8 +
    istore_2_7->9 + goto_8->10 + iload_1_9->13 +
    istore_2_10->14 + iload_2_11->15 + ireturn_12->16 }

fact lens { len = startup->1 + iload_1_1->1 + bipush_2->2 +
    if_icmpge_3->3 + iload_1_4->1 + iconst_5_5->1 + imul_6->1 +
    istore_2_7->1 + goto_8->3 + iload_1_9->1 + istore_2_10->1 +
    iload_2_11->1 + ireturn_12->1 }

fact rs { r = iload_1_1->1 + iload_1_4->1 + iload_1_9->1 +
    iload_2_11->2 }

fact ws { w = startup->0 + startup->1 + istore_2_7->2 +
    istore_2_10->2 }

fact ubts { ubt = goto_8->15 }

fact cbts { cbt = if_icmpge_3->13 }

```

```

fact terms { term = ireturn_12->1 }

fact smods { smod = startup->0 + iload_1_1->1 + bipush_2->1 +
  if_icmpge_3->(-2) + iload_1_4->1 + iconst_5_5->1 +
  imul_6->(-1) + istore_2_7->(-1) + goto_8->0 + iload_1_9->1 +
  istore_2_10->(-1) + iload_2_11->1 + ireturn_12->(-1) }

```

The *State* signature represents the dynamic execution state. Its *prog* relation models the current instruction being executed; its *readers* and *writers* relations model the current set of local variables that have been read or written up to the current program point, respectively, and its *depth* relation models the depth of the stack at the current program point. As the execution engine processes the instruction initializers, it effectively creates new *State* atoms representing the execution state after the effects of the current instruction have been applied.

The execution engine contains the Alloy code associated with *State* initialization, *State* sequencing, and execution termination. *State* initialization code is fixed within the model template. The *State* initialization code creates an initial state *s0*, sets the *readers* and *writers* relations of *s0* to be empty, sets the *depth* relation of *s0* to be 0, and sets the *prog* relation of *s0* to be the special *startup* instruction. Note that here is no actual JVM instruction named *startup*. However, when the JVM invokes a method it performs certain very specific startup actions (the method prologue) before the first instruction of that method is executed. The pseudo-instruction *startup* captures these actions. Specifically, when the JVM enters a method it will set the value of the local variable 0 to be Java's *this* object. If the method has arguments, these arguments are placed in local variables starting at index 1. Thus, the *startup* instruction will always have a nonempty value for its *w* relation; its *r* relation will be empty and the *depth* relation will be 0. The initializer for the *startup* instruction must be generated by the classfile parser. By convention, the *startup* instruction is located at a *map* value of -1 , and has length 1.

State transitions and also execution termination are handled by an Alloy fact known as *stateTransition*:

```

fact stateTransition {
  all s: State - ord/last |
    let s' = ord/next[s] |
      ( some t: s.prog.term | t = 1 ) =>
        sameState[s, s'] else
        nextState[s, s'] }

```

The model of execution is that the *nextState* predicate is executed for each nonterminal state. The *nextState* predicate is shown below. This predicate is responsible for updating the execution state relations (*readers*, *writers* and *depth*) and advancing the instruction state. This predicate calls the *nextInstruction* predicate, which updates the value of the current instruction for *s'*. It updates the *reader* and *writer* relations for the new state *s'* by calling predicates that take the unions of the corresponding *r* and *w* sets from the current instruction

$s.prog$ with the values of *readers* and *writers* from the current state s , respectively. Finally, it updates the *depth* relation for s' by adding the *smod* value of the current instruction to the *depth* in the current state s .

```

pred nextState[s, s': State] {
  nextInstruction[s.prog, s'.prog]
  nextReader[s.prog, s.readers, s'.readers]
  nextWriter[s.prog, s.writers, s'.writers]
  (s'.depth = add[s.depth, s.prog.smod]) }

```

The *nextInstruction* predicate calculates the next instruction for the state s' as follows. If the current instruction has an unconditional branch target, as indicated by the fact that the current instruction's *ubt* relation is not empty, then the unconditional branch is taken. The next instruction is the one whose *map* value (byte offset) matches the value of the *ubt* for the current instruction. This raises the interesting possibility that the JVM bytecodes might be sufficiently damaged that the *ubt* relation pointed to a *map* value that was not represented by any instruction, e.g. that the *ubt* pointed to the middle of an instruction, or outside the method entirely. This internal consistency constraint is checked by the *Terminates* predicate described below.

If there was no unconditional branch target, but there was a conditional branch target, then Alloy can choose to take that branch, or it can instead simply go to the next instruction by adding the current value of the *map* relation to the length of the current instruction. It will also perform this latter action in case there are no branches of either type. Note that the current model of exception handling treats an exception as a possible conditional branch for the entire range of instructions protected by a particular handler; the conditional branch target is the beginning of the handler code.

```

pred nextInstruction[from, to: Instruction] {
  some from.ubt =>
    ( to.map = from.ubt ) else
    ( ( to.map = add[from.map, from.len] ) ||
      some bt: from.cbt { to.map = bt } ) }

```

The Alloy model template captures some of the JVM security constraints checked by the Bytecode Verifier. The security constraints being checked are the local variable constraint, the stack depth invariance constraint, the stack guard constraint, the branch consistency constraint and the instruction length constraint. The local variable constraint states that no local variable can be read until it has first been written. The purpose of this constraint is to avoid accessing uninitialized local variables. The Java compiler enforces this constraint at the source code level for any variable (not just those that end up being stored in JVM local variables), and the Bytecode Verifier checks it at the classfile level. The stack depth invariance constraint states that the depth of the stack will always be the same at any program point, no matter how that program point was reached. The stack depth invariance constraint is one of the constraints that

is violated by the BlackBox applet, and will be discussed further below. The stack guard constraint is actually an amalgam of several closely related constraints. It states that the depth of the stack never becomes negative, and also that it should be zero on method entry and on any branch that leads to method exit, which is at any terminal instruction for the method. This latter constraint is a critical constraint for the JVM architecture. Unlike the architectures of many real machines, the JVM does not use the stack to pass parameters or return values; local variables are always used for both. Thus, the state of the stack (empty) should be the same on exit as on entry for every method. Each of the constraints corresponds to a single Alloy assertion. The branch consistency constraint and the instruction length constraint are different forms of the same consistency check, namely that neither normal flow of execution nor execution of any branch can put the JVM into a state in which it is not at an instruction boundary. Finally, there is also a special predicate that performs consistency checks on the model. The assertions and the predicate are:

```
assert LocalVar { all s: State | s.readers in s.writers }

assert StackDepth {
  all s, s': State | (s.prog.map = s'.prog.map) =>
    (s.depth = s'.depth) }

assert StackGTE { all s: State | gte[s.depth, 0] }

pred Terminates {
  some finalState: State | finalState.prog.term = 1 }
```

Note that each of the constraints is expressible in a single Alloy statement. The local variable constraint, *LocalVar*, asserts that for all states, the set of integers in the *readers* relation must be a subset of the set of integers in the *writers* relation. Since a state has a one-to-one correspondence with a program point (except for the special state that has *startup* as its instruction) this exactly expresses the local variable constraint. The stack depth invariance constraint, *StackDepth*, asserts that for any pair of states *s* and *s'* that have the same program point ($s.prog.map = s'.prog.map$) the depth of the stack must be the same ($s.depth = s'.depth$). The stack guard constraint, *StackGTE*, asserts that for all states the corresponding stack depth must be greater than or equal to zero. These constraints are positive constraints: if Alloy finds a counterexample this demonstrates that the constraint has been violated. A violation of the constraint then indicates that the corresponding JVM code does not conform to the classfile standard, and contains buggy or potentially malicious bytecode. It is worthwhile to observe, however, that the existence of nonconforming bytecode does not necessarily imply that the resulting code is exploitable.

The *Terminates* predicate bears closer examination, since it relates to the handling of looping constructs and also internal consistency checking. This predicate asserts that there is some state with an instruction that is terminal. In effect,

this predicate asserts that execution terminates for some set of branch choices. When faced with a conditional branch choice, Alloy will choose a possibility. Thus, if there is any path to a terminal instruction, it will be reached by some set of choices by Alloy (provided the search space is large enough). What conditions could cause this predicate to fail? One case would be the case of an unconditional branch whose target is an earlier program point corresponding to an unambiguous infinite loop. Another situation that would cause this predicate to fail is if the *map* and *len* relations are not internally consistent. Examination of the *nextInstruction* predicate shows that if there are no branches the instruction in the next state is calculated from the instruction in the current state by adding the length of the current instruction (the *len* relation) to the byte offset of the current instruction (the *map* relation). Alloy must then find a matching instruction whose offset (*map* relation) is equal to this sum. If no such instruction exists, then the *nextInstruction* predicate will return false and the *Terminates* predicate will never be satisfied. The *Terminates* predicate therefore also provides a test of the internal consistency of the *map* and *len* relations, and thus also indirectly checks the constraint that asserts that the JVM can never reach a program point that is not at an instruction boundary.

3.2 Class2Alloy Classfile Parser

The model template is not a complete Alloy model in that it does not encode any property information of an actual JVM method. That encoding is handled by the relation initializers, which must initialize all the instruction relations based on the bytecodes of a specified method. The initialization must also handle the creation of concrete signatures that extend the abstract *Instruction* signature. These concrete instruction signatures are based on exactly those instructions that are in the specified method.

A classfile parser, known as Class2Alloy, was written to generate these Alloy relation initializers given a Java classfile and also a method name. Class2Alloy was implemented in Java using the Byte Code Engineering Library, BCEL [11]. BCEL is an extremely powerful classfile analysis library that provides ready access to the instruction stream in Java classes. BCEL makes it straightforward to extract the requisite properties for each instruction under consideration.

Class2Alloy is implemented in two Java files, Class2Alloy.java and AlloyString.java. Class2Alloy.java contains the main analysis routines, while AlloyString.java is a utility class that handles the specific Alloy syntax needed to generate syntactically correct relation initializers. The operation of the parser is as follows. The *main* method receives three arguments: the name of a classfile, which must be in the classpath, the name of a method, and the name of an output file. The *main* method creates a Class2Alloy instance; the Class2Alloy constructor creates a set of empty AlloyStrings, one for each relation to be initialized, along with an empty AlloyString that will hold the instruction signature information. BCEL is then used to load the classfile, enumerate its methods, and search for the named method in the array of methods; on success a BCEL *Method* object

is obtained. `Class2Alloy` then parses this *Method* object to obtain a list of instructions contained within the method. For each instruction, it then queries that instruction for those properties that need to be initialized in the Alloy model, namely its byte offset from the beginning of the method, its byte length, the sets of local variables that it reads or writes, the set of possible conditional or unconditional branches that it can take, and also the number of bytes that it adds or removes from the stack. Once the instruction analysis is complete, each `AlloyString` prints itself to the output file. The `AlloyString` class handles the details of generating syntactically correct Alloy output for each of the relation initializers, as well as generating the appropriate extension signatures for each instruction in the method being analyzed.

Once this output file is combined with the model template, a complete model specialized for the method under analysis is obtained. The Alloy analyzer is then run on that model, and each of the constraint assertions, as well as the *Terminates* consistency predicate, is invoked to determine the presence of counterexamples or a failure to converge to a terminal state.

3.3 Analysis of the BlackBox Applet

The `BlackBox` applet is a malicious applet that breaks out of the applet execution sandbox, elevates its privilege level to the maximum possible value, and then downloads (and optionally executes) a completely arbitrary payload. The `BlackBox` applet uses a variety of exploitation techniques in order to achieve its goals. A complete description of the workings of this applet is beyond the scope of this paper; instead a reduced version will be described. It is very important to note that both the reduced version and the complete version of `BlackBox` are detected by the technique described herein.

The Java Virtual Machine loads classes using a series of helper classes known as classloaders. The classloader class responsible for loading classes across the network is the `URLClassLoader` class. `URLClassLoader` not only verifies classfile syntax, it works closely with the `SecurityManager` class to check for permitted or forbidden operations. As one might readily imagine, most of the methods and members of `URLClassLoader` are either protected or private. If it were possible to define a class (call it *myUCL*) that had the same methods and members as `URLClassLoader`, but which permitted all operations and did not consult the `SecurityManager`, one could then load and instantiate an arbitrary network class. This type of exploit is known as a type confusion exploit: an object of one class (`URLClassLoader`) is replaced by an object of another class (*myUCL*) without triggering a type coercion exception.

One way to cause type confusion to occur is to violate the stack depth invariance constraint. Suppose that an object of type *myUCL* is placed on the stack, followed by an object of type `URLClassLoader`. Suppose further than in the normal flow of execution the `URLClassLoader` object will be popped off the stack and loaded into a local variable of that type. If the flow of control can be modified such that the stack depth invariance constraint is violated, and such that the `URLClassLoader` object is popped off the stack while leaving the JVM

state unmodified, then a subsequent stack operation will pop the *myUCL* object off the stack and treat it as if it were an object of type *URLClassLoader*. One approach for doing this was through the use of malicious exception handling code. When an exception is thrown in Java, an exception object is placed on the stack so that the exception handler code may access it. After the handler completes, the state of the stack is supposed to be restored to the state it had just before the exception was thrown. If it can be arranged that the malicious exception code actually modifies the stack such that two objects are popped when the handler exits, the stack depth invariance constraint will be violated. The net effect in the actual BlackBox applet is that an object of type *myUCL* is substituted for an actual *URLClassLoader* object; this object is then used to load malicious code over the network and execute it. (As stated above, the actual process is significantly more complicated.)

More than a hundred benign applets were subjected to analysis using the Alloy analyzer; no stack depth invariance constraint violations were found. However, when the bytecodes from the full or reduced version of BlackBox were analyzed, a violation of the stack guard invariance constraint was detected. The approximate time to run the Alloy analyzer and find this counterexample was three minutes. Detailed security analysis with a Java disassembler (such as [12]) then revealed that a type confusion attack was being launched by this applet. Note that the applet itself could not have been compiled directly from Java; the malicious portions, in particular those portions handling the type confusion attack, had to have been constructed using a Java assembler, such as Jasmin [13].

3.4 Future Work

There are several areas in which the JVM security analysis approach described in this paper can be extended and improved. The most obvious, and certainly the most important, is to add constraint checking for additional constraints. The opcode argument constraint, which states that each JVM instruction is invoked with the correct number of type conforming arguments, is of particular importance. In addition, extending the current consistency checking of the *map* and *len* properties is also a worthwhile step, since the current model does not distinguish the two possible cases in which the *Terminates* predicate fails to converge, namely infinite loops versus an inconsistent set of *map* and *len* relation values. The latter should be checked explicitly.

The current model does not completely handle exceptions. In particular, only a single exception block per method is currently modeled, while actual bytecode can employ multiple (nested) exception blocks. Adding full exception handling to the model has high priority. This is an ongoing area of research.

4 Conclusion

This paper has demonstrated that Alloy is an extremely powerful tool for performing security constraint analysis on Java bytecodes. Even at this stage of

development, meaningful results have been obtained. Extensions to this work are ongoing, with the goal of increasing the scope of constraint checking and further refining and improving the analysis process. Extensions to other languages are also in work.

Acknowledgments. The author wishes to extend special thanks to Assaf Kfoury of the Computer Science Department of Boston University for suggesting this line of inquiry, for his valuable input, and for his continued support and encouragement.

References

1. Alloy website, <http://alloy.mit.edu>
2. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)
3. McGraw, G., Felten, E.: Securing Java: Getting Down to Business with Mobile Code, 2nd edn. Wiley, New York (1999)
4. Common Vulnerabilities and Exposures, <http://cve.mitre.org>
5. BlackBox Security Advisory, <http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=36725>
6. Java and Java Virtual Machine security vulnerabilities and their exploitation techniques, <http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-1sd.pdf>
7. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification Second Edition. Addison Wesley, Boston (2003)
8. Xu, H.: Java Security Model and Bytecode Verification, <http://www.cis.umassd.edu/~hXu/Papers/UIC/JavaSecurity.PDF>
9. Posegga, J., Vogt, H.: Java bytecode verification using model checking, <http://eprints.kfupm.edu.sa/47269>
10. Leroy, X.: Java Bytecode Verification: An Overview. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 265–285. Springer, Heidelberg (2001)
11. Jakarta BCEL, <http://jakarta.apache.org/bcel>
12. DJ disassembler, <http://members.fortunecity.com/neshkov/dj.html>
13. Jasmin assembler, <http://jasmin.sourceforge.net>

Alloy+HotCore: A Fast Approximation to Unsat Core

Nicolás D’Ippolito, Marcelo F. Frias, Juan P. Galeotti, Esteban Lanzarotti, and Sergio Mera

Departamento de Computación, UBA, Argentina,
{ndippolito,mfrias,jgaleotti,elanzarotti,smera}@dc.uba.ar

Abstract. Identifying a minimal unsatisfiable core in an Alloy model proved to be a very useful feature in many scenarios. We extend this concept to *hot core*, an approximation to unsat core that enables the user to obtain valuable feedback when the Alloy’s sat-solving process is abruptly interrupted. We present some use cases that exemplify this new feature and explain the applied heuristics. The NP-completeness nature of the verification problem makes hot core specially appealing, since it is quite frequent for users of the Alloy Analyzer to stop the analysis when some time threshold is exceeded. We provide experimental results showing very promising outcomes supporting our proposal.

1 Introduction

Alloy [1] is a relational modeling language. Its simplicity, object-oriented flavor and automated analysis support have made this formal language appealing to a growing audience. The Alloy Analyzer wisely transforms Alloy models in which domains are bounded to a fixed scope, into a propositional formula that is later fed to a selected SAT-solver (such as MiniSat [2] or SAT4J). Then, given an assertion to be verified in the model, the Alloy Analyzer attempts to produce a counterexample violating the assertion. If no such counterexample is found, it concludes that the analyzed property holds in the model within the given scopes.

A useful feature of Alloy Analyzer is the *unsat core* highlighting [3]. This feature allows an user to see a (possibly minimal) subset of the model constraints from which the assertion follows. As mentioned in [3], this information helps to mitigate a variety of modeling problems, such as overconstraining the model, using a weak assertion, or setting an analysis scope that is too small.

As the problem of knowing if a given propositional formula is satisfiable is NP-complete [4], it is quite frequent to exceed the time the user is willing to spend in analysis even for small scopes. That is why the Alloy Analyzer supports interrupting the SAT-solving process. The drawback is that when the user aborts the current analysis, he or she obtains no feedback on the verification process. This occurs even when the search space is almost covered.

Due to the analysis interruption, no conclusive answer is given by the SAT-solver on the satisfiability of the propositional formula. Nevertheless, the SAT-solver has spent a (possibly large) amount of time analyzing the input formula,

and has gained valuable knowledge about it. Our main contribution consists on using this knowledge to identify a collection of “problematic” or “hard” clauses, and present them to the user as a potentially unsatisfiable set of constraints. We call this set the *hot core* of the analysis. Intuitively, a problematic set of clauses is a set that adds significant computational cost to the SAT-solving process. Since the worst case scenario for a SAT-solver is to work with an unsatisfiable formula, a computationally hard set of clauses seems a reasonable symptom of unsatisfiability.

HotCore, our proposed extension to Alloy, profiles the SAT-solver execution in order to gather the required information as the solving process takes place. When the user interrupts the analysis, HotCore identifies a set of the most problematic clauses, applies the inverse translation (from propositional formulas to Alloy model) to this set, and shows the highlighted constraints to the user. Since Alloy users are used to the unsat core highlighting, HotCore uses this presentation style.

To the best of our knowledge, the idea of profiling an interrupted SAT-solving process in order to provide feedback to the user does not seem to be explored. We could only find some weakly related work in the direction of visualizing a DPLL run [5], and in the line of incomplete solving methods, like GSAT, Walksat and some applications [6,7,8,9].

The structure of this article is as follows. In Section 2 we present a methodological approach to HotCore through several user scenarios. In Section 3 we provide a brief theoretical background to SAT-solving, and we give a detailed explanation of the heuristics we used to identify a hot core. In Section 4 we provide a representative collection of chosen Alloy problems, showing a very promising behavior of HotCore. As we will see, on average HotCore covers above 90% of the unsat core within one fifth of the total solving time. Finally, in Section 5 we draw some final conclusions and suggest future lines of work.

2 Motivation

In this section we will present some of the benefits of identifying a hot core, and show how HotCore helps to solve some limitations of the current distribution of the Alloy Analyzer. In order to make it more amenable to the reader, we will do it through a running example. We will assume that the reader is familiar with Alloy’s syntax and semantics. Refer to [1] for a more detailed description of Alloy. We present below signature definitions for doubly linked list-like structures.

```

one sig null {
}
sig Object {
  owner: one Object + null
}
sig Node extends Object {
  next: one Node+null,
  previous: one Node+null
}

sig List extends Object {
  head: one Node+null,
  last: one Node+null
}

```

Signatures and fields are interpreted as sets of atoms and relations between atoms, respectively. Singleton signature `null` represents the *null* value. Signature `Object` includes all possible objects. A field `owner` marks the atom's owner (or `null` if no owner exists). Atoms in signature `Node` form a subset of the atoms in signature `Object`, and have two extra fields (`next` and `previous`) which are intended for storing the next and the previous nodes respectively (or the *null* value in case no next/previous node exists). Similarly, atoms in signature `List` have two extra fields (`head` and `last`). These fields are references to the first node and last node respectively. The following facts constrain atoms and relations.

```

fact head_is_null_iff_last_is_null {
  all list: List | list.head=null <=> list.last=null
}
fact head_last_nullity {
  all list: List | {
    list.head!=null implies list.head.previous=null
    list.last!=null implies list.last.next=null
  }
}
fact next_prev_symm {
  all node: Node | {
    node.next!=null implies node.next.previous=node
    node.previous!=null implies node.previous.next=node
  }
}
fact no_node_sharing {
  no l1, l2: List |
  some n: Node | {
    n in l1.head.*next
    n in l2.head.*next
  }
}

```

These facts state that: 1) head points to *null* if and only if last points to *null*, 2) the first node (respectively the last node) previous field (respectively next field) points to *null*, 3) symmetry between next and previous fields is preserved, and 4) no *Node* atom is shared between two lists. We will refer to these axioms as *list structure facts*.

Next, the model is completed with facts to characterize ownership relations among node and list objects. These facts were inspired by the Spec# programming methodology [10].

```

fact owner_of_head {
  all list: List | list.head!=null implies list.head.owner=list
}
fact owner_of_last {
  all list: List | list.last!=null implies list.last.owner=list
}
fact owner_of_next {
  all list: List | all node: list.head.*next - null |
  node.next!=null implies node.next.owner=node.owner
}
fact owner_of_prev {
  all list: List | all node: list.last.*previous - null |
  node.previous!=null implies node.previous.owner=node.owner
}

```

We will refer to these newly added axioms as *ownership facts*. Now, let us consider the scenario where an Alloy user writes an assertion to verify that every node has its container as owner:

```

assert container_is_owner {
  all list: List | all n: list.head.*next - null | n.owner = list
}

```

Running the analyzer with the command `check container_is_owner for 3`, it yields as result that the assertion has no counterexamples within that scope.

As it was previously mentioned, some SAT-solvers provide the feature of *unsat core extraction*. Exploiting this facility, the Analyzer manages to highlight the following subset of the Alloy model.

```

sig List extends Object {
  head: one Node+null,
  ...
}
sig Node extends Object {
  next: one Node+null,
  ...
}

fact next_prev_symm {
  all node: Node | {
    node.next!=null implies node.next.prev=node
    node.prev!=null implies node.prev.next=node }
}

fact no_node_sharing {
  no l1,l2: List | some n: Node | {
    n in l1.head.*next
    n in l2.head.*next }
}

assert container_is_owner {
  all list: List | all n: list.head.*next - null | n.owner = list
}

```

These constraints are marked as a (possibly minimal) cause of unsatisfiability. As no ownership fact was highlighted, the user could conclude that they were not needed to prove the validity of the assertion within the scope. Increasing the analysis scope up to 10 yields the same result. Although the scope has grown, the unsat core remains the same.

From the user point of view, ownership rules must play an important part in constraining the `owner` field. Therefore, he or she may find it hard to believe that the validity of the property does not depend on those axioms.

A further inspection of the highlighted constraints exposes a subtle error in the fact `no_node_sharing`. As we can see, the fact states that *no two lists share the same node*. This is stronger than *no two distinct lists share the same node*.

An overconstrained model represents one of the most common modeling mistakes using Alloy. In the absence of a counterexample, the unsat core highlighting helps the user to validate that the assertion does not follow trivially from the model. A revised definition of fact `no_node_sharing` is given below:

```

fact no_node_sharing {
  no disj l1,l2: List | some n: Node | {
    n in l1.head.*next
    n in l2.head.*next }
}

```

Once again, the user checks the assertion `container_is_owner`, but in this case considering the revised model. The Analyzer finds no counterexample within the scope of 3. But, in this case, the Analyzer highlights several ownership facts as part of the unsat core constrains. This means that ownership facts were used during the analysis to prove the assertion is valid. From the user's point of view, this is closer to the expected behavior of the Analyzer.

As we have seen, an Alloy user could profit enormously from inspecting a given unsat core. We now present a collection of use cases that exhibits some practical applications for HotCore.

Scenario 1: unsat core approximation: Let us consider the following scenario. In order to gain more confidence about the assertion validity, the user starts analyzing the faulty model using a scope of 10 instead of a scope of 3.

The analysis carried out by the Alloy Analyzer relies on solving a SAT problem. A larger scope leads to a formula with more variables to be solved. Since the analysis time grows (in the worst case) exponentially on the number of variables, analysis may take much longer.

As analysis takes a lot more to conclude the assertion validity, it is not rare that the Alloy user may interrupt the analysis once a certain time bound is reached. This bound may range from seconds to hours. Assume that he or she decides to abort the analysis after 5 minutes. Since the Alloy Analyzer failed to exhaust the search space and to produce a counterexample within this time bound, HotCore highlights some constraints as the hot core of the analysis. Among those highlighted constraints is the overconstrained fact `no_node_sharing`.

As the user interprets this hot core as an approximation to a potential unsat core, he or she inspects more deeply the constraints. During this task, the user detects the flaw in fact `no_node_sharing`, and fixes the constraint. Notice that it was not necessary neither to decrease the analysis scope nor wait until an unsat core was identified.

Scenario 2: constraint optimization: Another possible use of HotCore consists on optimizing a given formula. It is well known that equivalent Alloy formulas may produce different but equivalent CNF formulas. Given two equivalent CNF formulas, the SAT-solving time may vary enormously depending on factors such as clause ordering, number of variables and average length of clauses, just to name a few of them.

In large and complex Alloy models, it is not always a feasible task to re-write every constraint in a more compact, friendly to the Alloy Analyzer, equivalent constraint. But, knowing those constraints that played a key role during analysis, an experienced Alloy user may re-write those constraints in order to reduce the analysis time.

Going back to our revised model, the Alloy Analyzer still exceeds the 5 minutes bound to prove the validity of the assertion `container_is_owner` in a scope of 10. Once again, the user may interrupt the analysis once the time bound is exceeded. As the model was weakened, more constraints are identified as members of the hot core. Not only several ownership facts, but also the fact `next_prev_symm` is marked.

An experienced user may replace fact `next_prev_symm` with a relational formula with no quantifiers, defining `next:>Node=~(previous:>Node)`. It is easy to see that both facts are equivalent. Rewriting this fact leads to a CNF formula with less variables and clauses. Upon this model re-writing, the Alloy Analyzer is now able to end the validity analysis in less than 5 minutes. Observe that this use of HotCore can be always applied, independently of the assertion validity.

Scenario 3: gaining more confidence in larger scopes: Although the *small scope hypothesis* [11] argues that a high portion of counterexamples can be found by analyzing an assertion within small scopes, the usual practice shows that most Alloy users tend to try larger scopes as long as the time resources are not

exhausted. From the user’s perspective, a larger scope means a better confidence in the validity of an assertion.

As it was mentioned before, the analysis time grows as the scope increases. In fact, analysis time may scale from seconds to hours with a slight increase of scope. Because of this, although the Alloy Analyzer may have proved that an assertion is valid within a given scope, it may be infeasible to prove the same assertion for a larger scope. Notice that for smaller scopes the Alloy Analyzer will return an unsat core, while for the larger scope, our extension will identify a hot core. If a larger-scope hot core matches a smaller-scope unsat core, an Alloy user can conclude that most of the analysis time was spent dealing with a set of constraints which happen to be unsatisfiable within the smaller scope. This may be used as evidence that the assertion follows from the same premises within the larger scope. Of course this cannot be considered a proof of validity of the assertion for the larger scope. This is just a heuristic to gain more confidence when the Analyzer fails to return a conclusive result within the time bound.

As an example, let us consider the case where the user starts the analysis of the assertion `container_is_owner` in a scope of 20 instead of 10. If the user interrupts the analysis after 10 minutes, he or she can confirm that the 20-scope hot core matches the 10-scope unsat core. Under the previous premises, the user has more confidence on the assertion validity. Observe that, without having the HotCore extension, the user has no elements to produce a hypothesis on the assertion validity when the analysis is interrupted.

3 Finding Minimal Sets of Hot Clauses

In this section we describe the theory behind the procedure we use to compute the hot core. Let’s recall first the general schema used by the Alloy Analyzer to verify an assertion. Given a fixed bound in the size of the models to explore (which is also known as the *scope* of the analysis), the Alloy Analyzer verifies whether a given assertion follows from a specification by translating the problem to a set of propositional clauses S . Then the tool runs a SAT-solving process over that set: if the SAT-solver determines that S is unsatisfiable, that means that the property follows from the specification within the specified scope. Otherwise, a satisfiable assignment for S is found, and a counterexample for the assertion is constructed using that assignment. Finally, in the case S is unsatisfiable, a minimal unsatisfiable core $U \subseteq S$ is computed, and then U is mapped back to the associated constraints. The unsat core is then presented to the user highlighting the appropriate Alloy formulas.

We first give a global description of the process to compute the hot core. The rest of the section will be devoted to explain it in detail. The general idea follows the same steps used to extract the unsat core. The SAT-solving process starts with an initial clause database C given by S , and during the execution this set of clauses is augmented with new clauses, all of them being consequences of S . Since the user may interrupt the process at any time, we monitor the SAT-solving clause database keeping track of the clauses that are “hard” or

“problematic” to solve. We will give a more precise definition of this condition later. When the SAT-solving process is interrupted, we take a set $H \subseteq C$ of the most problematic clauses and we find a minimal subset $H' \subseteq S$ such that all the clauses in H are consequences of H' . The set H' represents a minimal subset of S which is potentially unsatisfiable. We then proceed as it is done with the unsatisfiable core U , applying the inverse translation over H' , (from propositional formulas to Alloy) to identify the hot core.

To determine which are the problematic clauses we take advantage of the heuristics already implemented in the SAT-solver. As we said before, we used Minisat to implement the hot core extraction, but this feature can be extended to any DPLL based SAT-solver that provides a way to measure the “hardness” of a clause. The efficiency of a SAT-solving search procedure depends on having well-tuned techniques to identify problematic clauses, so we base our procedure on those techniques. As several of the available SAT-solvers based on DPLL do, Minisat uses a heuristic to measure how actively a clause participates in the search process, in order to determine the next variable assignment. The idea behind that metric is that a clause with a high activity is a sign of unsatisfiability, and Minisat makes use of that knowledge to guide the backtracking process that searches for a satisfiable assignment. In this way, the aims of the SAT-solving optimizations and the identification of a probable unsatisfiable set of clauses coincide. We take advantage of this scenario to use already developed clause identification techniques that showed to enjoy a good empirical behavior.

3.1 Overview of the SAT-Solving Process

We will concentrate here on SAT-solvers based on the DPLL algorithm [12]. The key characteristics of the algorithm are: backtracking by conflict analysis, clause recording (which is also known as *learning*) and boolean constraint propagation (BCP) (see [13,14]). We start by giving some definitions.

Definition 1. A conjunctive normal form (CNF) formula φ on n variables x_1, \dots, x_n is the conjunction of m clauses $\omega_1, \dots, \omega_n$, each of which is the disjunction of one or more literals. A literal is the occurrence of a variable or its negation.

Most solvers operate with clauses in CNF. Observe that a formula φ can be thought of as an n -variable boolean function $f(x_1, \dots, x_n)$, where each clause of φ is an implicate of f .

Definition 2. Given an n -variable propositional formula φ , the satisfiability problem (SAT) consists in finding an assignment to the associated boolean function $f(x_1, \dots, x_n)$ that makes the function equal to 1, or otherwise proving that the function is the constant function 0.

Given a n -variable propositional formula, the backtracking search algorithm for SAT is implemented by a *search process* that explores the space of 2^n possible binary assignments to the variables. The exploration does not always traverse

the whole state space, but, as shown in [4], this may happen in a worst-case scenario.

The SAT-solving process works by extending a partial assignment of the formula variables. A variable whose binary value has already been determined is considered to be *assigned*; otherwise it is *unassigned*. An *assignment* for a formula φ is a set of assigned variables and their corresponding binary values. An assignment is *complete* when all the variables are assigned. A clause is said to be *unit* if the number of its unassigned literals is one.

We now give a general overview of the DPLL algorithm. For further details, see [12]. Starting from an empty truth assignment, the algorithm explores the assignment space and organizes the search for a satisfying assignment through a *decision tree*. In this way, each node of the tree represents the explicit assignment to an unassigned variable. The process iterates through the following steps:

1. **Search.** The current assignment is extended by deciding a binary value for an unassigned variable. This step involves deciding which unassigned variable to pick, and which value to assign. The search process terminates successfully if all the clauses become satisfied. It terminates unsuccessfully if some clauses are unsatisfied and all possible assignments have been tried.
2. **Propagation.** The current assignment is extended by analyzing the logical consequences of the assignments made so far. This step is known as Boolean Constraint Propagation (BCP), and it is based on unit clauses analysis. This analysis may extend the current assignment, and may also lead to the identification of unsatisfiable clauses, implying that the current assignment is not a satisfying assignment. This is known as a *conflict*, which is handled by the following step.
3. **Learning.** Given an assignment that produced a conflict, an analysis is made to determine a set of variable assignments that implied the conflict. From this set of variables a clause prohibiting that particular assignment is built and added to the clause database. Observe that this *learnt clause* is a consequence of the original set of clauses. This new clause should be thought of as a “witness” of the reason for the conflict, and it avoids regenerating the conflicting assignment that led to the current conflict.
4. **Conflict analysis.** This stage undoes the current assignment, if it is conflicting, so that another assignment can be tried. A conflict analysis is performed here, that identifies the point in the decision tree where precisely one of the literals of the learnt clause becomes unassigned. This is usually referred to as *backjumping* or *non-chronological backtracking* [13].

3.2 Activity Heuristics in the Search Process

The search procedure of a modern SAT-solver is usually a complex algorithm. Heuristics are needed to pick the next unassigned variable and to decide a value for it. Decision strategies (that range from randomly selecting variables to more sophisticated heuristics like JW-OS and JE-TS [15]) have a relevant impact over the SAT-solver’s performance (see [15]).

Minisat uses a variation of the so called Variable State Independent Decaying Sum (VSIDS) heuristic (which was originally introduced in the Chaff solver [14], and it is also used in BerkMin [16] with some differences in the updating process). Using VSIDS, a value is associated with each literal. When a clause is added to the database, the value associated with each literal in the clause is incremented. Periodically, all the counters are divided by a constant. The selection process in the search stage picks the variable with the highest associated value.

Minisat uses a variation of this technique, but applied to clauses. When a learnt clause is used in the analysis process of a conflict, its activity is incremented. Inactive clauses are periodically removed. This strategy can be viewed as attempting to satisfy the clauses involved in a conflict, but particularly attempting to satisfy the most *recent* clauses involved in a conflict. In fact, the decision heuristic of Minisat involves decaying the activity of clauses more often than the standard VSIDS heuristic. Benchmarks have shown that this schema responds faster to changes and avoid branching on out-dated variables [2].

Using this heuristic, the clauses with the highest activity values represent the clauses most actively involved in recent conflicts. Since a set of unsatisfiable clauses generates many conflicts, and therefore many conflict clauses, the high activity of a clause can be seen as a potential sign of unsatisfiability. The strategy we use to identify the hot core is to keep a set H (of fixed size) with clauses with the highest activity. HotCore updates this set every time the search process is triggered. When the user interrupts the SAT-solving process, HotCore uses the last updated set of clauses to calculate a hot core. This strategy can be thought of as an attempt to identify the set of clauses with the best chances to belong to an unsat core *given the current state* of the solving process. In Section 4 we discuss the empirical results obtained when choosing an appropriate size for H .

3.3 Extracting the Core

Once we have identified a set H of potentially unsatisfiable clauses, we want to associate H with its corresponding Alloy constraints. This will allow us to present to the user a highlighted part of the Alloy constraints representing the hot core, in the same way it is done for the unsat core. To implement the unsat core, the Alloy Analyzer applies an inverse translation that maps propositional clauses to Alloy formulas, and then it highlights them. We would like to reuse this feature and feed this translation with H , but the problem is that this inverse translation needs *original* clauses, that is, clauses that were in the output of the forward translation (from Alloy formulas to propositional clauses). The set H we have identified does not necessarily fulfill this requirement, since there may be clauses which are *consequences* of the original clause database. Therefore, we should be able to identify a set H' of clauses such that a) H' implies H and b) all the clauses in H' originally belong to the initial clause database.

To build H' , we implemented an extension of the algorithm proposed by Zhang and Malik in [17]. Let us briefly describe the original algorithm. Given an unsatisfiable set of Boolean formulas, the algorithm extracts a subset of this

set that is still unsatisfiable, using the unsatisfiability proof found by the SAT-solver. A SAT-solving process applied to an unsatisfiable set of clauses can be regarded as a resolution process that generates the empty clause. This can also be described with a *resolution graph*. Each node of the graph represents a clause, the root nodes are the clauses in the original set, and the internal nodes represent the learned clauses generated during the solving process. The edges in the graph represent resolution steps. An edge from a node a to a node b represents that a is a resolve source of the node b . This clearly defines a directed acyclic graph, where there is a node e that represents the empty clause. Observe that all the root nodes that are not in the transitive fan-in cone that has e as a vertex are not needed to construct that particular proof, and therefore can be ruled out. The root nodes in the cone represent a minimal subset of the original clauses that are needed *for that particular proof*.

The identified set is not a minimal subset in the general sense, but its size could be much smaller than the original set of clauses. The main advantage of this algorithm is that it scales well on very large instances [17]. This is one of the algorithms used by the Alloy Analyzer (among others, see [3,18]).

In our case, we do not have an unsatisfiability proof, but a set of clauses H that *could* lead to the empty clause. Looking at the resolution graph, every clause $c_i \in H$ defines a cone C_i , taking the transitive fan-in cone that has c_i as a vertex. Observe that the set of root clauses in C_i contains the original clauses needed to deduce c_i for that particular proof. Therefore we can define the set H' as the set of root nodes in $\bigcup_i C_i$. HotCore implements this idea in a relatively straightforward way, using a simple graph traversal algorithm.

4 Experimental Results

We are interested in measuring the quality of the unsat core approximation performed by HotCore. To achieve that, HotCore has been evaluated on ten unsatisfiable problems from the Alloy Analyzer distribution. The chosen problems come from a variety of domains and exhibit a wide range of behaviors (they are fully described in [1]). Each of them was ran on Alloy+HotCore for satisfiability in several scopes. The Alloy Analyzer offers three different algorithms to perform the unsat core extraction. All the comparisons were done with respect to the algorithm proposed by Zhang and Malik in [17]. We report the obtained results using scopes whose solving time is below a 10 minutes time bound. All experiments were performed on an Intel Core 2 duo 2.4GHz, with 2GB RAM. HotCore implementation was built on top of the Alloy Analyzer release 4.1.9. HotCore can be downloaded from <http://www.dc.uba.ar/hotcore>.

In order to measure the quality of the unsat core approximation, we have defined two different metrics: *Hit* and *Error*. *Hit* intends to represent how well the hot core approximates the unsat core, and *Error* tries to capture the error in that approximation. We provide a formal definition below.

Definition 3. *Given an unsat core U and a hot core H we define $\text{Hit} = \#(H \cap U) / \#(U) * 100$ and $\text{Error} = \#(H - U) / \#(H) * 100$.*

We are interested in using these metrics in terms of both propositional clauses and Alloy specification language. Therefore, we will add in *Hit* and *Error* the subscript p when we measure clauses at the level of propositional formulas, and the subscript a when we measure characters at the level of Alloy specification.

As we mentioned, HotCore shows the hot core to the user by highlighting a set of constraints, following the unsat core presentation schema. Consequently, evaluating HotCore at the Alloy model level seems a rather obvious choice, since the highlighted constraints conform the user expected feedback. On the other hand, due to the nature of the Alloy Analyzer’s translation, each Alloy constraint may result in several distinct propositional clauses. Additionally, some propositional clauses may not even have an Alloy model counterpart (such as the symmetry breaking predicates [19]). Therefore, measuring a hot core at the proposition formula level should lead to finer-grained results.

Problem	Size	Scope	Unsat	Hot	Hit _a	Error _a	Hit _p	Error _p	Hot	
									Unsat	Unsat
lists - reflexive	21	10	153	16	50	0	75	3	10%	10%
lists - symmetric	21	8	8	1	44	0	82	8	13%	13%
hotel2	65	15	5	1	100	0	78	4	20%	20%
hotel4	61	17	166	48	100	0	81	2	29%	29%
lights	20	200	8	4	100	0	96	1	50%	50%
addressBook1h	21	30	117	74	100	0	93	93	63%	63%
ringElection2	27	14	5	4	100	0	98	0	80%	80%
sets2	11	36	444	256	100	0	88	1	58%	58%
mediaAssets	61	30	31	19	91	1	90	1	61%	61%
p306-hotel	40	18	43	31	100	0	91	3	72%	72%
<i>Average</i>										45.6%

Fig. 1. Results shown with a stopping criterion of 75% Hit_p

In Figure 1 we show the results we have obtained running HotCore on the above-mentioned problems. The first four columns show the problem description in terms of: name of the problem, model size (in lines of code), scope of analysis and time required to compute the unsat core (in seconds). The fifth column records the instant in which the analysis was interrupted (also in seconds), that is to say, the instant when the hot core was identified. The stopping criterion was set to achieve *at least* 75% Hit_p . Observe that since the set of most active clauses is updated at every search iteration, we do not have a finer grained control over the coverage. This means that the verification was stopped in the first search iteration where the coverage was beyond that threshold. The remaining columns show Hit and $Error$ for the identified hot core using the two defined metrics and the rate between the times needed to compute the hot core and the unsat core.

We now discuss the results. Observe that in general we reach 75% Hit_p quite fast: on average, HotCore needed less than one half (specifically 45.6%) of the total solving time. Furthermore, these results were accomplished with a very low $Error_p$. Let’s look now at Hit_a and $Error_a$. We expect a high correlation between the coverage at the propositional and the Alloy level, and this is confirmed by the results. The example `addressBook1h` has a particular behavior: there is a significant difference between the two metrics, with 93 $Error_p$ and 0 $Error_a$.

This is a case where the set of identified propositional clauses outside the unsat core has no Alloy model counterpart. We believe that this may be a case where those clauses were introduced for symmetry breaking (or similar) purposes.

Let's turn now to the end-user perspective and let's analyze what happens in terms of the Alloy specification language. We want to show the HotCore results on the same ten problems but setting the stop criterion looking at the Alloy level coverage instead of at the propositional level. As we did for Hit_p , we set the threshold to 75% Hit_a . These results are shown in Figure 2. Observe that the time needed to reach 75% Hit_a was, in general, even less than the case when we used Hit_p as the metric to stop the solving process. On average, HotCore needed 20.6% of the total solving time. The improvement can be explained in terms of the exponential blow-up in the mapping from Alloy constraints to propositional clauses, together with a permissive behavior in Alloy's highlighting scheme.

Problem	Size	Scope	Unsat	Hot	Hit_a	$Error_a$	Hit_p	$Error_p$	$\frac{Hot}{Unsat}$
lists - reflexive	21	10	153	52	100	0	91	3	34%
lists - symmetric	21	8	8	5	97	0	96	7	63%
hotel2	65	15	5	1	97	0	28	12	20%
hotel4	61	17	166	7	99	0	44	3	4%
lights	20	200	8	3	100	0	37	2	38%
addressBook1h	21	30	117	23	91	0	4	95	20%
ringElection2	27	14	5	1	94	0	8	14	20%
sets2	11	36	444	1	100	0	3	14	0.23%
mediaAssets	61	30	31	2	88	0	7	0	6%
p306-hotel	40	18	43	1	98	0	19	17	2%
<i>Average</i>									20.6%

Fig. 2. Results shown with a stopping criterion of 75% Hit_a

Figure 3 (left) and Figure 3 (right) respectively show how Hit_a and $Error_a$ evolve during time for a subset of the studied examples. The X-axis represents time percentage (in a logarithmic scale) while the Y-axis shows the corresponding metric. Let's take a look at Figure 3 (left). On one hand, the hot core convergence grows exponentially fast. On the other hand, the convergence curve has very few oscillations, showing a quasi-monotone behavior. Let's analyze now Figure 3 (right). Observe that after 10% of the total time has flown, the error is below 15% for all the cases.

To sum up, the implemented heuristic has shown to behave successfully in many cases. In order to validate the unsat core approximation, we would also like to find some examples that show the weakness of our method. We could not find this behavior among the known available Alloy examples, so we designed some ad hoc cases specifically headed to make the heuristic fail. We combined two independent Alloy models: one hard satisfiable model together with one unsatisfiable. We expected to verify that when the SAT-solver is working with the satisfiable part of the specification, the most active clauses will not be related to the real unsat core. We used two satisfiable Alloy models presented in [20] that showed to be computationally hard to solve. We combined them with RingElection, known to be unsatisfiable. Our expectations were confirmed, and HotCore showed on both examples a Hit near 0% and a $Error$ near 100% for

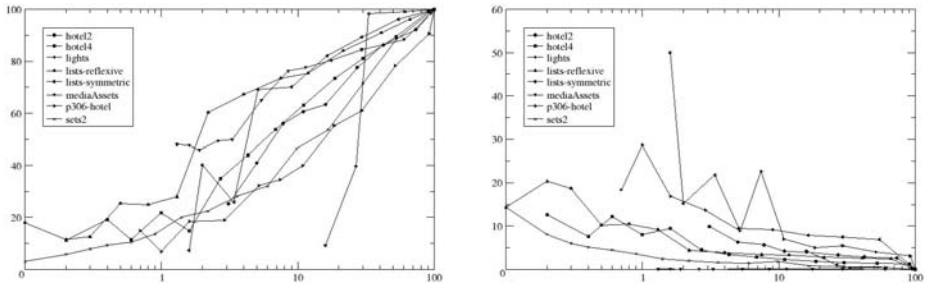


Fig. 3. Hit_a and $Error_a$ evolution

almost every search iteration (for both metrics). The hot core coincides with the unsat just in the last search iteration, when around 70% of the total solving time already went by. Nevertheless, the difficulty in finding these examples can be regarded as a good sign of the general behavior of the used heuristic.

To close this section we want to mention some technical details about the implementation. We ran the above mentioned examples varying the size of the set of most active clauses. We have seen that the results do not differ significantly when more than 100 clauses are used. The approximation begins to be less accurate when we use a set with less than 20 clauses. Hence, after several experiments we have chosen 100 to be the default size. This can be seen as a sign that MiniSat’s activity heuristic is efficiently tuned, and therefore a relatively small set clauses is enough to be representative. Furthermore, the heuristic we used has almost no computational overhead. The main reason for this is that we build HotCore on top of the existing MiniSat’s activity heuristic.

5 Conclusions and Further Work

In this article we presented HotCore, an extension of the Alloy Analyzer capable of approximating the unsat core of a given set of Alloy constraints. We motivated the use of this tool through several use cases, and we showed that the heuristic we proposed to identify the hot core has a very good empirical behavior. HotCore proved to have a nice convergence speed and a low miss rate. Since it is quite frequent to exceed the waiting time bounds during an Alloy analysis, an end-user could obtain a significant profit from using this extension.

Much work rest to be done. On one hand, using a fixed size set of highly active clauses seems a relatively naive approach, and therefore we want to test more refined heuristics. For example, we want to consider the set of clauses that shows a recent activity beyond a given threshold (or a given percentile). On the other hand, the core extraction algorithm we used is quite efficient, but not very accurate in some cases. We would like to evaluate other core extraction algorithms in the context of HotCore, like the proposed in [19]. We want to investigate other techniques to analyze the information profiled during the solving process. We are also interested in applying the same concept to the Alloy’s counterexample

generation and visualization. Our idea is to analyze the SAT-solver state when the solving process is interrupted in order to build a *potential* counterexample. For example, the current partial assignment may be a good lead for this purpose.

References

1. Jackson, D.: Software abstractions: logic, language, and analysis. MIT Press, Cambridge (2006)
2. Een, N., Sorensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
3. Torlak, E., Chang, F., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008)
4. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC 1971, pp. 151–158. ACM, New York (1971)
5. Sinz, C.: Visualizing sat instances and runs of the dpll algorithm. *Journal of Automated Reasoning* 39(2), 219–243 (2007)
6. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: *Procs. of the 10th Conf. on Artificial Intelligence*, pp. 440–446 (1992)
7. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1993)
8. Mazure, B., Saïs, L., Grégoire, É.: A powerful heuristic to locate inconsistent kernels in knowledge-based systems. In: IPMU 1996, pp. 1265–1269 (1996)
9. Grégoire, E., Mazure, B., Piette, C.: Boosting a complete technique to find mss and mus thanks to a local search oracle. In: *Proceedings of IJCAI*, pp. 2300–2305 (2007)
10. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
11. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the small scope hypothesis (2002), <http://sdg.csail.mit.edu/pubs/2002/SSH.pdf>
12. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Commun. ACM* 5(7), 394–397 (1962)
13. Silva, J.P.M., Sakallah, K.A.: GRASP – A new search algorithm for satisfiability. In: 1996 IEEE/ACM international conference on Computer-aided design, pp. 220–227. IEEE Computer Society, Washington (1997)
14. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Design Automation Conference*, pp. 530–535 (2001)
15. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. In: Barahona, P., Alferes, J.J. (eds.) EPIA 1999. LNCS (LNAI), vol. 1695, pp. 62–74. Springer, Heidelberg (1999)
16. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics* 155(12), 1549–1561 (2007)
17. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In: *Proceedings of SAT*, vol. 3 (2003)
18. Bruni, R., Sassano, A.: Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. *ENDM* 9, 162–173 (2001)
19. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
20. Galeotti, J.: Distributed sat-based analysis of object oriented code. In: *Proceedings of Symposium on Automatic Program Verification (APV 2009)*, Rio Cuarto, Argentina, ETH Zurich (February 2009)

Supporting Reuse in Event B Development: Modularisation Approach

Alexei Iliasov¹, Elena Troubitsyna², Linas Laibinis², Alexander Romanovsky¹,
Kimmo Varpaaniemi³, Dubravka Ilic³, and Timo Latvala³

¹ Newcastle University, UK

{alexei.iliasov,alexander.romanovsky}@ncl.ac.uk

² Åbo Akademi University, Finland

{linas.laibinis,elena.troubitsyna}@abo.fi

³ Space Systems Finland

{Dubravka.Ilic,Timo.Latvala,Kimmo.Varpaaniemi}@ssf.fi

Abstract. Recently, Space Systems Finland has undertaken formal Event B development of a part of the on-board software for the BepiColombo space mission. As a result, lack of modularisation mechanisms in Event B has been identified as a serious obstacle to scalability. One of the main benefits of modularisation is that it allows us to decompose system models into components that can be independently developed. It also helps to manage complexity of models that in the industrial setting are usually very large and difficult to comprehend. On the other hand, modularisation enables reuse of formally developed components in the formal product line development. In this paper we propose a conservative extension of Event B formalism to support modularisation. We demonstrate how our approach can support reuse in the formal development in the space domain.

1 Introduction

In the Deploy project [11], Space Systems Finland (SSF) has performed a pilot Event B development [13] of a part of on-board software for the BepiColombo space mission [8]. The developed system is responsible for controlling the instruments producing valuable scientific data critical for the success of the mission. The undertaken development aimed at identifying the strengths and weaknesses of the Event B method and its supporting tool – the RODIN platform [18]. The experience demonstrated that the refinement approach provides a suitable design technique. It allows us to structure complex requirements and promotes disciplined development via abstraction and proofs. However, it has also become obvious that the lack of modularisation makes Event B unscalable for formal development of industrial systems. In this paper we propose a conservative extension of Event B language that supports a simple modularisation idea.

The idea of modules is very well known and is supported by most of the formal frameworks. In its simplified form, a module interface can be defined via pre- and post-conditions of the provided operations. However, in our case introducing preconditioned operations in Event B was unacceptable due to two main reasons. Firstly, preconditioned operations are not supported by the RODIN platform and building a new tool of similar strength would require significant investments. Secondly, introduction of a

preconditioned operation would seriously complicate the proof obligations required to verify correctness and hence would lower the degree of automation in the development. Therefore, our approach (a conservative extension of Event B) is strictly driven by the pragmatic needs and oriented towards automation. We believe however that, by enabling modular development in Event B, we not only improve scalability of formal modelling but also potentially increase its productivity and reuse.

In this paper we briefly describe the on-board software as well as modelling experience gained by SSF. Then we present our proposal for introducing modularisation in Event B and demonstrate how the system can be redeveloped in a modular fashion.

2 Challenges and Experiences in Formal Development of Onboard Software

Spacecraft-embedded software – onboard software – is responsible for managing various spacecraft operations. For instance, the controlling software is critical to the mere survivability of a mission, while scientific software is responsible for correct and effective handling of high volume of data generated by scientific experiments. Therefore, failure of onboard software can have major repercussions. Yet, onboard software must withstand extreme conditions of the space environment and operate by using very limited resources provided by hardware. It is clear that these factors make the design, implementation and verification of onboard software very challenging.

Space Systems Finland (SSF) is one of software providers for the European Space Agency mission BepiColombo [8]. The main goal of the mission is exploration of the planet Mercury. The mission comprises various scientific studies, e.g., analysis of its internal structure and surface, geological evolution etc. To achieve the defined scientific goals, one of the mission orbiters will carry remote sensing and radioscience instrumentation. SSF is responsible for developing software for an important part of the orbiter – the data processing unit. The company has undertaken formal development [13] of it in the Event B framework with the support of the RODIN platform [18].

The data processing unit (DPU) is used to control two scientific instruments: Solar Intensity X-ray and particle Spectrometer (SIXS) that records the radiation from the Sun, and Mercury Imaging X-ray Spectrometer (MIXS) that records fluorescent X-rays from the planet surface. The system consists of three main software components: the Core Software (CSW), the SIXS instrument application software, and the MIXS instrument application software.

In general, the behaviour of the system consists of receiving telecommands (TC) from the BepiColombo platform and producing corresponding telemetry data (TM). CSW is responsible for validation of syntactical and semantical integrity of each received TC. In particular, it checks that each TC adheres to the Telemetry and Telecommand Packet Utilization standard [17]. If validation fails then the corresponding TM is generated. Otherwise, a TC is placed in the pool of TCs waiting for execution.

We have constructed a formal system model in the RODIN platform. The resultant specification has 20 variables, 61 events and 38 invariants. Additionally, the static data structures (15 sets, 88 constants) are defined by formulating 207 axioms and 20 theorems. The model text (apart from definition of the data structures) exceeds 40 pages.

The modelling effort highlighted some of the shortcomings of Event B method when building large-scale models:

- It is not clear how to reuse the conducted development in the similar projects;
- Lack of modularisation support hinders independent development of several sub-systems – it is not obvious how spread the effort in a team of developers;
- Without decomposition(modularisation), a specification of even a relatively simple realistic system becomes very large and difficult to comprehend.

There is a clear need for modularisation mechanisms in formal Event B development. Next we discuss our proposal for such a mechanism.

3 Event B

In this section we introduce our formal framework – The B Method [2]. It is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications. Recently the B method has been extended by the Event B framework [3, 15], which enables modelling of event-based (reactive) systems. In fact, this extension has incorporated the action system formalism [5, 6] into the B Method.

The B Method development starts from creating a formal system specification. The basic idea underlying stepwise development in B is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*.

A simple Event B specification has the following general form:

```

MACHINE AM
SEES Context
VARIABLES v
INVARIANT Inv
EVENTS
  INITIALISATION = ...
  E1 = ...
  ...
  EN = ...
END

```

Such a specification encapsulates a local state (program variables) and provides operations on the state. The operations (called *events*) can be defined as

```

ANY vl WHERE g THEN S END

```

where *vl* is a list of new local variables (parameters), *g* is a state predicate, and *S* is a B statement (assignment) describing how the program state is affected by the event. In case when *vl* is empty, the event syntax becomes **WHEN** *g* **THEN** *S* **END**. Both ordinary and non-deterministic assignments can be used to specify state change. The non-deterministic assignments are of the form $v := | \text{Post}(v, v')$, where *Post* is the post-condition relating the variable values before and after the assignment.

The events describe system reactions when the given **WHEN** or **WHERE** conditions are satisfied. The **INVARIANT** clause contains the properties of the system (expressed

as predicates on the program state) that should be preserved during system execution. The data types and constants needed for specification of the system are defined in a separate component called Context.

4 Introduction to Modules in Event B

Our primary goal is to conservatively extend the Event-B language with a possibility of (atomic) operation calls. Such an extension would naturally lead to the notion of modules – components containing groups of callable operations. Moreover, modules can have their own (external and internal) state and the invariant expressing properties on this state. The important characteristic of modules is that they can be developed separately and then composed with the main system during its formal development. Since we are interested in incorporating modules into Event B modelling, it should be also possible to statically check the correctness of such a composition.

Let us start with an “ideal” (somewhat extreme) example of a general Event B operation that we would like to be able to express in our formal language.

```

op =
    WHEN Guard( $v_1, \dots, v_N$ )
    THEN
         $v_1$  :| ... op1_call(parameters1) ...
        ...
         $v_N$  :| ... opN_call(parametersN) ...
        opN+1_call(parametersN+1)
        ...
        opN+K_call(parametersN+K)
    END

```

Here $op_i_call(\dots)$ are either function or procedure calls from available modules. A procedure call can be considered as special case of a function call (returning the void value). Thus from now on we will focus only on modelling function calls in Event-B.

Once an enabled event is chosen for execution, all its actions are executed atomically and in parallel. However, the standard semantics of a function call, realised in most programming languages, prescribes the well-defined order of execution steps:

1. Actual parameter expressions are evaluated and passed to a module operation;
2. The operation is executed on the given parameters and the module state. The operation result is returned to the calling operation;
3. The actions of the calling operation are executed, substituting the function calls with the returned results.

Moreover, the atomicity of an event operation with function calls should be preserved – no other event operation of the main system can intervene in between. Our challenge in this paper is to implement this standard functionality within the Event B semantics.

We split our task into two separate issues. First, we show how we can introduce modules and function calls using model decomposition. Next, we assume availability

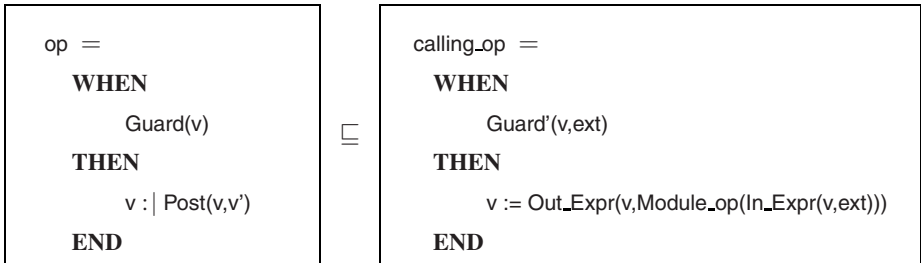
of pre-defined modules and demonstrate correctness of our specification containing module operation calls. The latter is a special case of verifying model composition.

4.1 Introducing Modules via Model Decomposition

In this paper we use the J.-R. Abrial's approach on Event-B decomposition [4]. The approach allows to split an Event B specification into several components (sub-models) that can be developed separately. It also formally guarantees that the final re-composed system will be a refinement of the original one.

The decomposition is based on partitioning the model operations among the new components. The model variables are distributed as well, either as *internal variables* belonging to some particular components, or as *shared variables* that can be accessed by several components. To make the components self-contained, each of them is complemented by special *external events*, abstractly modelling how the shared variables may be modified by other components. The approach also restricts data refinement of the shared variables to make the decomposed system consistent.

Let us start with a simple generic example of an Event B operation. We would like to refine it so that it delegates (part of) its functionality to an external operation and then uses the returned result. In other words, the operation refinement should be of the form:



where `Post` is the postcondition of the original event, `In_Expr(v, ext)` is the actual parameter expression, `Out_Expr(...)` is a state expression incorporating the result of the operation call, and `ext` is the externally visible part of the module state.

We interpret the refined operation as a syntactic sugaring hiding the actual definition in terms of the current Event B language. The idea is to model a function call by three events, simulating the three-step execution described above. Moreover, these three events should be introduced in such a way that we could decompose the system by distributing the system state and operations between the calling and called components.

The execution of a called module operation is abstractly modelled by `Module_op` presented below. Note that, in addition to calculating the result `res`, an operation call can also update the module state `ext`. The execution of a module operation is wrapped by two events of the calling component: `call_preparation`, which passes parameters to the module, and `call_finalisation`, which incorporates the returned results.

The variables `i_flag` and `o_flag` (of the type `0..1`) are used to enforce the fixed order of execution between the main component and a module: first `call_preparation`, then `Module_op`, and finally `call_finalisation`. In addition, to guarantee atomicity of an

```

Module_op =
  WHEN
    i_flag ≠ o_flag
  THEN
    ext,res :| M_Post(pars,ext,ext',res')
    o_flag := 1-o_flag
  END

```

```

call_preparation =
  WHEN
    Guard'(v,ext)
    i_flag = o_flag
    pars = NIL
  THEN
    pars := In_Expr(v,ext)
    i_flag := 1-i_flag
  END

```

```

call_finalisation =
  WHEN
    i_flag = o_flag
    pars ≠ NIL
  THEN
    v := Out_Expr(v,res)
    pars := NIL
  END

```

operation call, all the other operations of the calling component should be blocked until `call_finalisation` finishes. It can be achieved by correspondingly strengthening their guards. Essentially, the above solution is a special case of the alternating bit protocol.

This refinement step also achieves partitioning the state and operations between components. The variables `res`, `o_flag` can be put into the future module component, while `pars`, `i_flag`, `v` belong to the main specification. Following the Abrial's approach, we can decompose the system by moving `Module_op` into a separate module.

To prove operation refinement, we need to show the connection between the abstract guard `Guard` and the strengthened guard `Guard'`, as well as the postcondition `Post` in the main specification and the postcondition `M_Post` of the module operation. Specifically, the following two theorems should be proved as additional proof obligations:

$$\text{Guard}'(v, \text{ext}) \wedge \text{M_Inv}(\text{ext}) \Rightarrow \text{Guard}(v)$$

$$\text{M_Post}(\text{In_Expr}(v, \text{ext}), \text{ext}, \text{ext}', \text{res}) \wedge \text{M_Inv}(\text{ext}) \Rightarrow \text{Post}(v, \text{Out_Expr}(v, \text{res}))$$

where `M_Inv` is the module invariant on its external state.

4.2 System Development via Model Composition

In the previous section we showed how we can delegate a part of functionality of the main specification to a module by means of model decomposition. In practice, however, we are more interested in the opposite – composing our systems by using a collection of pre-defined modules.

In our examples above, execution of a module operation was specified as a single event. In general, a module implementation could contain many callable operations, each of them consisting of a group of events. Demonstrating the correctness of a operation call would then become a non-trivial task.

Since Event B is a refinement-based formalism, the problem can be solved by applying the classical rules of program correctness, in particular, the correctness rules for operation calls [10, 12]. Basically, following these rules, it is sufficient to show the relationships between the pre- / postcondition of a operation call and the corresponding pre- / postcondition of a module operation. Specifically, we need to prove that

$$\text{Guard} \wedge \text{M_Inv} \Rightarrow \text{M_Guard}$$

$$\text{M_Post} \wedge \text{M_Inv} \Rightarrow \text{Post}$$

where Guard, Post and M_Guard, M_Post specify respectively the calling and module operations.

The pre- and postconditions of a module operation then become a part of the externally visible module description, alongside with the external module variables and invariant. Such an external description is called a *module interface*. An exact structure of a module interface will be presented in the next section.

Let us recall the example from the previous section. However, this time the module interface describing the module external state, invariant, and operation guards and postconditions is available. Then it can be shown that the operation calling_op is just a syntactic sugaring for the following (provided that the above conditions on the guards and postconditions are proved):

```

calling_op =
  ANY
    ext', res
  WHERE
    Guard'(v,ext)
    M_Post(In_Expr(v,ext),ext,ext',result)
  THEN
    v := Out_Expr(res)
    ext := ext'
  END

```

The required sequence of parameter passing, external operation execution, and returning of its results is now implicitly modelled by new local variables and their initialization in the operation guard.

The described approach essentially represents the substitution principle used in reasoning about operation calls and their correctness. However, the examples considered so far are still pretty simple. In the next section we will discuss the structure and semantics of modules and their interfaces in a general case.

5 Extending Event B with Modules

Our main objectives are to facilitate model reuse and enable concurrent development of formal models. The interface concept plays a central role in achieving this. The introduction of an operation call can be validated by considering only an interface description of a called operation. Symmetrically, an implementation of an operation does not have to be aware of a possible context of an operation call since the validation is done against the requirements stated in the interface. In other words, a module interface allows a module user to invoke module operations and observe module external variables without having to inspect module implementation details.

In our approach, a module interface consists of external module variables (w), constants (c), and sets (s), the external module invariant, and a collection of module operations, characterised by their guards and postconditions.

```

MODULE_INTERFACE MI =
  SEES Interface_Context
  VARIABLES w
  INVARIANT M_Inv(c, s, w)
  OPERATIONS
    res ← op1(par) =
      GUARD M_Guard1(c, s, par, w)
      POSTCONDITION M_Post1(c, s, par, w, w', res')
  ...
END

```

A module interface does not have an initialisation (it is provided by a module implementation) and there are no events. However, an interface still must satisfy certain consistency conditions typical for Event B specifications – operation *feasibility* and preservation of the module invariant:

$$M_Inv(c, s, w) \wedge M_Guard(c, s, p, w) \Rightarrow \exists res', w' \cdot M_Post(c, s, p, w, w', res') \quad (1)$$

$$M_Inv(c, s, w) \wedge M_Guard(c, s, p, w) \wedge M_Post(c, s, p, w, w', res') \Rightarrow M_Inv(c, s, w') \quad (2)$$

Applicability of a module operation is typically specified by providing its precondition. In our case, using guards instead of preconditions theoretically allows the operation condition to be strengthened during refinement, which can lead to the operation becoming no longer applicable. We avoid this problem by requiring that each interface specification (essentially representing a single event) is separately refined by a group of events. The Event-B condition on deadlock freeness prohibits introducing new deadlocks into the system, thus effectively disallowing strengthening the disjunction of guards of such event group. More about module implementation is explained in the next section.

A module development always starts with the design of an interface. Once an interface is formulated and declared final it cannot be altered in any manner. This ensures that an operation call context is recomposable with an operation implementation, provided by the last refinement step of a module body.

5.1 Module Body

A module interface formally defines a collection of module operations. Obviously, it should be complemented by the corresponding module body that provides a suitable implementation for each operation. Since an Event-B specification has a flat structure, there is a problem of relating an operation interface to a set of events implementing the operation. To show correctness of a module implementation, we need a clear separation between the events implementing different module operations.

The solution we are putting forward is based on an introduction of a simple specification structuring mechanism. The events associated with a particular operation are put together forming an *event group*. Several event groups make up a body of a module implementation, one group for each operation interface. The defining property of an event group is the following: once a control is passed to a group, the group runs till termination without interference from other groups. This allow us to formulate correctness conditions by considering only an operation and its associated event group.

Events groups simply partition events of a machine. A module body defining a collection of groups has the following structure:

```

MODULE M =
  VARIABLES w
  INVARIANT M_Inv
  GROUP group_name_1
    (events)
  GROUP group_name_2
    (events) ...
END

```

The name of a group must match the name of an operation interface definition. Each operation interface is associated with one group and vice versa. The termination of an event group corresponds to the one of an operation call. Events of a group also obey the usual Event-B consistency and refinement conditions with an additional constraint requiring that a refined event inherits a group membership from its abstract counterpart.

The pre- and postconditions of an operation interface define high-level requirements to the behaviour of an event group. At least one event of an event group must be enabled in the state described by the operation guard.

$$M_Guard \Rightarrow G_1 \vee G_2 \vee \dots \vee G_n \quad (3)$$

Each of the events returning control back from an event group must satisfy the operation postcondition and provide suitable return values.

$$Post_{ev}(w, w', res') \wedge \neg(G_1(w') \vee \dots \vee G_n(w')) \Rightarrow M_Post(c, s, par, w, w', res') \quad (4)$$

where $Post_{ev}$ is the event postcondition. A divergent event group cannot be a proper implementation of an operation. Therefore, In the first model realising a given interface (that is, an abstract module implementation) all the event groups must be terminating. The further refinement steps have to demonstrate the non-divergence of new events, as it is done in a conventional Event-B development.

5.2 Operation Invocation

The syntactic shorthand for an operation invocation is a function call. The interpretation behind such a shorthand is based on the interface attributes of an operation: its guard and postcondition. We have already discussed a simple case with just one invocation. However, our approach scales well to several invocations possibly containing complex interlinks such as using the result of one operation call as a parameter for another.

The semantics of an operation call is given by the computation of an equivalent statement that would be free from the call. Let us consider the following general case of an event which action relies on an operation call:

$$E = \text{WHEN } G(v, w) \text{ THEN } v : | \text{Post}(v, w, v', \text{op}(a)) \text{ END}$$

Here the predicate $Post$ is the before-after predicate of the event E . It relates the current model state v to the next state v' and also, indirectly, via the operation call, the current external module state w to the next state w' . The result of the operation call $\text{op}(a)$ is used in $Post$ to constrain v' . The following rewrite rule replaces the operation call with an equivalent characterisation based on the module interface pre- and postconditions:

$$\begin{aligned} E = \text{ANY } res, w' \text{ WHERE } M_Inv(w) \wedge M_Guard(par, w) \wedge M_Post(par, w, w', res) [a/par] \\ \text{THEN} \\ \quad v : | \text{Post}'(v, w, v', res) \\ \quad w := w' \\ \text{END} \end{aligned}$$

where $M_Inv(w)$ is the module invariant and M_Pre and M_Post are the guard and postcondition of the operation op . The new postcondition $Post'$ is computed by replacing all the occurrences of op invocations with the local variable res , constrained in the event guard to possible return values of op .

Since there can be more than one such invocation, the rule has to be applied iteratively. The important point is the order in which invocations are eliminated. In a general case, there is a causal link between calls because each subsequent call may observe side effects of all the preceding calls. Another form of a causal link is passing the result of an operation call as a parameter to another call. Once this ordering of calls is defined, we apply the above rule iteratively. The result is the following syntactic translation. For some event depending on a sequence of operation calls $op_1(a_1), \dots, op_n(a_n)$

$$E = \text{WHEN } G(v, w) \text{ THEN } v : | \text{Post}(v, w, op_1(a_1), \dots, op_n(a_n), v') \text{ END}$$

the corresponding (free of operation calls) translation is computed as follows:

$$\begin{aligned} E = \\ \text{ANY } res_1, w'_1 \text{ WHERE } G(v, w) \wedge \text{call}(1)[a_1 / par_1][\text{osub}(0)] \\ \quad \text{ANY } res_2, w'_2 \text{ WHERE } \text{call}(2)[a_2, w'_1 / par_2, w_2][\text{osub}(1)] \\ \quad \dots \\ \quad \text{ANY } res_n, w'_n \text{ WHERE } \text{call}(n)[a_n, w'_{n-1} / par_n, w_n][\text{osub}(n-1)] \end{aligned}$$

THEN

$$w := w'_n$$

$$v : | \text{Post}(v, \text{par}, \text{op}_1(a_1), \dots, \text{op}_n(a_n), v')[\text{osub}(n)]$$

END

where $[\text{osub}(k)]$ is the substitution $[\text{res}_1, \dots, \text{res}_k / \text{op}_1(a_1), \dots, \text{op}_k(a_k)]$, and $\text{call}(k)$ stands for $M_Inv(w) \wedge M_Pre_k(w_k, \text{par}_k) \wedge M_Post_k(\text{par}_k, w_k, w'_k, \text{res}_k)$. Here Pre_k and $Post_k$ are the pre- and post-conditions of the operation op_k . A nested **ANY** construct is a syntactic sugaring that may be reduced to a single **ANY**. More details on this may be found in the Rodin deliverable on the Event-B language [15].

The expansion of operation calls into a plain Event-B notation reduces the problem of operation call verification to conventional set of proof obligations generated for an Event-B event. However, we are not proposing to do such conversion in practice – this would undermine all the benefits provided by a syntactical representation of an operation call. Instead, we rely on the expanded form to derive the proof obligations necessary to demonstrate event correctness. From practical view, a tool implementing the operation call mechanism would do the operation call expansion as an intermediate step prior to the generation of proof obligations.

6 Modularisation of the DPU Unit

This section presents an application of our modularization approach in Event B to model one of the important DPU subsystems, responsible for TC validation.

The arrived telecommands should be validated (i.e., checked for syntactic and semantic correctness of their fields) before they are forwarded to be executed. The core software is responsible for syntactic (“early”) checking, while the telecommand target software (which can be either the core software or application software) does more thorough (“late”) semantical checking.

In the Event B specification, the validation stage of telecommand processing corresponds to a group of events, covering different cases depending on the telecommand type, the software component (process) it is targeted to, the current core software mode etc. As a result of validation, the status of the processed telecommand is changed to either **Accepted** or **Rejected**. Furthermore, the additional set variable **Exclusive_Rej** is updated in the case when the core software rejects the telecommand. The information from **Exclusive_Rej** is needed by the core software later – in the reporting phase.

One example of such validation events is given below. **Reject_Private_TC_Early** is an abstract event specifying the case when the received TC belongs to private (i.e., mission-specific) TC type and the core software is not in the operational mode (i.e., is on standby or in the safe mode). As a result, the core software rejects the telecommand and marks it as “exclusively rejected”.

Many implementation details describing the validation process (especially the acceptance of TCs) are still missing and could be added in the later refinement steps. However, we would like to move the whole group of validation cases into a separate module (called **Validation**) and develop this module further independently. The case

```

Reject_Private_TC_Early =
  ANY tc_handler WHERE
    tc ∈ dom(TC_pool)
    TC_status(tc_handler) = TC_Unchecked
    TCpool(tc_handler) ∈ VALID_TCS
    Type_of_TC(TCpool(tc_handler)) ∈ PRIVATE_TC_TYPES
    CSW_mode ≠ Operational
  THEN
    TC_status(tc_handler) := TC_Rejected
    Exclusive_Rej := Exclusive_Rej ∪ {tc_handler}
  END

```

analysis and application of concrete validation actions would happen then within the Validation module. Therefore, we can specify the validation phase within a single operation event containing a call to the operation Validate described in this module.

```

Validate_op =
  ANY tc_handler WHERE
    tc ∈ dom(TC_pool)
    TC_status(tc_handler) = TC_Unchecked
  THEN
    TC_status(tc_handler) := Validate(tc_handler,CSW_mode)
  END

```

The parameters for calling the Validate operation are the TC being processed as well the current core software mode. The returned result is the new status of the processed TC. Please note the absence of the variable Exclusive_Rej in the calling operation. The reason for that is that we turn Exclusive_Rej into an external variable of the new module. The "external" status would allow other components to read the current value of this variable. The variable will be updated internally, when needed to record "exclusive" rejection. The additional module operation Remove_Exclusive would allow the calling component to remove tc_handler from Exclusive_Rej after it served its purpose (i.e., in the reporting phase).

The following excerpt of the Validation module interface contains declaration of the external module variable Exclusive_Rej as well as the interfaces for the operations Validate and Exclusive_Remove.

```

MODULE_INTERFACE Validation =
  VARIABLES Exclusive_Rej
  INVARIANT
    Exclusive_Rej ⊆ TC_ADDRESSES

```

OPERATIONS

$$\text{res1} \leftarrow \text{Validate}(\text{tc_handler}, \text{CSW_mode}) =$$
GUARD

$$\begin{aligned} &\text{tc_handler} \in \text{dom}(\text{TCpool}) \\ &\text{CSW_mode} \in \text{MODES} \\ &\text{TC_status}(\text{tc_handler}) = \text{TC_Unchecked} \end{aligned}$$
POSTCONDITION

$$\begin{aligned} &\text{res1} \in \{\text{TC_Accepted}, \text{TC_Rejected}\} \\ &\text{tc_handler} \in \text{Exclusive_Rej}' \Rightarrow \text{res1} = \text{TC_Rejected} \\ &\text{TC_pool}(\text{tc_handler}) \notin \text{VALID_TCS} \Rightarrow \text{tc_handler} \in \text{Exclusive_Rej}' \\ &\text{Type_of_TC}(\text{TC_pool}(\text{tc_handler})) \in \text{PRIVATE_TC_TYPES} \wedge \\ &\quad \text{CSW_mode} \neq \text{Operational} \Rightarrow \text{tc_handler} \in \text{Exclusive_Rej}' \end{aligned}$$

$$\text{res2} \leftarrow \text{Exclusive_Remove}(\text{tc_handler}) =$$
GUARD

$$\begin{aligned} &\text{tc_handler} \in \text{Exclusive_Rej} \\ &\text{TC_status}(\text{tc_handler}) = \text{TC_Rejected} \end{aligned}$$
POSTCONDITION

$$\begin{aligned} &(\text{res2} = \text{TRUE}) \Rightarrow (\text{Exclusive_Rej}' = \text{Exclusive_Rej} \setminus \{\text{tc_handler}\}) \\ &(\text{res2} = \text{FALSE}) \Rightarrow (\text{Exclusive_Rej}' = \text{Exclusive_Rej}) \end{aligned}$$

7 Conclusions

There are three major approaches to decomposition and modularisation. One is to identify a general theory that, once formally defined, would contribute to the main development. For instance, a model realising a stack-based interpreter could be simplified by considering the stack concept in isolation, creating a general theory of stacks and then reusing it in the main development. Such an approach was investigated in, e.g., [9].

Another form of decomposition is based on splitting a system into a number of parts and then proceeding with independent development of each part. At some point, the model parts are recomposed to construct the overall final model. This decomposition style often relies on the monotonicity of program refinement, although further constraints must be satisfied to ensure the validity of a recomposed model [1, 15, 7].

Finally, decomposition may be realised by hierarchical model structuring, where part of an overall system functionality is encapsulated in a self-contained modelling unit embedded into another unit. Such an approach is conceptually close to a procedure (function) call in programming languages. It helps to structure detailed models in a way a system would be realised in software. This is the approach we have taken in this work.

In particular, we have proposed a pragmatic approach to supporting modularisation in Event B. This work was motivated by the formal development conducted by Space Systems Finland [13]. The analysis of the development has shown that the lack of modularisation makes the approach unscalable. Yet the top-down development paradigm and

automated verification offer an attractive design platform. Our conservative extension of Event B alleviates scalability problem while preserving all the benefits.

The proposed approach to modularisation can be seen as a special case of the "shared variables" type of decomposition by J.-R. Abrial [15]. Abrial aims at enabling decomposition for distributed systems. In our case, the systems under construction are sequential, even though their functionality is distributed among several modules. Our goal was to enable parallel development of several subsystems as well as reuse formally developed modules. Other proposals include the "shared events" style decomposition for distributed systems [7] as well as supporting event fusion in Event B [14]. However, all these works offer more general and hence more difficult to implement alternatives for modularisation.

Currently we are working on implementing our approach as a plug-in to the RODIN platform. A prototype version of this plug-in is already available [16].

Acknowledgments

This work is supported by IST FP7 DEPLOY Project.

References

- [1] Abadi, M., Lamport, L.: Composing Specifications. *ACM Transactions on Programming Languages and Systems* 15(1), 73–132 (1993)
- [2] Abrial, J.-R.: *The B-Book*. Cambridge University Press, Cambridge (1996)
- [3] Abrial, J.-R.: Extending B without Changing it. In: *Proceedings of 1st Conference on the B Method*, Nantes, France, November 1996, pp. 169–191. Springer, Heidelberg (1996)
- [4] Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.* 77(1-2), 1–28 (2007)
- [5] Back, R.: Refinement calculus, Part II: Parallel and reactive programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) *REX 1989*. LNCS, vol. 430, pp. 67–93. Springer, Heidelberg (1990)
- [6] Back, R., Sere, K.: Superposition refinement of reactive systems. *Formal Aspects of Computing* 8(3), 1–23 (1996)
- [7] Butler, M.: *Decomposition Structures for Event-B*. In: *Integrated Formal Methods* (2009)
- [8] Factsheet: BepiColombo. ESA Media Center, Space Science (15.01.2008), http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html
- [9] Fitzgerald, J.: *Modularity in Model-oriented Formal Specifications and its Interaction with Formal Reasoning*. University of Manchester, Ph.D. Thesis (1991)
- [10] Gries, D., Levin, G.: Assignment and Procedure Call Proof Rules. *ACM Transactions on Programming Language Systems* 2, 564–579 (1981)
- [11] Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, <http://www.deploy-project.eu/>
- [12] Martin, A.J.: A General Proof Rule for Procedures in Predicate Transformer Semantics. *Acta Informatica* 20, 301–313 (1983)
- [13] OBSW formal development in Event B, <http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html>
- [14] Poppleton, M.: Decomposition Structures for Event-B. In: *Proc. of ABZ 2008: Int. Conference on ASM, B and Z*, London September 16-18 (2008)

- [15] Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, <http://rodin.cs.ncl.ac.uk/>
- [16] RODIN modularisation plug-in. Documentation, http://wiki.event-b.org/index.php/Modularisation_Plug-in
- [17] Space Engineering: Ground Systems and Operations Telemetry and Telecommand Packet Utilization, ECSS-E-70-41A. ECSS Secretariat (30.01.2003), <http://www.ecss.nl/>
- [18] The RODIN platform, <http://rodin-b-sharp.sourceforge.net/>

Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance

Andrew Ireland¹, Gudmund Grov², and Michael Butler³

¹ School of Mathematical & Computer Sciences, Heriot-Watt University,
Edinburgh, EH14 4AS, UK

A.Ireland@hw.ac.uk

² School of Informatics, University of Edinburgh, Informatics Forum,
Edinburgh, EH8 9AB, UK

ggrov@inf.ed.ac.uk

³ School of Electronics & Computer Science, University of Southampton,
Highfield, Southampton, SO17 1BJ, UK

mjb@ecs.soton.ac.uk

Abstract. The activities of formal modelling and reasoning are closely related. But while the rigour of building formal models brings significant benefits, formal reasoning remains a major barrier to the wider acceptance of formalism within design. Here we propose *reasoned modelling critics* – a technique which aims to abstract away from the complexities of low-level proof obligations, and provide high-level modelling guidance to designers when proofs fail. Inspired by proof planning critics, the technique combines proof-failure analysis with modelling heuristics. Here, we present the details of our proposal and outline future plans.

1 Introduction

The use of mathematical techniques for system-level modelling and analysis brings significant benefits, as well as challenges. While the rigour of mathematical argument can offer early feedback on design decisions, a key challenge centres on *how* feedback derived from the analysis of complex *proof obligations* (POs) can be used to improve design decisions. Such feedback currently requires user intervention, drawing upon skilled knowledge of the subtle interplay that exists between modelling and reasoning. For example, consider a simple cruise control system with the safety requirement that the brakes (*brake*) cannot be on while the cruise control (*cc*) is enabled. This can be formalised as the following invariant:

$$cc = on \Rightarrow brake = off$$

When the driver applies the brakes, i.e. $brake := on$ it must be shown that the safety requirement (invariant) is preserved, creating a PO of the form:

$$brake = off, cc = on \vdash on = off$$

Note that this PO is false. An (unsafe) solution to this failure would be to restrict the application of the brakes via a guard, i.e. only allow the brakes to be applied

when the cruise control is disabled ($cc = off$). Clearly, the desirable solution requires the introduction of an auto disable mechanism, i.e. the cruise control is disabled if the brakes are applied ($cc := off; brake := on$). While the two alternatives do not represent a significant burden to a designer, in general many solutions may arise from proof-failure analysis. In order to make good use of a designer’s time, modelling heuristics are needed in order to rank the alternatives.

This need for both proof and modelling heuristics is central to our proposal. We aim to identify common modelling and reasoning patterns, and use these to abstract away from complex POs. This will enable us to automatically provide designers with high-level decision support oriented towards modelling choices. In turn this will increase the productivity of designers as well as the accessibility of formal modelling tools. We plan to implement our proposal through what we call *reasoned modelling critics*.

A key inspiration for our proposal is *proof planning*, a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [6]. Central to proof planning is its proof-failure analysis and proof patching capabilities [10]. Our proposal aims to build directly upon these features. Specifically by combining the proof-failure analysis capabilities with modelling heuristics we aim to automatically generate modelling guidance. Our longer term aim is to combine the proof plan representation with a complementary notion of *modelling patterns*, which we call *reasoned modelling methods*. This, combined with reasoned modelling critics, are the building blocks of our notion of *reasoned modelling*: the study of the interplay between reasoning and modelling.

While our vision is generic, our starting point is Event-B, which we motivate and introduce together with proof planning in §2. §3 outlines the reasoned modelling critics mechanism and shows its application through examples. We discuss related and future work in §4 and conclude in §5.

2 Background

Event-B and the Rodin toolset: Event-B provides a formal framework for modelling discrete complex systems [1], and is mechanized through the Rodin toolset [2]. Event-B promotes an incremental style of formal modelling, where each step of a development is underpinned by formal reasoning. As a result, there is strong interplay between modelling and reasoning and this is partly supported by the Rodin toolset. This interplay requires skilled user interaction, i.e. typically a user will analyse failed proofs, and translate the analysis by hand into corrective actions at the level of modelling. This is exemplified in e.g. [18]. Typical corrective actions include strengthening invariants and guards or modifying actions. Our aim is to provide high-level decision support, by automating the generation, filtering and ranking of modelling suggestions. Event-B models and POs are closely aligned [1], whilst Rodin [2] is an extensible framework. Event-B and the Rodin toolset thus represent a unique opportunity for us to investigate reasoned modelling.

We will use the term *models* for both Event-B *contexts* (the static part) and *machines* (the dynamic part). We will only use the basic features of Event-B and use standard notation:

$$\begin{aligned} \mathbf{EVENT} \langle name \rangle &\hat{=} \mathbf{BEGIN} \langle general\ substitution \rangle \mathbf{END} \\ \mathbf{EVENT} \langle name \rangle &\hat{=} \mathbf{WHEN} \langle guard \rangle \mathbf{THEN} \langle general\ substitution \rangle \mathbf{END} \end{aligned}$$

where the event is only executed when the guard holds. Moreover, we use the term *action* for the generalised substitution, while INITIALISATION is a special event without guards defining the initial state.

Proof planning and proof critics mechanism: Proof planning builds upon the tactic-based tradition of theorem proving, where primitive proof steps are packaged-up into programs known as *tactics*. Starting with a set of general purpose tactics, plan formation techniques are used to construct a customised tactic for a given conjecture. The search for a customised tactic is constrained by a set of *methods*, collectively known as a *proof plan*. Using preconditions, each method specifies the applicability of a general purpose tactic. Having explicit preconditions provides insights when proof planning fails. The proof *critics* mechanism [10] enables “interesting” failures at the level of precondition evaluation to be represented. For a given method a number of critics will typically exist. Each critic represents an alternative generic proof patch, e.g. conjecture generalisation. The analysis of a specific failure, and subsequent proof planning, provides guidance in the selection and instantiation of a generic proof patch. In Fig 1 we present a method and critic. These are deliberately simple – their role is to illustrate the basic critics mechanism. Proof patching has been applied successfully to the problems of inductive conjecture generalisation and lemma discovery [5], as well as loop invariant discovery [11]. Proof critics have also been developed for *abductive reasoning*, i.e. the speculation of hypotheses in order to explain a consequence, and applied to the problem of patching faulty conjectures [12]. The proof critics mechanism provides the starting point for reasoned modelling critics. Moreover, in terms of exploiting existing critics, we see the work on abduction playing an important role in our reasoned modelling critics.

3 From Proof Critics to Reasoned Modelling Critics

As described above, our reasoned modelling critics will extend the existing proof critics mechanism. This extension will involve two key components:

1. Firstly, in order to combine proof and modelling, our new critics need access to models as well as POs. In addition, critics will now have the option of providing modelling guidance, as well as proof patches.
2. Secondly, both methods and critics are associated with preconditions. We need to extend the meta-language of critics to allow us to represent preconditions which specify modelling heuristics.

<p>method (rewrite)</p> <p>INPUT: $\Delta \vdash G$</p> <p>PRECONDITIONS:</p> <ol style="list-style-type: none"> 1. $exp_at(G, Pos, L)$ 2. $rewrite_rule(C \Rightarrow L := R)$ 3. $provable(\Delta \vdash C)$ <p>EFFECTS:</p> <p>$replace_at(Pos, R, G, NewG)$</p> <p>OUTPUT:</p> <p>$[\Delta \vdash NewG]$</p>	<p>critic (casesplit)</p> <p>INPUT: POs</p> <p>PRECONDITIONS:</p> <ol style="list-style-type: none"> 1. $\exists failed_po \in \{\langle rewrite, _ , PO \rangle \in POs \mid failed_proof(PO)\}$ $failed_po = \langle _ , Pre, _ \rangle$ 2. $\langle exp_at(_ , _ , _),$ $rewrite_rule(C \Rightarrow _ := _),$ $fail \rangle \in Pre$ <p>PATCH:</p> <p>$insert_casesplit(C, PO)$</p>
---	---

The meta-predicates used above are defined in Fig 3. Note that Δ denotes a list of hypotheses, while G and $NewG$ denote single goal formula. Note also the use of Prolog meta-variables, in particular the use of “_” for anonymous variables. The scope of meta-variables extends across all slots of the method and critic schemas.

Methods: A method takes as input a PO. The applicability of a method is determined by evaluating its associated preconditions. In the case of our example method, i.e. **rewrite**, there are three preconditions, i.e. i) there exists an expression L at position Pos within the goal G , ii) there exists a rewrite rule such that L matches the left-hand side of the rule, and iii) any condition C attached to the rule is provable within the proof context. If the preconditions of a method succeed, then the output POs (potentially empty) are computed via the effects slot. In the case of **rewrite**, the effects slot applies the selected rule, i.e. the expression at position Pos within the goal G is replaced by R (rule right-hand side) giving rise to new goal $NewG$ from which the output PO is constructed.

Critics: During proof planning, both the success and failure (*fail*) of a method’s preconditions are recorded. In particular, when a method fails a set of partial instantiations of the preconditions are associated with the PO. In the **casesplit** critic above, this set is denoted by Pre . The critics mechanism uses Pre to search for patchable exceptions to the method. If the pattern represented by a given critic’s preconditions matches with a recorded failure pattern then the associated patch is applied. This is illustrated above where the **casesplit** critic associates the failure (*fail*) of precondition 3 of the **rewrite** method with a casesplit patch. In general, a critic may require multiple failures in order to trigger a patch. In this way, critics can apply both local and global proof-failure analysis.

Fig. 1. An example proof method and critic

```

critic (name)
  INPUTS:
    PO_SET POs
    MODEL_SET Ms
  PRECONDITIONS:
    precondition 1
    ...
    precondition N
  OUTPUTS:
    PATCH proof patch (optional)
    GUIDE modelling guidance (optional)

```

The above schema represents our proposed refinement to the proof critic schema:

- The INPUT slot has been extended to include models, as well as proof obligations. Note that the schema allows for a critic to access multiple models. As a consequence, critics can be developed that consider the internal consistency of individual models as well as refinements and decompositions.
- The declarative PRECONDITIONS determine the applicability of the critic. Preconditions have access to proof obligations as well as models.
- The OUTPUT slot has been extended to include modelling guidance (GUIDE), as well as proof patches (PATCH). While proof patches are automatically applied, it is envisaged that modelling guidance will be communicated to the designer so as to inform their decision making.

As with the original proof critics mechanism, the output slot becomes instantiated as a side-effect of the instantiation of the input and precondition slots.

Fig. 2. Reasoned modelling critic schema

To support access to models and modelling guidance, an extended critic schema will be required. Our proposed extension is described in Fig 2. In order to illustrate the new schema and the kind of meta-language extensions we propose for reasoned modelling, we develop below some critics and show their application to simple control system problems.

3.1 Guidance for Invariant Proof Failures via Local Analysis

Typically, a proof failure arising from an inconsistency in a model may be overcome in a number of ways. For example, consider an invariant which takes the form of an implication, i.e. $X \Rightarrow Y$. If a corresponding invariant proof fails, then the failure may be overcome by revising the model so that either,

1. X is false, or
2. Y is true.

At the level of modelling these effects can be achieved by changing the guards or actions associated with the invariant proof, or the invariant itself. Here we

$disjoint_sub(S_1, S_2)$	$\hat{=}$ $dom(S_1) \cap dom(S_2) = \emptyset$
$failed_proof(P)$	$\hat{=}$ “ P is a PO which is provably false or has failed to be proven”
$provable(P)$	$\hat{=}$ “ P is a PO that is provable”
$max(A)$	$\hat{=}$ $\epsilon x. (x \in A \wedge \forall y \in A. y \leq x)$
$exp_at(X, Y, Z)$	$\hat{=}$ “ Z is the subexpression at position Y within expression X ”
$rewrite_rule(X \Rightarrow Y \Rightarrow Z)$	$\hat{=}$ “A conditional rewrite rule”
$pri_var(V, M)$	$\hat{=}$ “Priority of variable V within model M ”
$priority(S, M)$	$\hat{=}$ $max(\{pri_var(v, M) \mid v \in dom(S)\})$
$replace_at(W, X, Y, Z)$	$\hat{=}$ “ Z is obtained by replacing the subexpression at position W within Y by X ”
$sub2act(\{V \mapsto D\})$	$\hat{=}$ $V := D$
$sub2act(\{V_1 \mapsto D_1, \dots, V_n \mapsto D_n\})$	$\hat{=}$ $sub2act(\{V_1 \mapsto D_1\}) \parallel \dots \parallel sub2act(\{V_n \mapsto D_n\})$
$sub2grd(\{V \mapsto D\})$	$\hat{=}$ $V = D$
$sub2grd(\{V_1 \mapsto D_1, \dots, V_n \mapsto D_n\})$	$\hat{=}$ $sub2grd(\{V_1 \mapsto D_1\}) \wedge \dots \wedge sub2grd(\{V_n \mapsto D_n\})$
$guards(E)$	$\hat{=}$ “Conjunction of guards of event E ”
$sat(P)$	$\hat{=}$ “The predicate P is satisfiable”
$generalisable(H)$	$\hat{=}$ $\exists r, e_1, e_2, g_1, g_2. \langle e_1, g_1 \rangle \in H \wedge \langle e_2, g_2 \rangle \in H$ $\wedge e_1 \neq e_2 \wedge r \Rightarrow g_1 \wedge r \Rightarrow g_2 \wedge sat(r)$
$generalise(H)$	$\hat{=}$ “The weakest r such that $sat(r)$ and for the most $\langle e, g \rangle \in H, r \Rightarrow p$ (for at least 2 events)”
$add_guard(G, E, M)$	$\hat{=}$ “Adds guard G to event E of model M ”
$add_action(A, E, M)$	$\hat{=}$ “Adds action A to event E of model M ”
$add_invariant(I, M)$	$\hat{=}$ “Adds invariant I to model M ”
$insert_casesplit(C, P)$	$\hat{=}$ “Progress the proof of P by a casesplit on X ”

Note that S denotes a substitution, while H denotes a set of event-guard pairs.

Fig. 3. Meta-terms for reasoned modelling critics

focus on *local* changes to an event, i.e. changes to guards and actions, and delay the discussion of global analysis and invariant changes until §3.3. In general, proof-failure analysis will generate a large number of proof patches. But not all proof patches will make sense in terms of modelling. To address this problem we wish to exploit modelling knowledge in order to rank the proof patches that are offered as guidance. To achieve this we envisage designers providing meta-data in addition to their models. The preconditions of our new critics can then exploit modelling heuristics as well as proof-failure heuristics.

To illustrate this idea, we develop two critics below. Each critic targets a different pattern of invariant proof failure, as outlined above. In addition, the critics exploit a notion of variable priority – which is based upon the observation that not all variables within a model may carry equal importance. Crucially, such “meta-data” must be supplied by the designer. For example, being able to brake is clearly more important than driving with the cruise control enabled. This notion of *priority* can be expressed as a heuristic, and used to rank or even filter modelling suggestions. Informally, where proof-failure analysis suggests that the

critic (priority action speculation)

INPUTS:

PO_SET POs MODEL_SET $\{M\}$

PRECONDITIONS:

1. $\exists failed_po \in \{ \langle -, -, -, PO \rangle \in POs \mid failed_proof(PO) \}$.
 $failed_po = \langle M, E, -/INV, (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y)) \rangle$
2. $\exists \tau \in sub. disjoint_sub(\tau, \sigma) \wedge provable(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false)$
3. $priority(\tau, M) < priority(\sigma, M)$

OUTPUTS:

GUIDE $add_action(sub2act(\tau), E, M)$

Note that *sub* denotes the set of all substitutions, and elements of PO_SET are quadruples, i.e. model identifier, event identifier, PO label/type, and PO. Note also that the scope of the quantification extends across the critic slots.

Fig. 4. A reasoned modelling critic based on a priority heuristic to modify an action

value of a variable should be changed within the context of a given event, we adopt the following heuristics:

- H1:** If the priority of the candidate variable is lower than the priorities of all the variables updated by the event, then it is strongly suggestive that the change should be achieved via a new action.
- H2:** If the priority of the candidate variable is higher than the priorities of all the variables updated by the event, then it is strongly suggestive that the change should be achieved via a new guard.

Where priorities are the same, no ranking of the alternatives will be provided. We represent heuristics H1 and H2 as critics in Fig 4 and Fig 5 respectively. The meta-logical terms that appear within the preconditions of these critics are defined in Fig 3. Note that meta-logical predicates, such as *priority*, require input from designers, as mentioned above. Both critics are applicable where an invariant proof has failed, and specifically where the invariant takes the form of an implication. We focus on solutions that involve the addition of either guards or actions, where guards are restricted to equalities. Later we discuss how these basic critics could be generalised.

The critic representing H1 (Fig 4) has three preconditions. The first precondition holds if there exists a PO of the required form that has failed to be proved, i.e. an invariant PO of the form $X \Rightarrow Y$, which is associated with the event E where substitution σ represents the actions corresponding to E . The second precondition holds if there exists a substitution (τ) which falsifies the antecedent, where τ and σ are disjoint. The third precondition expresses a modelling constraint, i.e. the variable(s) introduced by the new substitution have lower priority than the variable(s) updated by the existing actions. If the preconditions hold, then the guidance provided takes the form of a guard – constructed from τ .

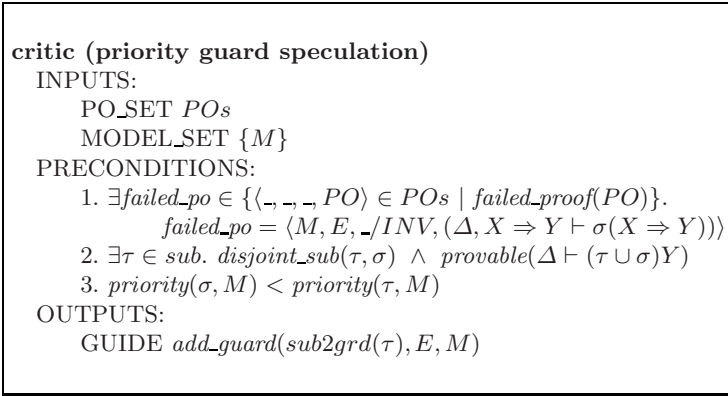


Fig. 5. A reasoned modelling critic based on a priority heuristic to modify a guard

The critic representing H2 (Fig 5) again has three preconditions, but illustrates a slightly different failure analysis pattern. That is, instead of making the antecedent false, the analysis tries to make the consequent true. Note that the third precondition ensures that the variable(s) introduced by the new substitution are of a higher priority than the variable(s) updated by the existing actions. Here the guidance provided takes the form of an action – constructed from τ .

3.2 The Cruise Control System in Event-B

To illustrate the application of critics introduced above, we will first formalise in Event-B the cruise-control system that was outlined in §II. Recall that the status of the brakes is represented by a variable *brake*, while the status of the cruise control is represented by the variable *cc*. Both variables are of type $Val = \{on, off\}$, where $on \neq off$. The variable *brake* is *on* if the user currently presses the brake (*pressbrake*), and *off* otherwise. While the variable *cc* is *on* if the cruise control is enabled (*enable_cc*), and *off* otherwise. The system invariant is represented as follows:

$$inv1 : cc = on \Rightarrow brake = off$$

and initially both variables are *off*:

$$INITIALISATION \hat{=} \mathbf{BEGIN} \textit{brake} := off \parallel cc := off \mathbf{END}$$

We assume that the following priority relationship between the variables has been specified by the designer:

$$pri_var(cc, \textit{cruise-control}) < pri_var(\textit{brake}, \textit{cruise-control})$$

critic (priority guard speculation)

INPUT

PO_SET pos MODEL_SET $\{cruise-control\}$

PRECONDITIONS:

1. $\langle cruise-control, enable_cc, inv1/INV, po \rangle \in pos \wedge$
 $po = (cc = on \Rightarrow brake = off \vdash \{cc \mapsto on\}(cc = on \Rightarrow brake = off)) \wedge$
 $failed_po((cc = on \Rightarrow brake = off \vdash brake = off))$
2. $disjoint_sub(\{brake \mapsto off\}, \{cc \mapsto on\}) \wedge$
 $provable(cc = on \Rightarrow brake = off \vdash \{brake \mapsto off, cc \mapsto on\}(brake = off))$
3. $priority(\{cc \mapsto on\}, cruise-control) < priority(\{brake \mapsto off\}, cruise-control)$

OUTPUTS:

GUIDE $add_guard(brake = off, enable_cc, cruise-control)$ **Fig. 6.** An instantiation of the ‘priority guard speculation’ critic

Failure and modelling suggestion 1: The model contains events that control the brakes and cruise control. The consistency proofs of invariant $inv1$ over these events fails in two cases. Firstly, the event to enable the cruise control is defined as follows:

$$\text{EVENT } enable_cc \hat{=} \text{BEGIN } cc := on \text{ END}$$

This gives rise to the following proof obligation:

$$cc = on \Rightarrow brake = off \vdash \{cc \mapsto on\}(cc = on \Rightarrow brake = off)$$

which can be reduced to the unprovable goal:

$$cc = on \Rightarrow brake = off \vdash brake = off.$$

However, the preconditions of the ‘priority guard speculation’ critic given in Fig 5 hold, and an instantiation of this critic for this example is shown in Fig 6 (after some simplification and unfolding). This instantiation suggests the addition of a new guard of the form $brake = off$. If the suggestion is accepted by the designer, then the updated event takes the form:

$$\text{EVENT } enable_cc \hat{=} \text{WHEN } brake = off \text{ THEN } cc := on \text{ END}$$

Failure and modelling suggestion 2: The second failure with respect to $inv1$ occurs when a driver presses the brakes, as defined by the event:

$$\text{EVENT } pressbrake \hat{=} \text{BEGIN } brake := on \text{ END}$$

This creates the following proof obligation:

$$cc = on \Rightarrow brake = off \vdash \{brake \mapsto on\}(cc = on \Rightarrow brake = off)$$

critic (priority action speculation)

INPUT

PO_SET pos MODEL_SET { *cruise-control* }

PRECONDITIONS:

1. $\langle \textit{cruise-control}, \textit{pressbrake}, \textit{inv1}/INV, po \rangle \in pos \wedge$ $po = \left(cc = on \Rightarrow brake = off \vdash \{ brake \mapsto on \} (cc = on \Rightarrow brake = off) \right) \wedge$ $failed_po((cc = on, brake = off \vdash false)$ 2. $disjoint_sub(\{cc \mapsto off\}, \{brake \mapsto on\}) \wedge$ $provable(cc = on \Rightarrow brake = off \vdash \{brake \mapsto on, cc \mapsto off\}((cc = on) \Rightarrow false))$ 3. $priority(\{cc \mapsto off, \textit{cruise-control}\} < priority(\{brake \mapsto on\}, \textit{cruise-control})$

OUTPUTS:

GUIDE $add_action(cc := off, \textit{pressbrake}, \textit{cruise-control})$ **Fig. 7.** An instantiation of the ‘priority action speculation’ critic

which, when simplified, becomes false:

$$cc = on, brake = off \vdash false.$$

In this case the ‘priority action speculation’ critic given in Fig 4 triggers, and Fig 7 shows the corresponding instantiation. As a result, the following updated *pressbrake* event is suggested to the designer:

EVENT *pressbrake* $\hat{=}$ **BEGIN** *brake := on* || *cc := off* **END**

Generalisations of the critics: An alternative definition of the invariant is

$$inv2: brake = on \Rightarrow cc = off$$

which is the contrapositive of *inv1*. To keep our presentation as simple as possible, our critics will not trigger on POs arising from *inv2*. However, this limitation could be overcome by generalising the failure analysis, i.e. allowing the critics to consider the alternatives of falsifying an antecedent and validating a consequent. This could be achieved by changing precondition 2 as follows:

$$provable(\Delta \vdash (\tau \cup \sigma)X \Rightarrow false) \vee provable(\Delta \vdash (\tau \cup \sigma)Y)$$

Another limitation is that currently we only consider the addition of guards that take the form of equalities. The ‘priority guard speculation’ critic could be extended to capture a more general notion of a guard. However, this would require a new notion of priority, which is currently not defined for arbitrary terms (only variables and substitutions). Finally, global consistency issues have been ignored. When an action is modified, the validity of other invariants may change, or a deadlock may arise. The critics should be updated with such consistency checks. Depending on how global analysis is dealt with – as discussed below – the ‘priority action speculation’ critic of Fig 4 may have to also include such a *consistency* predicate.

3.3 From Local to Global Modelling Suggestions

As mentioned earlier, we envisage that our critics will support global analysis, as well as the local analysis illustrated via the cruise control example. For instance, when attempting to prove the internal coherence of a machine, local analysis may suggest the need to add guards across multiple events. Taking a more global perspective, these local suggestions may reveal a more general modelling suggestion. For instance, if a formula can be found that is logically weaker than all the suggested guards, then the formula could be suggested as a new invariant.

A concrete example of this kind of global analysis is found in Abrial’s ‘Cars on a Bridge’ example [11]. In this example a single-laned bridge between an island and the mainland is modelled. Since it is single-laned, cars can only travel in a given direction at a given time – and this is controlled by two traffic-lights: m_tl is the traffic light on the mainland; and i_tl is the traffic light on the island. The lights can have two colours – *green* and *red* – and it is assumed that $green \neq red$. Our discussion will focus on two events: IL_out , where cars enter the bridge from the island; and ML_out , where cars enter the bridge from the mainland. Without giving all the details [2], two different invariant proofs fail for these events. The corresponding POs have the following form:

$$\dots, i_tl = green \vdash \sigma_1(m_tl = green \Rightarrow \dots) \quad \text{where } m_tl \notin dom(\sigma_1) \quad (1)$$

$$\dots, m_tl = green \vdash \sigma_2(i_tl = green \Rightarrow \dots) \quad \text{where } i_tl \notin dom(\sigma_2) \quad (2)$$

A local analysis of each of these failures suggests making the antecedent false, by finding a substitution τ such that:

$$disjoint_sub(\tau, \sigma_{\{1,2\}}) \wedge provable(\Delta \vdash (\tau \cup \sigma_{\{1,2\}})X \Rightarrow false)$$

In (1), τ will be instantiated to $\{m_tl \mapsto red\}$ while in (2), τ will be instantiated to $\{i_tl \mapsto red\}$. However, there is a common solution to these failures. Firstly, we assume that the substitutions are turned into guards (by *sub2grd*). Now, if an event is constrained by the guards G_1, \dots, G_n , then adding an additional guard G_{n+1} to patch a failure is equivalent to adding the weaker guard $G_1 \wedge \dots \wedge G_n \Rightarrow G_{n+1}$ (via modus ponens). Thus the suggested additional guards can be “weakened” in this way with the existing guards of the events. For this particular example, we ignore all guards except those shown in (12) – the other guards are irrelevant for this failure. For example, from (1) the guard $m_tl = red$ is derived and “weakened” to give $i_tl = green \Rightarrow m_tl = red$. We then try to find a common solution for (12), i.e. an r such that:

$$\begin{aligned} r &\Rightarrow i_tl = green \Rightarrow m_tl = red \\ r &\Rightarrow m_tl = green \Rightarrow i_tl = red. \end{aligned}$$

¹ The example is modified slightly with respect to types.

² These details can be found in [1] and <http://www.event-b.org>.

Moreover, we require that r is satisfiable, i.e. the trivial solution $r = \text{false}$ is not considered. The only valid such generalisation r (modulo equivalences) is:

$$il_tl = \text{red} \vee ml_tl = \text{red}$$

which is suggested as an invariant. This represents a key safety requirement – at all times at least one traffic-light is red.

critic (multiple failure invariant speculation)

INPUTS:

PO_SET POs

MODEL_SET $\{M\}$

PRECONDITIONS:

1. $\exists h \subseteq hs. h = \{(E, G) \mid \exists po \in pos. \langle _, E, _ / INV, po \rangle \in POs \wedge \text{failed_proof}(po) \wedge po = (\Delta, X \Rightarrow Y \vdash \sigma(X \Rightarrow Y)) \wedge \exists \tau \in sub. \text{disjoint_sub}(\tau, \sigma) \wedge \text{provable}(\Delta \vdash (\tau \cup \sigma)X \Rightarrow \text{false}) \wedge G = \text{guards}(E) \Rightarrow \text{sub2grd}(\tau) \}$
2. $\text{generalisable}(h) \wedge \exists i \in forms. i = \text{generalise}(h)$

OUTPUTS:

GUIDE $\text{add_invariant}(i, M)$

Note that hs denotes the set of all event-guard pairs, while pos and $forms$ denote the sets of all proof obligations (sequents) and formulae respectively.

Precondition 1 holds if a local analysis suggests that proof failures can be overcome by the introduction of additional guards.

Precondition 2 holds if the suggested guards can be generalised to give an invariant.

Fig. 8. A global reasoned modelling critic suggesting an invariant from multiple failures

With regards to realizing such global reasoning, we are considering two distinct alternatives via our proposed critic schema:

1. Combine both local and global analysis of the available POs within a single critic, as illustrated in Fig 8.
2. Firstly use critics to perform local analysis of the available POs, recording any modelling suggestions. Secondly, use separate critics to perform a global analysis, exploiting the results of the local analysis.

From a scientific point of view, the approaches are not very different – and should be seen more as an implementation issue. Experimentation will be required in order to determine the most practical approach.

4 Related and Future Work

Our notion of *reasoned modelling* represents a new paradigm for exploring the interplay between reasoning and modelling. As evident from the above discussion, our general ideas on reasoned modelling are strongly influenced by the proof

planning paradigm [6], while the particular work discussed here follows from the notion of *proof critics* [10]. Currently, Event-B users have to manually analyse failed proof attempts and patch their models, e.g. [13]. This form of user interaction represents a significant barrier to the accessibility of the Event-B toolset.

Previously, the application of *design patterns* [9] has been suggested for Event-B [3]. The ability to reuse design patterns, and their associated proofs, has obvious benefits over the conventional notion of design patterns. Our work with failure-analysis is more closely aligned with *anti-patterns* [4], i.e. common patterns of bad design, coupled with solutions. Bad design, however, does not necessarily mean incorrect design – i.e. the POs may still be provable. For this reason the failure-driven nature of the critics presented here differ from the conventional notion of an anti-pattern. Another related area is ontology repair plans [7] – proof failure is explored using proof planning techniques to repair ontologies – i.e. proof planning is used to describe evolving models.

In terms of future work, our short-term aim is to prototype and further develop the proposal presented here – in particular, investigate modelling heuristics in addition to variable priorities. This will involve developing a plug-in for the Rodin toolset. The core of this plug-in will most likely be implemented in Prolog or Ocaml, both of which are well suited to the development of automated reasoning techniques. In terms of proof, we will initially rely on the existing Rodin theorem provers. However, in the longer term we aim to incorporate a proof planner. This will enable us to also develop critics that analyse successful proofs and provide guidance along the lines of conventional anti-patterns.

5 Conclusion

We have motivated the need to abstract away from the complexities of proof obligations and proof-failure analysis. To address this need, we have proposed a technique of proof management that attempts to turn proof-failure analysis into modelling guidance via *reasoned modelling critics*. In doing so we aim to increase the productivity of designers as well as the accessibility of formal modelling. Our technique builds upon proof planning and, in particular, proof critics. The proposal differs from previous work in that it provides a uniform framework in which proof-failure heuristics can be combined with modelling heuristics. It is this combined approach that will enable us to abstract away from the complexity of proof obligations. While our initial critics have focused on variable priorities, our framework will be extensible, allowing designers to record meta-data relating to other kinds of modelling decisions. A prototype in the form of a Rodin plug-in is currently under development, this will enable us to empirically test and further develop our technique.

Acknowledgements. This research is supported by EPSRC grants EP/F037058 and EP/E005713. Butler’s involvement is part of the EU research project ICT 214158 DEPLOY (www.deploy-project.eu). Our thanks go to Alan Bundy,

who motivated the use of variable priorities. Thanks also go to Jean-Raymond Abrial, Cliff Jones, Ewen Maclean and Maria Teresa Llano Rodriguez for their feedback and encouragement with this work. Finally we thank the three anonymous ABZ 2010 reviewers for their constructive feedback.

References

1. Abrial, J.-R.: *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2009) (To be published)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Abrial, J.-R., Hoang, T.S.: Using Design Patterns in Formal Methods: An Event-B Approach. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) *ICTAC 2008*. LNCS, vol. 5160, pp. 1–2. Springer, Heidelberg (2008)
4. Brown, W., Malveau, R., McCormick, H.W.S., Mowbray, T.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Chichester (1998)
5. Bundy, A., Basin, D., Hutter, D., Ireland, A.: *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, Cambridge (2005)
6. Bundy, A.: The Use of Explicit Plans to Guide Inductive Proofs. In: Lusk, R., Overbeek, R. (eds.) *CADE 2009*, pp. 111–120. Springer, Heidelberg (1988)
7. Bundy, A., Chan, M.: Towards ontology evolution in physics. In: Hodges, W., de Queiroz, R. (eds.) *Logic, Language, Information and Computation*. LNCS (LNAI), vol. 5110, pp. 98–110. Springer, Heidelberg (2008)
8. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. *Formal Aspects of Computing* 20(1), 61–77 (2008)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
10. Ireland, A.: The Use of Planning Critics in Mechanizing Inductive Proofs. In: Voronkov, A. (ed.) *LPAR 1992*. LNCS, vol. 624, pp. 178–189. Springer, Heidelberg (1992)
11. Ireland, A., Stark, J.: Proof Planning for Strategy Development. *Annals of Mathematics and Artificial Intelligence* 29(1-4), 65–97 (2001)
12. Monroy, R., Bundy, A., Ireland, A.: Proof Plans for the Correction of False Conjectures. In: Pfenning, F. (ed.) *LPAR 1994*. LNCS (LNAI), vol. 822, Springer, Heidelberg (1994)

Applying the B Method for the Rigorous Development of Smart Card Applications*

Bruno Gomes¹, David Déharbe¹, Anamaria Moreira¹, and Katia Moraes²

¹ Federal University of Rio Grande do Norte (UFRN), Natal, RN, Brazil
{bruno,david,anamaria}@consiste.dimap.ufrn.br

² Petróleo Brasileiro S.A. (PETROBRAS), Rio de Janeiro, RJ, Brazil
katicaka@yahoo.com.br

Abstract. Smart Card applications usually require reliability and security to avoid incorrect operation or access violation in transactions and corruption or undue access to stored information. A way of reaching these requirements is improving the quality of the development process of these applications. BSmart is a method and a corresponding tool designed to support the formal development of the complete Java Card smart card application, following the B formal method.

1 Introduction

Smart card applications are present in our everyday life in a wide range of sectors such as banking and finance, communication, Internet, public transport, health care, etc. These applications are stored in a resource constrained device and usually manage confidential information, such as bank account data, the medical history of a patient or user authentication data.

Java Card [1] is a version of the Java platform with a restricted API and Virtual Machine optimized for smart cards and other memory and processor constrained devices. The Java Card developer can benefit from most of the Java features, such as portability, type-safe language, object oriented development, and the available tools.

To prevent undesirable behavior and to avoid security violations, it is helpful to improve the quality of the smart card development process with the adoption of rigorous software engineering process, methods and tools, to ensure that the final product is in conformance with the specified requirements.

The BSmart project aims to contribute with a method and its corresponding tool support which support the formal development of Java Card application services through a development process that starts from a platform independent B Specification of these services (i.e., a specification where Java Card characteristics are not specified). The application installed on card, which contains the

* This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) www.ines.org.br, funded by CNPq grant 573964/2008-4 and by CNPq grant 553597/2008-6. The author Bruno Gomes is supported by a doctoral degree scholarship from CNPq.

implementation of the services accessed by the client, is developed following a customized B refinement/implementation process of the initial specification. For the client (host application) we provide the generation of an API to communicate with the applet that takes care of all the work related with Java Card protocol aspects and transparent Java/Java Card data coding/encoding.

Previous work [2] [3] introduced the basic structure of the *BSmart* method. Its tool support has been also presented as a short paper in [4]. Here we describe the work current stage, including the API generation for the host side and the developed library machines that may be used in the process of specification, verification and code generation of these applications. With respect to these previous works, the current paper contributes with a more complete support and formalization.

This paper is organized as follows. The Section 2 introduces the Java Card system and its applications. An overview of the process is presented in Section 3. The host API generation and the development of the card application are discussed respectively in Sections 4 and 5. The tool support and the developed libraries are subject of the Section 6. Finally, in Section 7 we present some final considerations and related work.

2 Java Card

Java Card [1] is a restricted and optimized version of the Java platform to allow memory and processor constrained devices, such as smart cards, to store and run small applications. A developer can benefit of many Java features, such as portability, type-safe language, object oriented development, and available tools. This infrastructure allows a rapid application build, test and installation cycle, reducing the time and the cost of software production. However, in face of hardware limitation, Java Card presents some restrictions on resources and API. For example, dynamic class loading, threads, *Strings*, the types *float* and *double*, and multi-dimensional arrays are not present in the current versions of Java Card. The integer type (*int*) and garbage collection are optional.

The main component of the Java Card platform (Fig. 1) is its runtime environment (JCRE), composed of a Java Card Virtual Machine (JCVM), a small API, and, usually, system and industry-specific classes [1]. The JCRE acts as a small operating system, being responsible for the control of the application lifetime, security and resource management.

In this work we are interested in the Java Card software development, presented in Section 2.2. However, before going into the software details, it is helpful to understand some aspects of the smart card system in Section 2.1.

2.1 Smart Card System

A smart card application is distributed between on-card and off-card components. The server application on the card side (called applet) provides the application services and is installed in the smart card EEPROM memory. The off-card

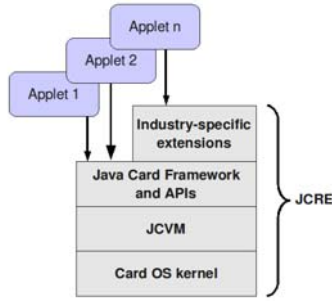


Fig. 1. Java Card platform components. Adapted from: [5].

client (called host application) resides in a computer or electronic terminal. A hardware device, named Card Acceptance Device (CAD), provides power to the card chip and the physical means that the applications uses to communicate [5]. The communication can be processed by electrical contact, when working with contact cards, or by radio frequency for contactless cards.

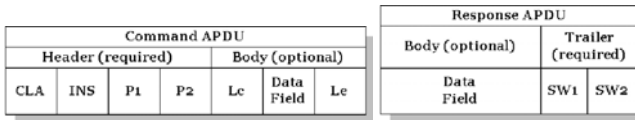


Fig. 2. Command and response APDUs. Source: [5].

The information exchange between the host and card applications is made through a half-duplexed low level communication unit, named *Application Protocol Data Unit* (APDU). The ISO 7816-4 standard specifies two kinds of APDU's, which are the command and the response APDU (Fig. 2). Both specify data packets. A command APDU is sent by the host, requiring some applet service and a response APDU is sent by the applet, responding to the host request with the result of the service processing. Examining Figure 2, one can note that the APDU packets are a low-level structure, requiring information coding (instruction code, data, etc.) when requesting a service and information decoding (data, status), after the service execution.

2.2 Developing Java Card Applications

The complete development process of a Java Card application involves essentially the (i) development of the card side applet and some auxiliary classes when needed, (ii) test and simulation of the developed application, (iii) conversion of the generated bytecode into an appropriate format to be installed on a smart card, and (iv) development of the client side host application. The activities (i) and (iv) are the focus of our work. In the following we give more information on the host application and on the Java Card applet.

Java Card host application: Java Card allows the inter-operability of the developed applet among different smart cards with compatible Java Card specifications. However, complete compatibility can only be obtained when the whole smart card environment, including cards, readers, protocols, and host applications are in accordance to common standards. To achieve this goal, some initiatives taken by a consortia of smart card companies emerged, such as PC/SC [6] and Global Platform [7]. These standards define complete APIs to initialise terminals and cards and to manage the communication through APDU encoding, taking in consideration not only on-card security, but global system security aspects. Other recent API, part of Java 6, is the Smart Card I/O. It is simpler than the others, but is compatible with PC/SC readers and is suitable to most application needs, offering the basic structure for applet communication. A portable host application must then use one of these standards in its implementation.

Java Card applets: A Java Card applet is a class that inherits the *javacard.framework.Applet* class of the Java Card API and is implemented upon the Java Card subset of Java. During the applet conversion for card installation, a verification phase is performed to check conformance of the classes with Java Card restrictions.

The current usual Java Card specification (2.2.x) allows two kinds of applets. The older, and most commonly used, kind of applet manipulates directly the APDU packages while the newer one abstracts from the lower level protocol using Remote Method Invocation (RMI). In this paper we will call the lower level applet *APDU applet* and the higher level one, *RMI applet*. RMI introduces a layer of abstraction above the APDU protocol, and, due to this fact, it is usually less efficient than APDU applets.

3 Formal Java Card Development with the B Method

Smart card applications usually require the management of confidential information, such as monetary values and data for secure authentication. Thus, a major part of smart card development is devoted to the implementation of an API containing the services provided by the card whilst protecting these data. Being a restricted domain of usually small applications with the need for safety and correctness, smart card applications represent a suitable domain to the adoption of formal development methods.

In this work we present a method to develop a Java Card application following the B method [8], from specification to refinement and code generation. By using B in the whole development process we aim to guarantee the preservation of the specified functional properties from the abstract specification modules until their implementation in Java Card.

Starting from a high-level formal specification of the API, the development (Fig. 3) follows two lines, one for the card side application and the other related to the host application:

host-side: The full automatic generation of an API that encapsulates the communication between the host and card applications, hiding this communication, as well as most of the details of the standards cited on section 2.2, from the developer, who can focus on the functional aspects of the application.

card-side: The development is based on refinement of the initial B specification. It progressively adds Java Card related aspects and a more concrete representation of the application towards its implementation. At the end of this chain of refinements, we are able to generate the implementation of the application.

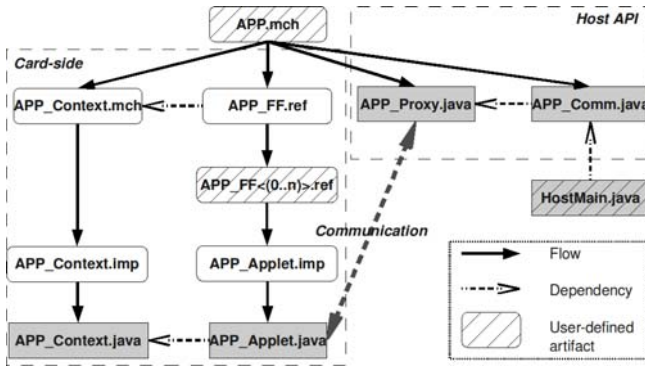


Fig. 3. A view of the BSmart method development and its artifacts

Figure 3 gives a general idea of the artifacts that appear in the development process. Most of the development can be performed using the support of tools (Section 6), but, as usual in B, the developer can add refinement levels to manage complexity or to reduce proof effort.

The Host API component encapsulates communication standards and protocols, so that the application is able to use the card services through the API method calls. In the next sections we detail the aspects related to the generation of the host side API and the development of the card-side application.

4 Generation of the Host Application API

As explained in Section 2.2, for the development of the host application there are some standards which define APIs to supply all the necessary resources to establish the communication with the card. In this work we propose to generate a set of classes (Fig. 4) to transparently communicate with the card application, using one of that APIs, freeing the user of the tedious and error-prone task of manipulating the lower-level details of the Java Card system. This encapsulation includes the management of the connection with a card application and the coding and decoding of the data sent to and received from the applet. The user

keeps the responsibility for the functional aspects of the application only, leading to an increased productivity.

The code of the API components for the host-side is generated from the original specification in a fully automatable process, since, in addition to the desired communication standard API to be used, we only need to know the expected services of the applet and the necessary data to process them. As the B method imposes that the signature of each operation does not change in the refinement process, we can obtain it directly from the operations of the high-level machine.

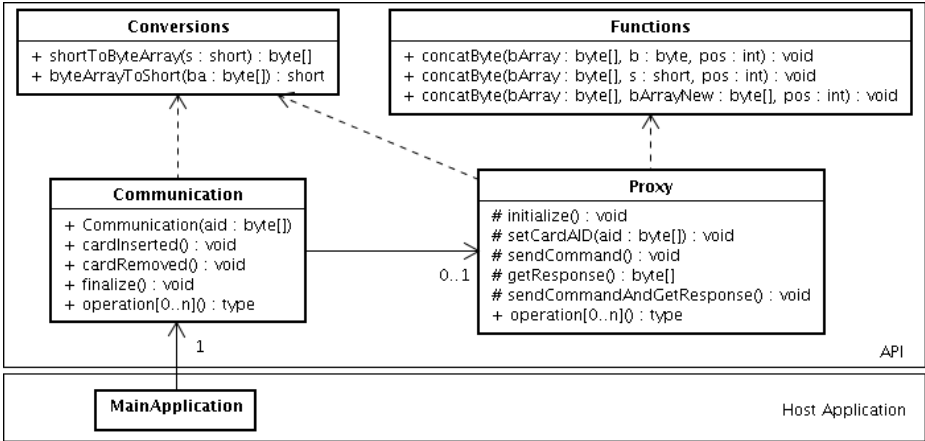


Fig. 4. API structure

As shown by the diagram of Figure 4, two main classes are generated for the host developer, named *Communication* and *Proxy*, with implementations that may vary depending on the kind of Java Card Applet (APDU or RMI) and communication standard. The *Communication* class is seen by the host application and contains high-level Java methods to call each applet service and to control the life-time of the applet. For APDU applets, the service calls are dispatched to the *Proxy* class, which is responsible for coding the received data into a packet in the command APDU format (Fig. 2) and send it to the applet. This class also decodes the returned response, sending it back to the *Communication* class. The *Proxy* class is not necessary for the generation of the RMI host application API since it allows the communication to be taken at a higher level than the APDU level.

The classes *Conversions* and *Functions* contain useful methods to help in the conversion tasks of data between the card and host applications. These are predefined library classes that can be reused in different developments.

Until now, we have implemented the generation for the APIs *OpenCard framework*, *Smart Card I/O* and RMI Client. The last is a simple API, part of the Java Card specification, for the communication with RMI applets.

4.1 Rules for Host API Code Generation

This section briefly describes the rules for the generation of the API classes, namely the rules associated with data emission and reception in the communication process. The set of translation rules is too large to be fully described here; they define a translation that follows the syntactical structure of the B specification. Most of them are straightforward, and the only complication occurs in the case of APDU applets due to the encoding and decoding of data when passed to the communication medium between the card and the host components.

In all cases, each specified operation is mapped into a method of the generated class. In the case of RMI applets, where the data encoding/decoding is taken care by the Java Card RMI library, the general rule for the translation of a B operation is:

```

generation_from(B_operation) =
let
  jres = java_return_type_of (B_operation)
  name = java_name_of (B_operation)
  jparams = java_param_list_of (B_operation)
  jthrows = (exception throwing declarations: depends on API)
  init_try = "try {"
  end_try = "} catch (Exception e) { e.printStackTrace(); }"
in
  "public" jres name "("jparams")" jthrows "{"
    init_try
      javarmi_app_method_call_of (B_operation)
    end_try
  "}"

```

The above rules rely on auxiliary functions, such as *java_return_type_of*, *java_param_list_of* and *java_name_of* that, given a B operation, respectively yield the return type, the list of parameters and the name of the corresponding Java method. The function *javarmi_app_method_call_of* uses these predefined functions to make the operation calling statement.

For the APDU version, the *Proxy* class is responsible for the encoding of data for the APDU buffer and the corresponding decoding. These rules are more complex than that for RMI as the arguments and the result of any method call need to be encoded as bytes and stored in an APDU packet. The APDU format has three fields to store data (see Fig. 2): *p1* (1 byte), *p2* (1 byte) and *data* (arbitrary size). We have thus defined and implemented a conversion algorithm that computes the number of bytes needed to encode the method arguments, and generates the Java code to convert such arguments to bytes and store them in the APDU packet. The code thus generated is optimized to reduce communications as it first fills the *p1* and *p2* fields, and uses the *data* field only when the encoding of the arguments is larger than two bytes.

5 Development of the Card-Side Application

As we introduced in Section 3, the formal development of the card applet starts from a high-level B specification of the application. This specification does not need to observe Java Card aspects. To illustrate some concepts of this branch of the development, let us introduce a simple abstract specification of a *counter* (Fig. 5) with an operation to increment the counter. This specification uses a Java compatible int type (JINT), provided in the *JInt* library machine (Fig. 6), which also defines arithmetic operations restricted to the range of the type, such as *sum_jint* to compute the sum of its two integer arguments and *gt_jint*, which returns *true* when its first argument is greater than the second one.

```

MACHINE JCounter
SEES JInt
VARIABLES value
INVARIANT  $value \in JINT$ 
INITIALISATION
   $value := 0$ 
OPERATIONS
  increment ( vv ) =
  PRE
     $vv \in JINT \wedge$ 
     $gt\_jint(vv, 0) = true \wedge$ 
     $sum\_jint(value, vv) \in JINT$ 
  THEN
     $value := sum\_jint(value, vv)$ 
  END
END

```

```

MACHINE JCCounter
SEES JCInt, JInt, (...), InterfaceContext
VARIABLES jc_value
INVARIANT  $jc\_value \in JCINT$ 
INITIALISATION
   $jc\_value := jcint\_of\_jint(0)$ 
OPERATIONS
  jc_increment ( vv ) =
  PRE
     $vv \in JCINT \wedge$ 
     $gt\_jcint(vv, jcint\_of\_jint(0)) = true \wedge$ 
     $sum\_jcint(jc\_value, vv) \in JCINT$ 
  THEN
     $jc\_value := sum\_jcint(jc\_value, vv)$ 
  END
END

```

Fig. 5. JCounter machine and its Java Card version JCCounter

Here we introduce a common situation that can occur in a typical development: the abstract specification uses types that are not compatible with Java Card types. The *JCounter* abstraction uses the *int* type, which is not built-in Java Card, and the B method does not allow interface and type changing in the refinement process. Thus, to follow the card-side development, we need to deal with this type incompatibility by introducing a refinement pattern.

To achieve the goal of type/interface adaptation without going out the strict rules of B refinement we use library machines that model each type and intermediate machines containing conversion functions and properties relating them. We will present here the general notion and excerpts of the corresponding modules to illustrate the solution for our example in which we model the *int* type through a pair of *shorts*. The detailed approach is described in 9. The important point in favor of this solution is that the development rigorously obeys B refinement restrictions.

```

MACHINE InterfaceContext
SEES JInt , JCInt
CONSTANTS jint_of_jcint, jcint_of_jint
PROPERTIES
  jint_of_jcint  $\in$  JCINT  $\rightsquigarrow$  JINT  $\wedge$ 
  jcint_of_jint  $\in$  JINT  $\rightsquigarrow$  JCINT
ASSERTIONS
  jint_of_jcint-1 = jcint_of_jint  $\wedge$ 
  dom ( jint_of_jcint ) = JCINT  $\wedge$ 
  dom ( jcint_of_jint ) = JINT
END

```

Fig. 6. The *InterfaceContext* machine

The right side of Figure 5 presents a counter machine providing the same services as *JCounter* but with interface and typing restrictions compatible with Java Card. The *JCINT* is a definition of the Java integer type represented as a pair of *shorts*, included in the *JCInt* library machine (not detailed here). The *JCCounter* machine is also the initial model of a B development to provide an implementation of the card-side component.

The conversion function linking these two abstract (*JINT*) and concrete (*JCINT*) integer representations are put in *InterfaceContext*. This machine also contains some corollaries in the assertions clause. These additional properties are useful to simplify interactive proofs of the development. Since *JCCounter* is a machine, not a refinement, we want to be able to prove that the type adaptation succeeds as a refinement relation. We can do this by refining the abstract *JCounter*, relating the abstract and concrete types in its invariant using the functions defined in *InterfaceContext*, as we can see in Figure 7. So, in case of successful verification, we are able to continue our card development using the concrete *JCCounter*. Although the process of typing adaptation is automatable (possibly with some user assistance), the tool support for the method does not include it yet.

To allow the smart card implementation of the API, an additional refinement is applied to make it closer to Java Card code. We achieve this making it full-function, i.e., weakening the preconditions of the operations so that they only define typing of the parameters. The remaining restrictive conditions are handled in its body through conditional substitutions, whose non-validity leads to the throwing of an exception. This is performed by modeling simple Java Card exception classes in an *Exception* library machine. A dedicated context machine contains the identifier code of each exception and any other constants or Java Card related information that the refinement needs. The full function refinement of *JCounter* and its context machine can be shown in Figure 8. The restrictive precondition stating that the value of the increment must be greater than zero was moved to its body, this way allowing the generation of this verification condition in the translated Java Card code.


```

REFINEMENT JCounter_ref
REFINES JCounter
SEES JInt, JCIInt, InterfaceContext
INCLUDES JCCounter
INVARIANT value = jint_of_jcint (jc_value)
OPERATIONS
  increment ( vv ) =
  PRE
    vv ∈ JINT ∧
    sum_jint(jint_of_jcint (jc_value), vv) ∈ JINT
  THEN
    jc_increment (jcint_of_jint (vv))
  END
END

```

Fig. 7. A refinement of JCounter to verify the type adaption correctness

<pre> MACHINE <i>JCCounterContext</i> SETS <i>EXCEPTIONS = {non_positive_value}</i> END REFINEMENT <i>JCCounterFF_ref</i> REFINES <i>JCCounter</i> INCLUDES <i>ISOException.Exception(EXCEPTIONS)</i> SEES <i>JCCounterContext, (...)</i> VARIABLES <i>jc_value</i> INVARIANT <i>jc_value</i> ∈ <i>JCINT</i> INITIALISATION <i>jc_value := jcint_of_jint (0)</i> OPERATIONS </pre>	<pre> jc_increment (<i>vv</i>) = PRE <i>vv</i> ∈ <i>JCINT</i> ∧ <i>sum_jcint (jc_value, vv)</i> ∈ <i>JCINT</i> THEN IF \neg(<i>gt_jcint(vv, jcint_of_jint(0))=</i> <i>true</i>) THEN ISOException.throwIt(<i>non_positive_value</i>) END; <i>jc_value := sum_jcint (jc_value, vv)</i> END END </pre>
--	---

Fig. 8. The full function version of JCCounter and its context machine

The translation of the B0 implementations to Java Card code is the last stage in the card-side development. The main development implementation, containing the services offered to the host, generates the applet class. As usual, other modules may have to be generated, such as the context machine. In Section 6.1 we can see part of the counter implementation, emphasizing the use of the APDU library machine for data sending and receiving.

6 Tool Support

The BSmart tool [4] is an Eclipse plugin connecting several software components, each responsible for implementing a different step of the BSmart method. Essential software for the B formal method is also included, such as a type checker,

and connection with external tools, such as Atelier B, for proof obligation generation and verification. Also, as explained in the next Subsection, we supply jointly with the tool a library of B modules modeling essential classes of the Java Card API, types and useful data structures.

The main components that provide support for the method are the *BSmart Modules Generator* and the *B to Java Card code translator*. The former is responsible to generate the B refinements required by the method and the latter translates all the B implementation modules into Java Card programming code and also generates the API classes for the host side client. The translator has been developed based on the Java translator of JBtools [10]. We modified this open-source B method tool to allow the translation for Java Card.

6.1 A Library of Reusable B Components

We have developed B machines to model Java/Java Card primitive types and some classes of the Java Card API. We plan to supply these verified B models to all classes of the Java Card API and to other useful tasks for Java Card applications, such as manipulation of time, date, currency, etc.

The specification of the Java Card API specification was realized using as basis the official documentation of the classes, as well as JML-based and OCL-based specifications [11,12]. In our approach, the specification serves for the purpose of: (i) providing verified B models of the API (ii) using these verified modules in the refinement of the card-side application, allowing us to verify the correctness of its use in relation to adequate data and the necessary dependencies, for example, when an operation requires the calling of a previous one and (iii) facilitating the generation of the Java Card code, since an operation of an API model is translated to its corresponding method call in Java Card.

As an example, we can see in Figure 9 some excerpts of the APDU class model in B, one of the most important of the Java Card API. Through it one can access the APDU buffer for exchanging data with the host application. The details of this process is treated internally with the Java Card Runtime Environment (JCRE) and it is not our propose to model it. As we said before, we are interested in the practical use of the operations to allow verification and code generation. On the right side of the figure, we show a practical example of the use of APDU machine in the implementation of the *counter* development. The APDU machine is imported and the operations are called as we do in a Java Card applet method to receive data and to send it after processing.

In the case of the types library we have developed machines to deal with the types *short*, *int*, *boolean* and a module to represent the type *int* for Java Card as a pair of *shorts*. As we can see for the type *int* in Figure 10, each machine has constants for type definition and useful functions to operate within the bounds of the type.

The development of the library is still in progress but we expect that when concluded the developed modules can be reused by B specifiers and Java Card developers with the advantage of being fully verified using the B method. We

```

MACHINE APDU (...)
CONCRETE_VARIABLES (...)
  state, buffer
INVARIANT
  state ∈ TBYTE ∧
  state ∈ ST_INITIAL ..
    ST_FULL_OUTGOING ∧
  buffer ∈ (0 .. 132) → TBYTE (...)
OPERATIONS
  res ← setIncomingAndReceive =
PRE
  state = ST_INITIAL
THEN
  CHOICE
  state := ST_PARTIAL_INCOMING
  OR
  state := ST_FULL_INCOMING
END ||
ANY value
WHERE
  value ∈ TSHORT ∧ value ≥ 0 ∧
  value ≤ buffer(OFFSET_LC) ∧
  buffer(OFFSET_LC) +
  BUFFER_HEADER_LENGTH
    ≤ BUFFER_LENGTH
  THEN res := value
END
END (...)

```

```

IMPLEMENTATION
  JCounter_imp
REFINES
  JCounterFF_ref
SEES
  JCounterContext,
  TShort (...)
IMPORTS
  ISOException.Exception(
    EXCEPTIONS),
  apdu.APDU(...)
OPERATIONS
  jc_increment ( vv ) =
  VAR buffer, (...), value_lc, le, res
IN
  buffer ← apdu.getBuffer; (...)
  value_lc ←
  apdu.setIncomingAndReceive;
  ( ... data processing ... )
  le ← apdu.setOutgoing;
  apdu.setOutgoingLength(1);
  buffer(0) := res;
  apdu.sendBytes(0, 1)
END (...)

```

Fig. 9. Part of APDU machine (left) and its use in an implementation (right)

```

MACHINE JInt
SEES TBoolean
CONCRETE_CONSTANTS
  MAXJINT, MINJINT, JINT,
  sum_jint, subt_jint, ... , equal_jint, gt_jint
PROPERTIES
  MINJINT ∈  $\mathcal{Z}$  ∧ MINJINT = - 2147483648 ∧
  MAXJINT ∈  $\mathcal{Z}$  ∧ MAXJINT = 2147483647 ∧
  JINT = MINJINT .. MAXJINT ∧
  sum_jint ∈ JINT × JINT ↔ JINT ∧
  sum_jint = λ ( a1 , a2 ) . ( a1 ∈ JINT ∧ a2 ∈ JINT ∧
    ( a1 + a2 ) ∈ JINT | a1 + a2 ) (...)
END

```

Fig. 10. Part of JInt library machine: type definition and *sum* operation

therefore contribute for the correctness of the generated application as a whole, since not only the core application, but all its necessary support classes are subject to formal development and verification.

7 Conclusions

The starting point of this work was to identify the general structure of Java Card applications, and to develop B specifications of some typical Java Card applications, e.g. ticketing, electronic wallet, etc. These case studies evolved to the BSmart method and the first version of its tool support.

Current B development tools include code generation for imperative languages such as C and ADA. The development of optimized C code for smart cards has been subject of study of the *B with Optimized Memory* (BOM) project [13] [14], proposing optimizations such as method inlining. Optimization is an open issue in our work, but we plan, for instance, to reduce the number of local variables introduced in an operation and to minimize class instantiation. Regarding the translation for Java, a recent initiative is the integration of a translator in the *Rodin* platform [15]. A first Java Card synthesis approach has been proposed in [16]. It was implemented in the *JBtools* platform [10] and provides a code generator for Java optimized for Java Card compatibility. However, there is no specific generation for the Java Card applet and no API support is provided for the host application. The code generation for Java Card is also restricted to the *short* integer type.

Our goal is to provide a complete Java Card service generation method, consisting of the card-side application development, as well as an API for host applications to transparently access the card services. Thus the user does not need to deal with type adaptation/conversion and the Java Card lower-level protocols. The method, jointly with the provided B library of Java Card classes, types and useful data structures, form the basis of an environment to effectively and efficiently develop fully-verified Java Card software.

As future work, we want to verify the result of the translation to Java and Java Card. An approach is the inclusion of JML annotations in the generated code to allow runtime checking, as in the work of [17]. We also plan to apply advanced language transformation techniques, such as TXL [18] or ASF+SDF [19] to generate part of the B refinements and the Java Card application final code to replace our *ad hoc* low level implementation of the transformation rules.

Finally, to better validate the proposal and its tool support, we have to develop a more complex case study. A good candidate is the Mondex electronic purse, a case study that is part of the Verified Software Initiative. In the Mondex system some amount of monetary value is transferred from a source to a target smart card purse in a non-atomic protocol. Each purse must be implemented in isolation, without sharing properties through a global control. The Mondex system has been formally specified in Event-B in Butler and Yadav [20] work and in other work using several formalisms. We started to adapt this system specification to a programming specification, extracting the card and host specifications, modeling them according to the BSmart method.

References

1. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Reading (2000)
2. Gomes, B., Moreira, A.M., Déharbe, D.: Developing Java Card applications with B. In: Brazilian Symposium on Formal Methods (SBMF), pp. 63–77 (2005)
3. Deharbe, D., Gomes, B.G., Moreira, A.M.: Automation of Java Card component development using the B method. In: ICECCS, pp. 259–268. IEEE Comp. Soc., Los Alamitos (2006)
4. Déharbe, D., Gomes, B.G., Moreira, A.M.: Bsmart: A Tool for the Development of Java Card Applications with the B Method. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 351–352. Springer, Heidelberg (2008)
5. Ortiz, E.C.: An Introduction to Java Card Technology, <http://java.sun.com/javacard/reference/techart/javacard1> (2003)
6. PC/SC Workgroup: PC/SC Workgroup Web site (2009), <http://www.pcscworkgroup.com>
7. Global Platform: Global Platform Web site (2009), <http://www.globalplatform.org>
8. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge U. Press, Cambridge (1996)
9. Déharbe, D., Gomes, B.G., Moreira, A.M.: Refining Interfaces: The Case of the B Method. Technical report, Fed. Univ. of Rio Grande do Norte (2009) (to appear)
10. Voisinet, J.C.: JBtools: an experimental platform for the formal B method. In: Principles and Practice of Programming, Maynooth, NUI, pp. 137–139 (2002)
11. Meijer, H., Poll, E.: Towards a Full Formal Specification of the Java Card API. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 165–178. Springer, Heidelberg (2001)
12. Larsson, D.: OCL Specifications for the Java Card API. Master's thesis, School of Computer Science and Engineering, Göteborg University (2003)
13. Requet, A., Bossu, G.: Embedded formally proved code in a smart card: Converting B to C. In: ICFEM 2000, York, UK, p. 15. IEEE Computer Society, Los Alamitos (2000)
14. Bert, D., et al.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003)
15. Edmunds, A., Butler, M.: Code Generation for Event-B with Intermediate Specification. In: Rodin User and Developers Workshop (2009), http://wiki.event-b.org/index.php/Rodin_Workshop_2009
16. Tatibouet, B., Requet, A., Voisinet, J., Hammad, A.: Java Card Code Generation from B Specifications. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 306–318. Springer, Heidelberg (2003)
17. Costa, U., Moreira, A., Musicante, M., Neto, P.: Specification and Runtime Verification of Java Card Programs. In: Brazilian Symp. on Formal Methods (2008)
18. Cordy, J.: The TXL Programming Language (2009), <http://www.meta-environment.org>
19. Meta-Environment.org: The ASF+SDF Meta-Environment (2009), <http://www.txl.ca/index.html>
20. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. Formal Aspects of Computing 20(1), 61–77 (2007)

Automatic Verification for a Class of Proof Obligations with SMT-Solvers

David Déharbe

UFRN / DIMAp / ForAll

Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Formal Methods and Languages Research Laboratory
Natal, RN, Brazil

Abstract. Software development in B and Event-B generates proof obligations that have to be discharged using theorem provers. The cost of such developments therefore depends directly on the degree of automation and efficiency of theorem proving techniques for the logics in which these lemmas are expressed. This paper presents and formalizes an approach to transform a class of proof obligations generated in the Rodin platform in a language that can be addressed by state-of-the-art SMT solvers. The work presented in the paper handles proof obligations with Booleans, integer arithmetics and basic sets.

1 Introduction

Formal software development using frameworks such as B, Event-B and Z produces large quantities of proof obligations (POs), typically expressed in a first-order language including arithmetic and set-theoretic constructs. In the case of B and Event-B, platforms such as Atelier-B [1] and Rodin [2] include theorem provers able to discharge automatically a significant portion of these POs. The remaining POs need the intervention of the users to be addressed. For each such PO, three outcomes are possible. First, when there is an error in the model, the PO is not valid. The user must inspect the PO to understand the cause of the error, correct the model and verify it again. Second, the PO may be valid, but cannot be automatically proved because the proof system lacks some axioms, as the specification logic is incomplete. In that case, it is possible to patch the prover with additional rules so that it finds a proof for the verification condition. Care must be taken not to make the system unsound. Third, the PO may be true, but cannot be proved automatically within the space and time bounds set by the user, due to computational complexity of the verification system. In that case, the user has the possibility to interact with the verification sub-system and help the theorem prover find the proof. Such interactions are a time-consuming activity and have a direct impact on the cost of software development. Progress in automatic theorem proving techniques for formal software development framework is therefore key to increase the application and dissemination of formal methods. This may be achieved with (1) more cost-effective

algorithms, (2) better integration with the development framework (providing counter-examples and dependencies between proofs and specification elements), (3) native support for more expressive logics.

The Satisfiability Modulo Theory (SMT) approach to theorem proving is successful to address many software-related problems. SMT-solvers are tools that implement this approach. They combine a boolean satisfiability engine to handle the propositional structure of the PO, optimized decision procedures for individual theories, such as arrays and arithmetics, a framework to combine these decision procedures [3], and other optimizations, such as theory propagation [4]. Some solvers are also able to build a proof of their result, which is often as useful as the result for the users. The international SMT-LIB initiative provides a common input format [5] language and a repository of benchmarks for SMT-solvers.

Recently, the formal methods community has shown interest in applying SMT-solvers in the verification of POs, which seem to be good candidates to achieve progress in at least two of the above mentioned directions. Indeed, the inclusion in the SMT-LIB, the standard input format for SMT-solvers, of a theory for state-based specification languages has recently been proposed [6]. This proposition considers many specification constructs such as sets, sequences and maps that are not yet fully handled by SMT-solving techniques.

The goal of the work presented in this paper is to present an approach to apply *existing* SMT-solvers to POs, restricted to a subset of the specification constructs of B and Event-B: booleans, linear integer arithmetics and basic sets, plus operators mapping sets to numbers such as cardinality. This experiment is based on a translation of the POs generated in the Rodin platform to the SMT-LIB language and, if successful, lays the basis for the implementation of a verification plug-in based on SMT-solvers for this platform. The scope and languages involved in this translation are presented in Section 2. The general principles of this translation are described in Section 3 and the detailed presentation of its formalization in Section 4. This translation was applied to a set of benchmarks supplied by an industrial partner and the result of their verification with an existing SMT-solver are presented in Section 5. The conclusions of this work are presented in Section 6.

2 The Source and Target Languages

2.1 Proof Obligations in Rodin

The Rodin platform [2] stores POs as XML files. The abstract syntax of the format is described hereafter. A PO file is structured in four sections:

$$L := T^+ \quad E \quad H^* \quad G$$

(theories) (typing environment) (hypothesis) (goal)

The theories are pre-defined names corresponding to different fragments of the specification logic. This work only considers theories that correspond to booleans,

integers and simple sets (i.e. no set of sets is allowed), but it could be straightforwardly extended to include real numbers. Another possible extension would be basic binary relations (i.e. binary relations over basic sets).

Since this work only considers booleans, integers and simple sets, the following grammar defines the typing environment:

$$\begin{array}{ll}
 E := & \text{(typing environment)} \\
 & V^* \text{ variables} \\
 S := & \text{(sort)} \\
 & C \text{ carrier} \\
 & | \mathbb{P}(C) \text{ set} \\
 \\
 V := & \text{(variable)} \\
 & \textit{name} \text{ name} \\
 & S \text{ sort} \\
 C := & \text{(carrier set)} \\
 & \mathbb{Z} \text{ integers} \\
 & | \textit{name} \text{ user-defined}
 \end{array}$$

Sorts **BOOL** and \mathbb{Z} are pre-defined. In B, new sorts (also called basic types) are carrier sets and may be introduced either as deferred sets (only their name is given, and they are assumed to be non-empty) or as enumeration (and the only values in the sorts are those that are enumerated, and they are pairwise distinct).

Finally, the hypothesis and the goal are first-order formulas. The language for formulas is specified by the following grammar, where non-terminals ϕ and τ stand respectively for formulas and terms. POs are well-typed and no type checking or type inference is needed.

$$\begin{array}{l}
 \phi := \neg\phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \phi \wedge \dots \wedge \phi \mid \phi \vee \dots \vee \phi \mid \exists x \bullet \phi \mid \forall x \bullet \phi \mid \\
 \tau := \tau \mid \tau \neq \tau \mid \tau < \tau \mid \tau > \tau \mid \tau \leq \tau \mid \tau \geq \tau \mid \tau \subseteq \tau \mid \tau \subset \tau \mid \tau \in \tau
 \end{array}$$

The considered terms have sort **BOOL** and \mathbb{Z} as well as carrier sets or their powerset. Sets may be ranges of integers, or defined in extension or intentionally.

$$\begin{array}{l}
 \tau := \textit{name} \mid \mathbf{TRUE} \mid \mathbf{FALSE} \mid \textit{num} \mid \mathbf{bool}(\phi) \mid \\
 \quad - \tau \mid \tau - \tau \mid \tau \mathbf{div} \tau \mid \tau \mathbf{mod} \tau \mid \tau + \dots + \tau \mid \tau \times \dots \times \tau \mid \\
 \quad \tau.. \tau \mid \{\tau, \dots, \tau\} \mid \{x \mid \phi\} \mid \tau \cup \dots \cup \tau \mid \tau \cap \dots \cap \tau \mid \tau \setminus \tau
 \end{array}$$

Finally there are mixed operators that are neither basic set operators nor integer arithmetic operators; they are:

$$\begin{array}{l}
 \phi := \mathbf{finite}(\tau) \\
 \tau := \mathbf{min}(\tau) \mid \mathbf{max}(\tau) \mid \mathbf{card}(\tau)
 \end{array}$$

2.2 The SMT-LIB Format

The SMT-LIB format [5] is the result of an international effort to establish a common input language for SMT-solvers. In addition, this initiative also collects and classifies proof obligations expressed in this format, thus establishing a benchmark library for SMT-solvers. Both the format and the benchmarks are

used as a reference to establish quantitative measures of SMT-solvers and their evolution. The SMT-LIB format is thus the *de facto* standard input language for SMT-solvers. Although there is an on-going discussion to extend the format, this paper considers version 1.2, which is the most recent official version.

A SMT benchmark is a sequence of declarations of different entities: logic, sorts, functions, predicates, assumptions, and goal. The logic is a pre-established name, to which are associated sort, function and predicate declarations, possibly some syntactical restrictions, as well as a semantics. For instance, the logic QF_IDL corresponds to quantifier-free formulas with integer arithmetic terms where the constraints bound numerically subtractions between integer-sorted terms. Related to our work is the recent proposal [6] to include a logic for POs from the Vienna Development Method [7] in the SMT-LIB. This theory includes finite sets, lists and maps; it is currently not supported by existing SMT-solvers.

In general, a SMT benchmark also contains declarations for additional sorts, functions (including constants) and predicates (including atomic propositions). A sort is defined by its name. A function declaration is composed of the function symbol, and the list with the sorts of the parameters followed by the sort of the result. A predicate declaration is composed of the predicate symbol and the list of the sorts of its arguments. The semantics of such symbols may be defined axiomatically with first-order formulas as assumptions.

Finally, the assumptions and the goal are first-order multi-sorted logic formulas expressed using both the symbols of the declared logic and the additional symbols declared in the benchmark. In the general case, formulas may contain arbitrary combinations of quantifiers; the language also contains if-then-else constructs for both terms and formulas.

3 Rationale of the Translation

On the one-hand, SMT-solvers integrate and combine reasoning engines for a number of logics, such as arrays, bit-vectors, abstract data types, and different fragments of integer and real-number arithmetics. On the other hand, Rodin POs may contain arbitrary integer arithmetic expressions, arbitrary combinations of set constructs, including derived entities such as relations, functions and sequences, as well as operators mapping sets to integers and vice-versa.

The logic of Event-B and B contains terms of sort Boolean which is reflected in Rodin POs. The translation maps such terms to formulas, as this approach should take better advantage of the efficient handling of complex Boolean structures in SMT-solvers.

Considering arithmetics, the SMT-LIB benchmarks are divided into several divisions according to the class of constraints: difference logic, linear arithmetics, non-linear arithmetics. The benchmarks are further divided whether the formulas are quantified or quantifier-free and whether the numeric sort is integers or real numbers. This information is available to SMT-solvers in a benchmark attribute; they may use it to select the most efficient procedure for the given benchmark. The translation systematically sets this attribute to quantified linear integer arithmetic as it reflects the addressed class of POs.

Currently, no SMT solver provides direct support for set theory. A possible approach is to map set constructs to symbols of a theory that can be handled in one of the existing SMT-solvers. It is possible to map sets to arrays of booleans indexed by the elements of the domain of the set (see e.g. in [8,6]). Another solution is to represent sets by their characteristic predicate, i.e. a boolean-valued function f taking as argument a value of the corresponding carrier set. This is the approach used to discharge set-theoretic arguments in a proof assistant embedding a SMT-solver [9] as well as in the *Predicate Prover*, available in Rodin and Atelier-B. The former solution has the advantage of being more general, as it allows nesting sets. The latter solution can only deal with basic sets (no set of sets) but provides a direct mapping to first-order logic with uninterpreted functions and predicates, for which efficient reasoning engines are available: this paper investigates this approach.

Sets and set operators are thus translated to predicates and boolean connectors as usual: false for the empty set, disjunction for union, implication for set inclusion, predicate application for set membership, etc. To implement this approach, we employ extensions to the SMT-LIB library implemented in the veriT SMT-solver [10], namely macro definitions and lambda expressions. veriT processes such constructs, applying macro expansion and beta-reduction, producing formulas conforming to the SMT-LIB format that may be directly processed in veriT or by any SMT-solver equipped with the right decision procedures.

For instance, `union` is a macro used in the translation of set union (Section 4):

$$\text{union} \equiv (\text{lambda } (p \text{ ('s Bool)})(q \text{ ('s Bool)})(\text{lambda } (?x \text{ 's}) (\text{or } (p \text{ ?x}) (q \text{ ?x}))))$$

Here, `union` is declared as a macro with two arguments `p` and `q`. Both arguments are sorted as `('s Bool)`, i.e. as a unary predicate, where `'s` is a sort variable. The macro `union` expands to the lambda expression given in the right hand side of the expression: it denotes a unary predicate with formal parameter `?x` of sort `'s`, defined as disjunction of the application of `p` and `q` to `?x`. veriT implements a type inference mechanism to handle macros with sort variables similar to that for “let polymorphism” [11]. The remaining macros are:

<i>empty set:</i>	<code>empty</code> \equiv <code>(lambda (?x 's) false)</code>
<i>intersection:</i>	<code>inter</code> \equiv <code>(lambda (p ('s Bool))(q ('s Bool)) (lambda (?x 's) (and (p ?x) (q ?x))))</code>
<i>difference:</i>	<code>setminus</code> \equiv <code>(lambda (p ('s Bool))(q ('s Bool)) (lambda (?x 's) (and (p ?x) (not (q ?x)))))</code>
<i>membership:</i>	<code>in</code> \equiv <code>(lambda (x 's) (p ('s Bool))(p x))</code>
<i>inclusion:</i>	<code>subseteq</code> \equiv <code>(lambda (p ('s Bool))(q ('s Bool)) (lambda (?x 's) (implies (p ?x) (q ?x))))</code>
<i>strict inclusion:</i>	<code>subset</code> \equiv <code>(lambda (p ('s Bool))(q ('s Bool)) (and (subseteq p q) (not (= p q))))</code>
<i>range:</i>	<code>range</code> \equiv <code>(lambda (lo Int)(hi Int) (lambda (?x Int) (and (<= lo ?x)(<= ?x hi))))</code>

A last observation is in order. The result of the application of macro expansion may result in formulas where equality is applied at the predicate level. For

instance, assume that S is a set over some sort s , the formula $S \cup \emptyset = S$, valid in set theory, expands to:

$$(\text{lambda } (?x s) (\text{or } (\text{ps } ?x) \text{false})) = (\text{lambda } (?x s) (\text{ps } ?x)),$$

where ps is the unary predicate symbol characterizing set S . The original equality between sets results in an equality between formulas. In first-order logic, this equality is expressed with universal quantification and equivalence. This can be performed by rewriting, and corresponds to the application of the set extensionality axiom:

$$(\text{=} \text{p } \text{q}) \rightsquigarrow (\text{forall } (?x t) (\text{iff } (\text{p } ?x) (\text{q } ?x)) \quad \text{if } \vdash_t \text{p} : (t \text{ Bool}) \text{ and } \vdash_t \text{q} : (t \text{ Bool})),$$

where $\vdash_t \text{e} : \text{s}$ is the typing judgement that s is the sort of expression e . Applying this rule followed by beta reduction to the example yields the first-order logic tautology: $(\text{forall } (?x s) (\text{iff } (\text{or } (\text{p } ?x) \text{false})(\text{p } ?x)))$.

4 Formalizing the Translation

This section presents the formalization of the translation of Rodin lemmas to SMT-LIB format extended with macros and lambda expressions. The translation is as a tree traversal, recursing over the syntactic structure of the lemma. This traversal is specified as a set of rules that follow the style of structural operational semantics.

4.1 Preliminary Definitions and Notations

The rules propagate an evaluation context Γ that gathers incrementally the contents of the different sections that compose the SMT-LIB format. They are: **sorts**, a set of SMT-LIB identifiers; **preds** (predicate symbols) and **funs** (function symbols), both maps SMT-LIB identifiers to a list of sorts; **assumptions**, a set of formulas in SMT-LIB syntax; **formula**, the goal, a formula in SMT-LIB syntax. In addition, the context maintains a mapping nm from Rodin symbols to SMT-LIB identifiers. Γ_I denotes the initial context and is such that:

$$\begin{aligned} \Gamma_I = \{ \text{nm} &= \{ \mathbb{Z} \mapsto \text{Int}, \text{BOOL} \mapsto \text{Bool}, \\ &\quad \text{TRUE} \mapsto \text{true}, \text{FALSE} \mapsto \text{false} \\ &\quad \wedge \mapsto \text{and}, \vee \mapsto \text{or}, \Rightarrow \mapsto \text{implies}, \Leftrightarrow \mapsto \text{iff}, \\ &\quad \exists \mapsto \text{exists}, \forall \mapsto \text{forall}, \\ &\quad = \mapsto =, < \mapsto <, \leq \mapsto <=, > \mapsto >, \geq \mapsto >=, \\ &\quad \subset \mapsto \text{subset}, \subseteq \mapsto \text{subsetq}, \in \mapsto \text{in} \\ &\quad + \mapsto +, \times \mapsto *, \text{div} \mapsto /, \text{mod} \mapsto \%, - \mapsto - \\ &\quad \cup \mapsto \text{union}, \cap \mapsto \text{inter}, \\ &\quad \setminus \mapsto \text{setminus}, \emptyset \mapsto \text{empty} \} \\ \text{sorts} &= \{ \text{Int}, \text{Bool} \}, \\ \text{funs} &= \{ \}, \text{preds} = \{ \}, \text{assumptions} = \{ \}, \text{formula} = \{ \} \} \end{aligned}$$

In the following, $\Gamma.\text{sec}$ denotes the content of Section sec of Γ ; when sec is a map, $\Gamma[\text{sec} \oplus s]$ denotes update of sec with the map s , otherwise it denotes inclusion of s to the set sec ; $\Gamma[\text{sec} \ominus s]$ sec denotes removal of s from sec . We assume the existence of a function *btype* that, given an expression e in a Rodin expression, returns the basic type e . Also *fresh* denotes a predicate that tests if each element of a given set of SMT-LIB identifiers is fresh with respect to the current context.

4.2 Translation Rule for a PO

Rule [1](#) specifies how the evaluation of a PO L composed of the sections TEH^*G results in a context Γ . The operators $\llbracket L \rrbracket_L$, $\llbracket E \rrbracket_E$, $\llbracket H \rrbracket_H$, $\llbracket G \rrbracket_G$ are responsible for translating a full PO, the typing environment, the hypothesis and the goal respectively. The resulting context Γ represents the components of SMT-LIB format for the Rodin PO L .

$$1 \frac{L = TEH^*G \quad \llbracket E; \Gamma_I \rrbracket_E = \Gamma_0 \quad \llbracket H^*; \Gamma_0 \rrbracket_H = \Gamma_1 \quad \llbracket G; \Gamma_1 \rrbracket_G = \Gamma}{\llbracket L \rrbracket_L = \Gamma}$$

4.3 Translation Rules for the Typing Environment

Rules [3](#) and [2](#) specify that the typing environment lemma is translated by a sequential traversal, applying the translation operator $\llbracket \cdot \rrbracket_V$ to each variable declaration:

$$2 \frac{\llbracket V^*; \Gamma \rrbracket_E = \Gamma_0 \quad \llbracket V; \Gamma_0 \rrbracket_V = \Gamma_1}{\llbracket V^*V; \Gamma \rrbracket_E = \Gamma_1} \quad 3 \frac{}{\llbracket ; \Gamma \rrbracket_E = \Gamma}$$

The declarations of a Rodin lemma, grouped in the typing environment, are pairs n, t , where n is the name of the declared entity, and t is its basic type. When n and t are identical, then a new basic type is introduced in B, which is mapped to a new sort in SMT-LIB (Rule [4](#)). When n and t differ, t may be a basic type, and then n is a value of type t , and a corresponding constant function n is added in the SMT-LIB (Rule [5](#)). Another possibility is that the type be the powerset of some basic type t , then n is a set, and a corresponding unary predicate n is added in the SMT-LIB (Rule [6](#)).

$$4 \frac{n = t \quad \Gamma' = \Gamma[\text{nm} \oplus \{t \mapsto \mathbf{t}\} \mid \text{sorts} \oplus \{\mathbf{t}\}]}{\llbracket n t; \Gamma \rrbracket_V = \Gamma'} \text{ (basic type)}$$

$$5 \frac{n \neq t \quad \Gamma.\text{nm}(t) = \mathbf{t} \quad \mathbf{t} \in \Gamma.\text{sort} \quad \Gamma' = \Gamma[\text{nm} \oplus \{n \mapsto \mathbf{n}\} \mid \text{funs} \oplus \{n \mapsto (\mathbf{t})\}]}{\llbracket n t; \Gamma \rrbracket_V = \Gamma'} \text{ (set element)}$$

$$6 \frac{n \neq t \quad \Gamma(t) = \mathbf{t} \quad \mathbf{t} \in \Gamma.\text{sort} \quad \Gamma' = \Gamma[\text{nm} \oplus \{n \mapsto \mathbf{n}\} \mid \text{preds} \oplus \{n \mapsto (\mathbf{t})\}]}{\llbracket n \mathbb{P}(t); \Gamma \rrbracket_V = \Gamma'} \text{ (set)}$$

4.4 Translation Rules for Hypothesis and Goal

Rules [7](#) and [8](#) specify that the hypothesis section is translated by a sequential traversal, applying the translation operator $\llbracket \cdot \rrbracket_\phi$ to each hypothesis, and yielding a formula f and a context Γ . The former is added to the latter as a result of the translation.

$$7 \frac{}{\llbracket ; \Gamma \rrbracket_H = \Gamma} \quad 8 \frac{\llbracket \phi^* ; \Gamma \rrbracket_H = \Gamma_0 \quad \llbracket \phi ; \Gamma_0 \rrbracket_\phi = f ; \Gamma_1}{\llbracket \phi^* \phi ; \Gamma \rrbracket_H = \Gamma_1[\text{assumptions} \oplus \{f\}]}$$

The goal section of the lemma is also translated using operator $\llbracket \cdot \rrbracket_\phi$, and the resulting formula is set as the goal formula in the context:

$$9 \frac{\llbracket G ; \Gamma \rrbracket_\phi = g ; \Gamma_1}{\llbracket G ; \Gamma \rrbracket_G = \Gamma_1[\text{formula} \oplus \{g\}]}$$

4.5 Translation Rules for Formulas

The translation operator $\llbracket \cdot \rrbracket_\phi$ recurses over the structure of the formulas, up to the level of atoms. There are two classes of atoms: boolean constants, and applications of relational operators. The definition of the latter uses the term translation operator $\llbracket \cdot \rrbracket_\tau$ (defined in Section [4.6](#)) to process the arguments. Their translation is specified by rules [10](#) and [11](#):

$$10 \frac{o \in \{=, <, >, \leq, \geq, \subset, \subseteq, \in\} \quad \llbracket \tau_1 ; \Gamma \rrbracket_\tau = \mathbf{t}_1 ; \Gamma_1 \quad \llbracket \tau_2 ; \Gamma_1 \rrbracket_\tau = \mathbf{t}_2 ; \Gamma_2 \quad \mathbf{o} = \Gamma.\text{nm}(o)}{\llbracket \tau_1 \circ \tau_2 ; \Gamma \rrbracket_\phi = (\mathbf{o} \mathbf{t}_1 \mathbf{t}_2) ; \Gamma_2}$$

$$11 \frac{\Gamma.\text{nm}(\text{name}) = \text{name}}{\llbracket \text{name} ; \Gamma \rrbracket_\phi = \text{name} ; \Gamma}$$

The rules for non-atomic formulas ([12](#)–[14](#)) are straightforward recursive applications to their arguments, propagating the context accordingly, and combining the results in the SMT-LIB syntax.

$$12 \frac{\llbracket \phi ; \Gamma \rrbracket_\phi = f ; \Gamma'}{\llbracket \neg \phi ; \Gamma \rrbracket_\phi = (\text{not } f) ; \Gamma'}$$

$$13 \frac{o \in \{\wedge, \vee\} \quad \mathbf{o} = \Gamma.\text{nm}(o) \quad \llbracket \phi_1 ; \Gamma \rrbracket_\phi = \mathbf{f}_1 ; \Gamma_1 \cdots \llbracket \phi_n ; \Gamma_{n-1} \rrbracket_\phi = \mathbf{f}_n ; \Gamma_n}{\llbracket \phi_1 \circ \cdots \circ \phi_n ; \Gamma \rrbracket_\phi = (\mathbf{o} \mathbf{f}_1 \mathbf{f}_2 \cdots \mathbf{f}_n) ; \Gamma_n}$$

$$14 \frac{o \in \{\Rightarrow, \Leftrightarrow\} \quad \mathbf{o} = \Gamma.\text{nm}(o) \quad \llbracket \phi_1 ; \Gamma \rrbracket_\phi = \mathbf{f}_1 ; \Gamma_1 \quad \llbracket \phi_2 ; \Gamma_1 \rrbracket_\phi = \mathbf{f}_2 ; \Gamma_2}{\llbracket \phi_1 \circ \phi_2 ; \Gamma \rrbracket_\phi = (\mathbf{o} \mathbf{f}_1 \mathbf{f}_2) ; \Gamma_2}$$

Rule [15](#) handles quantified formulas. As SMT-LIB requires that quantified variables be sorted, the basic type of the quantified variable x is identified and the corresponding sort s is obtained from the context. A temporary association is associated to the context that is used to translate the matrix of the quantified formula.

$$15 \frac{\begin{array}{c} Q \in \{\exists, \forall\} \\ \Gamma.\text{nm}(\text{btype}(x)) = s \end{array} \quad \begin{array}{c} Q = \Gamma.\text{nm}(Q) \\ \llbracket \phi; \Gamma[\text{nm} \oplus \{x \mapsto ?x\}] \rrbracket_\phi = f; \Gamma_1 \end{array}}{\llbracket Qx \bullet \phi; \Gamma \rrbracket_\phi = (Q(?x s) f); \Gamma_2 = \Gamma_1[\text{nm} \ominus x]}$$

4.6 Translation Rules for Terms

The operator $\llbracket \cdot \rrbracket_\tau$ recurses over the structure of terms found in Rodin lemmas, and builds terms according to the SMT-LIB syntax. The base cases of the recursion are identifiers (Rule [16](#)) and numeric literals (Rule [17](#)). Boolean conversion is also dealt with (Rule [18](#)).

$$16 \frac{\text{id} = \Gamma.\text{nm}(\text{id})}{\llbracket \text{id}; \Gamma \rrbracket_\tau = \text{id}; \Gamma} \quad 17 \frac{}{\llbracket \text{num}; \Gamma \rrbracket_\tau = \text{num}; \Gamma} \quad 18 \frac{\llbracket \phi; \Gamma \rrbracket_\phi = f; \Gamma_1}{\llbracket \text{bool}(\phi); \Gamma \rrbracket_\tau = f; \Gamma_1}$$

Rules [19](#)–[21](#) specify the translation of arithmetic terms:

$$19 \frac{o \in \{+, \times\} \quad o = \Gamma.\text{nm}(o) \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1; \Gamma_1 \cdots \llbracket \tau_n, \Gamma_{n-1} \rrbracket_\tau = \mathbf{t}_n; \Gamma_n}{\llbracket \tau_1 o \dots o \tau_n; \Gamma \rrbracket_\tau = (o \mathbf{t}_1 \cdots \mathbf{t}_n); \Gamma_n}$$

$$20 \frac{o \in \{\text{div}, \text{mod}, -\} \quad o = \Gamma.\text{nm}(o) \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1; \Gamma_1 \quad \llbracket \tau_1, \Gamma_2 \rrbracket_\tau = \mathbf{t}_2; \Gamma_2}{\llbracket \tau_1 o \tau_2; \Gamma \rrbracket_\tau = (o \mathbf{t}_1 \mathbf{t}_2); \Gamma_2}$$

$$21 \frac{\llbracket \tau, \Gamma \rrbracket_\tau = \mathbf{t}; \Gamma_1}{\llbracket -\tau; \Gamma \rrbracket_\tau = (\sim \mathbf{t}); \Gamma_1}$$

The following rules specify the translation of set terms. The first group of rules deal with the base cases, i.e. the empty set (Rule [22](#)), and sets defined in intention (Rule [23](#)) and in extension (Rule [24](#)).

$$22 \frac{}{\llbracket \emptyset, \Gamma \rrbracket = \text{empty}, \Gamma} \quad 23 \frac{\Gamma.\text{nm}(\text{btype}(x)) = s \quad \llbracket \phi, \Gamma \rrbracket_\tau = f; \Gamma_1}{\llbracket \{x \mid \phi\}; \Gamma \rrbracket_\tau = (\text{lambda } (?x s) f); \Gamma_1}$$

$$24 \frac{\Gamma.\text{nm}(\text{btype}(\tau_1)) = s \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1; \Gamma_1 \quad \cdots \quad \llbracket \tau_n, \Gamma_{n-1} \rrbracket_\tau = \mathbf{t}_n; \Gamma_n}{\llbracket \{\tau_1, \dots, \tau_n\}; \Gamma \rrbracket_\tau = (\text{lambda } (?x s)(\text{or } (= ?x \mathbf{t}_1) \cdots (= ?x \mathbf{t}_n))); \Gamma_n}$$

The remaining classes of set expressions are translated according to Rules [25](#)–[27](#). Note that the translation of intersection and union needs to transform the variadic set connectives of Rodin to binary macros.

$$25 \frac{o \in \{\cup, \cap\} \quad o = \Gamma.\text{nm}(o) \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1, \Gamma_1 \quad \llbracket \tau_2, \Gamma_1 \rrbracket_\tau = \mathbf{t}_2, \Gamma_2}{\llbracket \tau_1 o \tau_2, \Gamma \rrbracket = (o \mathbf{t}_1 \mathbf{t}_2), \Gamma_2}$$

$$26 \frac{o \in \{\cup, \cap\} \quad o = \Gamma.\text{nm}(o) \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1, \Gamma_1 \quad \llbracket \tau_2 o \cdots o \tau_n, \Gamma_1 \rrbracket_\tau = \mathbf{t}, \Gamma_2}{\llbracket \tau_1 o \tau_2 o \cdots o \tau_n, \Gamma \rrbracket = (o \mathbf{t}_1 \mathbf{t}), \Gamma_2}$$

$$27 \frac{o \in \{\setminus, \dots\} \quad o = \Gamma.\text{nm}(o) \quad \llbracket \tau_1, \Gamma \rrbracket_\tau = \mathbf{t}_1, \Gamma_1 \quad \llbracket \tau_2, \Gamma_1 \rrbracket_\tau = \mathbf{t}_2, \Gamma_2}{\llbracket \tau_1 o \tau_2, \Gamma \rrbracket = (o \mathbf{t}_1 \mathbf{t}_2), \Gamma_2}$$

4.7 Mixed Operators

Finally, translation of mixed-sort operators require additional definitions. The following operators are considered for translation: **min** and **max** that yield respectively the smallest and highest value of a non-empty set of integers, the set predicate operator **finite** and the cardinality operator **card**.

In order to define the translation of the first two operators, the following two macros are introduced:

$$\text{ismin} \equiv (\text{lambda } (m \text{ Int}) (t \text{ (Int Bool)}) \\ (\text{and}(\text{in } m \text{ t})(\text{forall } (?x \text{ Int}) (\text{implies } (\text{in } ?x \text{ t})(\leq m ?x))))))$$

$$\text{ismax} \equiv (\text{lambda } (m \text{ Int}) (t \text{ (Int Bool)}) \\ (\text{and}(\text{in } m \text{ t})(\text{forall } (?x \text{ Int}) (\text{implies } (\text{in } ?x \text{ t})(\leq ?x m))))))$$

Thus, $(\text{ismin } m \text{ t})$ expands to a formula stating that m is equal to the smallest value in the set t . The translation of the operators **min** and **max** is specified by Rules 28 and 29. Each application of these rules add a fresh integer constant and an assumption to the context.

$$28 \frac{\frac{\llbracket \tau, \Gamma \rrbracket_{\tau} = \mathbf{t}; \Gamma_1 \quad \text{fresh}(\{m\})}{\Gamma_2 = \Gamma_1[\text{funs} \oplus \{m \mapsto \text{Int}\} \mid \text{assumptions} \oplus \{(\text{ismin } m \text{ t})\}]}}{\frac{\llbracket \mathbf{min}(\tau), \Gamma \rrbracket_{\tau} = m; \Gamma_2 \quad \text{fresh}(\{m\})}{\llbracket \tau, \Gamma \rrbracket_{\tau} = \mathbf{t}; \Gamma_1} \quad \text{fresh}(\{m\})}{\Gamma_2 = \Gamma_1[\text{funs} \oplus \{m \mapsto \text{Int}\} \mid \text{assumptions} \oplus \{(\text{ismax } m \text{ t})\}]}} \\ 29 \frac{\llbracket \mathbf{max}(\tau), \Gamma \rrbracket_{\tau} = m; \Gamma_2$$

The operator **finite** is a predicate that is true of finite sets. The following macro relates a proposition p , a unary predicate t , a labelling function f and a constant k . Informally, p is an atomic proposition stating that the argument set is finite, k is an upper bound on the cardinality of the set, and f maps injectively elements of the set with a positive integer smaller than k .

$$\text{finite} \equiv (\text{lambda } (p \text{ Bool}) (t \text{ ('s Bool)}) (f \text{ ('s Int)}) (k \text{ Int}) \\ (\text{iff } p(\text{and}(\text{forall } (?x \text{ s})(\text{implies } (\text{in } ?x \text{ t})(\text{in } (f ?x)(\text{range } 1 \text{ k})))) \\ (\text{forall } (?x \text{ s})(?y \text{ s})(\text{implies } (\text{and } (\text{in } ?x \text{ t}) \\ (\text{in } ?y \text{ t}) \\ (\text{not } (\text{equal } ?x ?y))) \\ (\text{not } (\text{equal } (f ?x)(f ?y))))))))))$$

Rule 30 specifies the translation of the predicate application $\mathbf{finite}(\tau)$: the context is enriched with an atomic proposition p , a constant k and a function f , of domain s , the sort for the basic type of the elements of τ . The context is also augmented with an assumption obtained by an expansion of macro **finite**.

$$30 \frac{\frac{\llbracket \tau, \Gamma \rrbracket_{\tau} = \mathbf{t}; \Gamma_1 \quad \text{btype}(\tau) = \mathbb{P}(s) \quad \Gamma.nm(s) = s \quad \text{fresh}(\{p, k, f\})}{\Gamma_2 = \Gamma_1[\text{preds} \oplus \{p \mapsto ()\} \mid \text{funs} \oplus \{k \mapsto \text{Int}, f \mapsto (s \text{ Int})\} \mid \\ \text{assumptions} \oplus (\text{finite } p \text{ t } f \text{ k})]}{\llbracket \mathbf{finite}(\tau), \Gamma \rrbracket_{\phi} = p; \Gamma_2}$$

The operator **card** is a function from sets to integers. The following macro relates a unary predicate t , a labelling function f and a constant k . Informally, t is the characteristic function of a set, k is the cardinality of the set, and f maps bijectively elements of the set with the range $[1; k]$.

$$\text{card} \equiv (\text{lambda } (t \text{ ('s Bool)}) (f \text{ ('s Int)}) (k \text{ Int})) \\ (\text{forall } (?x \text{ s})(\text{implies } (\text{in } ?x t)(\text{in } (f ?x)(\text{range } 1 k)))) \\ (\text{forall } (?x \text{ s})(?y \text{ s})(\text{implies } (\text{and } (\text{in } ?x t) (\text{in } ?y t)) \\ (\text{iff } (\text{equal } ?x ?y) (\text{equal } (f ?x)(f ?y))))))$$

Rule [B1](#) specifies the translation of the application of operator **card** to a set τ ; the context is enriched with a constant k and a function f , and an assumption that relates both new symbols to set τ using the macro **card**.

$$\frac{\llbracket \tau, \Gamma \rrbracket_{\tau} = t; \Gamma_1 \quad \text{btype}(\tau) = \mathbb{P}(s) \quad \Gamma.\text{nm}(s) = s \quad \text{fresh}(\{p, k, f\}) \\ \Gamma_2 = \Gamma_1[\text{funs} \oplus \{k \mapsto \text{Int}, f \mapsto (s \text{ Int})\} \mid \\ \text{assumptions} \oplus (\text{card } t \text{ f } k)]}{31 \text{-----} \\ \llbracket \text{card}(\tau), \Gamma \rrbracket_{\phi} = k; \Gamma_2}$$

5 Experimental Results

An initial set of benchmarks was made available by an industrial partner. From this initial set were extracted the proof obligations that use only the constructs listed in Section [2.1](#): booleans, integers, and basic sets. Since the prover is a satisfiability modulo theory (SMT) solver, to check the validity of a lemma, the negation of the lemma is given to the solver. If this negation is found unsatisfiable, then the original formula is valid.

The selected benchmarks were translated manually following the rules presented in Section [4](#). The resulting files were then processed with the SMT-solver **veriT** [\[10\]](#), that integrates the necessary pre-processing steps to handle the constructs introduced in these translation rules.

In the general case, an execution of **veriT** may yield four results. First, the execution may not halt within the resource bounds allocated (be it space or time). The second possible outcome is “unsat”, i.e. unsatisfiability is detected; for some logics, **veriT** is then able to produce a trace of the reasoning steps that were applied to detect this unsatisfiability: this trace can be checked by a third-party tool and used as a certificate of the result. The last two possible results happen when **veriT** is not able to show the input formula is unsatisfiable. The input formula is then inspected and, if it belongs to a logic for which the solver is recorded to be complete, the result is “sat”; otherwise it is “unknown”. In the case of the formulas addressed in this experiment, **veriT** is not complete and may only return the verdicts “unsat”, “unknown” or timeout.

Table [1](#) summarizes the results. The first column identifies each lemma. The second column indicates what is the expected result. The third and fourth columns contain the result obtained with two versions of the SMT-solver **veriT**. The version labeled **veriT 1** in the table, is the version that participated at SMT-COMP’2009, the yearly contest for SMT-solvers. For four lemmas, it is unable

Table 1. Experimental results: verifying RODIN lemmas with veriT

Name of RODIN lemma	logic	veriT 1	veriT 2	time
BepiColombo-thm15.smt	unsat	unsat	unsat	0.049s
BepiColombo-thm2.smt	unsat	unsat	unsat	0.010s
BoschSwitch-4.smt	unsat	unsat	unsat	0.030s
SSF_pilot-3.smt	unsat	unknown	unsat	0.031s
SSF_pilot-5.smt	unsat	unsat	unsat	0.011s
SimpleLyra-7.smt	sat	unknown	unknown	0.012s
ch4_other_file-1.smt	unsat	unsat	unsat	0.011s
ch7_conc-13.smt	unsat	unsat	unsat	0.011s
ch7_conc-21.smt	unsat	unsat	unsat	0.012s
ch7_conc-26.smt	unsat	unknown	unsat	0.033s
ch7_conc-28.smt	unsat	unknown	unsat	0.032s
ch910_ring-2.smt	unsat	unsat	unsat	0.010s
ch912_mobile-1.smt	unsat	unsat	unsat	0.031s
ch915_bin-2.smt	unsat	unknown	unsat	0.032s
ch915_maxi-7.smt	unsat	unsat	unsat	0.016s
ch915_mini-5.smt	unsat	unsat	unsat	0.015s
ch915_sort_other-3.smt	unsat	unsat	unsat	0.013s
gen_hotel_new-14.smt	unsat	unsat	unsat	0.035s
ssf-1.smt	unsat	unsat	unsat	0.016s
ssf-3.smt	unsat	unsat	unsat	0.016s
ssf-4.smt	unsat	unsat	unsat	0.014s
ssf-7.smt	unsat	unsat	unsat	0.013s

to prove that it is unsatisfiable. The reason was that all the corresponding formulas contained quantifiers, and the original instantiation heuristics of veriT failed to find the good instances. This motivated the extension of the quantifier instantiation module of veriT to improve these results.

In the initial experiments, the quantifier instantiation in veriT was handled in two modules: an external theorem prover (namely, the E prover) to reason on purely equational first-order logic using the superposition calculus, and a component implementing custom quantifier instantiation heuristics based on equalities and atom polarities in the quantified formulas. The latter module was then extended to implement a new heuristic that instantiates quantified variables using information from congruence closure, an internal module responsible for reasoning about equalities. Congruence closure maintains equivalence classes between the terms that are present in the solver; each such equivalence class has a representative term. In this heuristic, a quantified variable is instantiated with all the representative terms that have the same sort as the the variable. After including the new heuristics, no performance regression was observed compared to the results that were obtained with the initial version.

This new version, labeled veriT 2 in the table, includes the heuristics that uses the representative terms computed by congruence closure. With this new version, all unsatisfiable lemmas were successfully discharged. As expected, veriT reports an “unknown” verdict on the only lemma that cannot be proved.

Note that the verification time is negligible: the total time to prove all lemmas is less than one second. Even though the benchmarks are indeed small formulas with few hypothesis, these results are promising. Even in the case of the proof obligation that cannot be proved unsatisfiable, the result is returned very quickly.

6 Conclusions

This paper addresses the verification of proof obligations generated in formal systems and software developments using SMT-solvers. A fragment of the XML-based format for proof obligations in the RODIN platform was chosen: essentially it combines basic sets, fragments of integer arithmetics and booleans. The scope therefore compares to that of existing tools such as the Predicate Prover.

The use of pre-processing constructs simplifies the specification and implementation of the translation from set theory to predicate logic: e.g. macros are associated to the main operators of set theory. A set of rules specifies the translation of proof obligations to the SMT-LIB format extended with macro-definitions and lambda expressions. A second stage of translation, applying classic pre-processing techniques such as macro-expansion, produces fully compliant SMT-LIB files that can be handled with existing SMT-solvers.

A set of benchmark proof obligations, supplied by an industrial partner, was used to assess the usefulness of the approach. The translation system was applied to the proof obligations and the resulting SMT-LIB files were verified with an existing SMT-solver. The initial version of the solver was already fast but, in some cases, was unable to produce the expected result. After extending the solver with a new (and simple) quantifier instantiation heuristics, all valid proof obligations were proved almost instantaneously. This experiment validate the approach: existing SMT-solvers may be employed to discharge automatically and quickly a number of proof obligations in formal software developments such as Event-B.

Future Work. The translation presented in this paper will thus be used as the specification for the implementation of a verification plug-in for the Rodin platform targeting SMT-solvers. The translation will then be extended to handle a larger number of specification constructs, namely basic binary relations and arithmetics for real numbers.

Further, the capabilities of SMT-solvers can be used to improve the workflow in the development platform. First, one can define classes of proof obligations where the solver is complete and where more accurate results can be reported, using theoretical results such as those reported in [12]. In those cases where the solver is complete, counter-models can be reported to the user, when a proof obligation cannot be verified. Also, SMT-solvers may identify the subset of hypothesis that was actually useful to verify a proof obligation. This information can be used by the environment platform to reduce the number of generated proof obligations when the user modifies a definition. Finally, other translations can be defined to consider other classes of proof obligations. For instance, as

suggested in [8,6], sets can be mapped to arrays of booleans, for which efficient SMT-solving techniques are available.

Acknowledgements. The author thanks Laurent Voisin for providing examples and Carine Pascal for sharing information on plug-in development for Rodin. The initial implementation of macro expansion, beta reduction and lifting of equalities to formulas in veriT is due to Pascal Fontaine. He also proposed to use them as a means to handle basic set connectors.

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)¹, funded by CNPq grant 573964/2008-4.

References

1. ClearSy: Atelier B User Manual Version 4.0. Clearsy System Engineering (2009), <http://www.atelierb.eu>
2. Coleman, J., Jones, C., Oliver, I., Romanovsky, A.: E.Troubitsyna: RODIN (rigorous open development environment for complex systems). In: Fifth European Dependable Computing Conference: EDCC-5 supplementary volume, pp. 23–26 (2005)
3. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
4. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
5. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2 (August 2006)
6. Kröning, D., Rümmer, P., Weissenbacher, G.: A proposal for a theory of finite sets, lists, and maps for the SMT-LIB standard. In: Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22 (2009)
7. Bruun, H., Damm, F., Dawes, J., Hansen, B., Larsen, P., Parkin, G., Plat, N., Toetenel, H.: A formal definition of VDM-SL. Technical Report Technical Report 1998/9, University of Leicester (1998)
8. Couchot, J.F., Déharbe, D., Giorgetti, A., Ranise, S.: Scalable automated proving and debugging of set-based specifications. Journal of the Brazilian Computer Society 9(2), 17–36 (2003)
9. Hurlin, C., Chaieb, A., Fontaine, P., Merz, S., Weber, T.: Practical proof reconstruction for first-order logic and set-theoretical constructions. In: Dixon, L., Johansson, M. (eds.) The Isabelle Workshop 2007, Bremen, July 16 (2007)
10. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient smt-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
11. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
12. Fontaine, P.: Combinations of theories for decidable fragments of first-order logic. In: Ghilardi, S. (ed.) FroCoS 2009. LNCS, vol. 5749, pp. 263–278. Springer, Heidelberg (2009)

¹ www.ines.org.br

A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking

Edd Turner¹, Michael Butler², and Michael Leuschel³

¹ Department of Computing, University of Surrey

² Electronics and Computer Science, University of Southampton

³ Institut für Informatik, Heinrich-Heine Universität Düsseldorf

Abstract. Symmetry reduction is a model checking technique that can help alleviate the problem of state space explosion, by preventing redundant state space exploration. In previous work, we have developed three effective approaches to symmetry reduction for B that have been implemented into the PROB model checker, and we have proved the soundness of our state symmetries. However, it is also important to show our techniques are sound with respect to standard model checking, at the algorithmic level. In this paper, we present a retrospective B development that addresses this issue through a series of B refinements. This work also demonstrates the valuable insights into a system that can be gained through formal modelling.

Keywords: B, refinement, model-checking, symmetry reduction.

1 Introduction

The B language is an established formal modelling notation whose salient feature is its support for the incremental refinement of abstract specifications into concrete implementations. A B specification (machine) comprises a collection of variables and operations that may manipulate these variables, together with an invariant on the variables.

Formal verification in B typically requires the use of semi-automatic theorem provers (e.g., B4Free [1], Atelier-B [2], the B-Toolkit [3] and Rodin [4]) to prove that the operations of a machine preserve the invariant, and that each refinement is valid. Model checking is a valuable, alternative approach that can perform these tasks automatically, as with the PROB model checker [5].

Previously, we have focused on addressing the *state space explosion* challenge that faces model checking [6,7,8]. This is where a linear increase in the size of a specification leads to a combinatorial increase in the number of states that the model checker must explore. The impact is that checking large specifications becomes intractable. Our work relied on the identification of *symmetric* states that satisfy the same predicates [6, Theorem 1], and the implementation of an augmented model checking algorithm in PROB that checks only one state from each symmetry class. Experimental results were encouraging and have been shown to reduce the time of model checking by up to three orders of magnitude, e.g., [8].

Moreover, these techniques have been integrated into the final version of the tool. Complementary to this work, it is also important to guarantee the soundness of our approaches, with respect to standard model checking. That is, if standard model checking exhausts its search space without finding an error, called a *counterexample*, then it must be guaranteed that symmetry reduced checking exhausts its constrained search space without finding a counterexample. In [6], we sketched a proof that shows this. In this paper, we go a step further and present a complete B development that shows the soundness of our methods through B refinement. In doing so, we provide details of the model checking algorithms used in terms of their key variables, and we make clear the system properties that contribute to the soundness result.

The B development we present was specified and proved interactively using B4Free’s graphical interface, Click’n Prove [9]. Alternatively, we could have used the next generation of B, Event-B [10] and the Rodin tool [4]. However, we did not find our choice inhibited development. Instead, as is common to formal modelling in general, the most time-consuming aspect was the iterative process of finding a suitable abstraction of the system that captures the information we required, in addition to discovering invariants important for refinement.

We proceed by presenting the abstract specification for model checking in Section 2 and an immediate refinement in Section 3. Section 4 provides a refinement machine whose behaviour closely models standard model checking in PROB. Next, the refinement in Section 5.1 adheres to our style of symmetry reduction implemented in [6], and Section 5.2 gives a refinement that matches our symmetry reduction strategy used in [7,8]. Finally, we provide a discussion of our work in Section 6. For clarity of presentation, each machine is broken into several parts, which are individually explained. Each machine specifies the same set of operations, as required by B refinement, although they are only included in the commentary when necessary.

2 An Abstract Specification for Model Checking

The abstract specification, *mc0*, introduces the sets and constants that are required to capture the overall behaviour of a model checking procedure, as used by PROB. These are used to specify two mutually exclusive events that determine when model checking can terminate. We begin by introducing the sets, constants and properties used by this machine. The B encoding is given in Figure 1.

The *mc0* machine uses two sets, S and *ANSWER*. Deferred set, S denotes all possible states of the system being model checked (i.e., the cartesian product of types of the machine variables). Given that bounds are placed on system types during model checking in PROB, $|S|$ is finite. The enumerated set, *ANSWER* denotes the two, mutually exclusive, choices of message that are output once model checking terminates; either *Pass* (the reachable search space has been exhausted without finding a state that violates the invariant, i.e., a counterexample), or *Fail* (a reachable counterexample has been found).

There are four important constants used for the abstract specification (see also Figure 3). Defining the behaviour of the system is *tr*, the transition relation over

```

MACHINE mc0
SETS
  S; ANSWER = {Pass,Fail}
CONSTANTS
  i, /* special initial state */
  tr, /* transition relation */
  inv, /* states satisfying invariant */
  reach /* reachable states */
PROPERTIES
  tr ∈ S ↔ S ∧
  inv ∈ ℙ(S) ∧
  i ∈ inv ∧
  i ∉ ran(tr) ∧
  /* the reachable states */
  reach ∈ ℙ(S) ∧
  i ∈ reach ∧
  /* reach is a fix-point */
  tr[reach] ⊆ reach ∧
  /* reach is the smallest fix-point of
  the reachable states */
  ∀(r).(r ∈ ℙ(S) ∧
  i ∈ r ∧
  tr[r] ⊆ r ⇒
  reach ⊆ r)

```

Fig. 1. The Sets, Constants and Properties of the Abstract Machine, *mc0*

states in S . The set of correctness conditions checked by the algorithm is defined implicitly through *inv*; the subset of S satisfying the correctness conditions. Such an approach is sufficient for the standard model checking of B systems in PROB, since checking involves only the evaluation of the invariant for the variables values [4]. A special state, i , is used to indicate the case where the variables used by the specification have not yet been initialised. Successors of i represent the initialisation of a machine, $tr[\{i\}]$. It follows that i is always the root state of the search space. The set of states encountered during model checking, denoted *reach*, is defined by a fix-point on *tr*, where $tr[reach] \subseteq reach$, i.e., the successor of a reachable state is also reachable. Further, we specify *reach* as the least fix-point of *tr*.

```

ok ← pass ≐
  WHEN reach ⊆ inv
  THEN ok := Pass
  END;
ok ← fail ≐
  WHEN reach ⊈ inv
  THEN ok := Fail
  END

```

Fig. 2. The Operations of the Abstract Machine, *mc0*

The operations of *mc0* are given in Figure 2. These include *pass* and *fail*, which are mutually exclusive events that specify the conditions under which model checking terminates. The *pass* operation is enabled if all reachable states satisfy the correctness conditions used during checking ($reach \subseteq inv$). In which case, the *Pass* message is specified as a return parameter. Conversely, *fail* is enabled if the set of reachable states do not satisfy the correctness conditions, and the *Fail* message is output by the algorithm. In contrast to an implementation of a model checking algorithm, this abstract specification either immediately passes or fails. However, this is sufficient since its single goal is to capture the key properties of

¹ PROB also supports the bounded verification of LTL formulae [11].

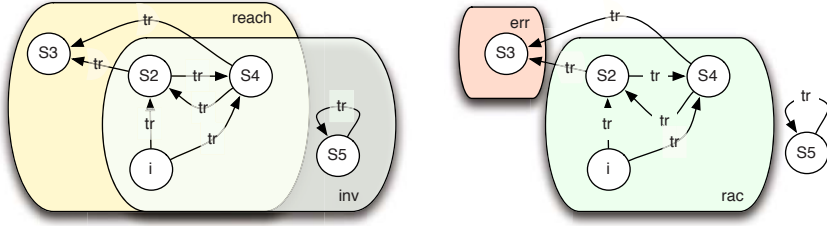


Fig. 3. Illustrating the constants of $mc0$ and variables of $mc1$

the procedure. Details used by an implementation, such as variable information, are given in refinements of $mc0$.

3 Refinement Level 1

Let us now present $mc1$, the first level of refinement for $mc0$ (i.e., $mc0 \sqsubseteq mc1$). This refinement introduces two key variables and two events that will be required in an implementation of a model checking algorithm. Their use is generalised so that later refinements can specify their precise roles during both standard and symmetry reduced model checking. We found that this modularised the proof effort required for these algorithms. Note that this generalisation was devised *after* developing and attempting to prove the separate models for the two algorithms (presented in Sections 4 and 5), when we realised that this refinement was a common abstraction that facilitates proof. Figure 4 presents the new variables, invariant and initialisation clauses of $mc1$ (see also Figure 3).

Variable rac is introduced to store all states reached by model checking so far, which satisfy the correctness conditions. Conversely, err stores those states reached by model checking that violate the correctness conditions.

Regarding the operations, $mc1$ introduces two events used during the *traversal* of the state space. The operation, add_inv , models the checking of states that satisfy the correctness conditions (and in later refinement machines also determines states yet to be checked). Conversely, add_err , models the checking of counterexamples. We separate the events for state space traversal since we find this style convenient for proof. The operations of $mc1$ are given in Figure 5.

REFINEMENT $mc1$

REFINES $mc0$

VARIABLES

rac , /* reached and checked */
 err /* reached errors */

INVARIANT

$rac \subseteq reach \wedge$

$rac \subseteq inv \wedge$

$i \in rac \wedge$

$err \subseteq reach \setminus inv$

INITIALISATION

$rac := \{i\} \parallel$

$err := \emptyset$

Fig. 4. The Variables, Invariant and Initialisation of $mc1$

$add_inv \hat{=} /* new event */$ <p>ANY ss WHERE</p> $ss \subseteq reach \setminus rac \wedge$ $ss \subseteq inv \wedge$ $ss \neq \emptyset$ <p>THEN</p> $rac := rac \cup ss$ <p>END;</p> $ok \leftarrow pass \hat{=}$ <p>WHEN</p> $reach \subseteq rac$ <p>THEN</p> $ok := Pass$ <p>END;</p>	$add_err \hat{=} /* new event */$ <p>ANY ss WHERE</p> $ss \subseteq reach \setminus rac \wedge$ $ss \neq \emptyset \wedge$ $ss \cap inv = \emptyset$ <p>THEN</p> $err := err \cup ss$ <p>END;</p> $ok \leftarrow fail \hat{=}$ <p>WHEN</p> $err \neq \emptyset$ <p>THEN</p> $ok := Fail$ <p>END</p>
--	---

Fig. 5. The Operations of *mc1*

Observe that the *add_inv* event selects a non-empty subset from the reachable states, which are yet to be reached, and which also satisfy the correctness conditions. This subset is added to *rac*, ensuring they will not be encountered again. Similarly, the *add_err* event selects a non-empty subset from the reachable states, yet to be reached, but which contain no elements satisfying the correctness conditions, i.e., are invariant violations. These violations are added to *err* for a permanent record.

We refine the *pass* operation by specifying its guard as $reach \subseteq rac$. That is, *pass* should become enabled when the reachable search space has been fully covered by *add_inv*. To show the validity of a refined event, we must prove that the guard of the abstract operation (G) can be derived from the new guard (G') together with the machine invariant (Inv), i.e., $Inv \wedge G' \Rightarrow G$ [12]. This is straightforward, since $rac \subseteq inv \wedge reach \subseteq rac \Rightarrow reach \subseteq inv$. We also change the guard of the *fail* operation to simply, $err \neq \emptyset$, which is intuitive because its satisfaction indicates an error has been encountered by the *add_err* operation. Proving that this refinement is valid is also simple since, $err \subseteq reach \setminus inv \wedge err \neq \emptyset \Rightarrow reach \not\subseteq inv$.

Given that we are model checking a finite state system, it is desirable to prove the termination of the state space exploration algorithm specified in *mc1*, which occurs when *pass* or *fail* enables. This can be shown by providing the variant, $|reach \setminus (rac \cup err)|$, which represents the number of remaining states yet to be explored. Then, we note that successive applications of *add_inv* and *add_err* decreases the value of the variant progressively, until at some point no new states can be added to *rac* or *err*, and therefore, *add_inv* or *add_err* can no longer be enabled. This ensures that *pass* or *fail* will eventually engage. In the case where errors exist, *fail* enables. If *add_inv* and *add_err* block, then all reachable states have been checked, without error, and *pass* enables. Thus, we have shown the algorithm specified in *mc1* terminates. The addition of variants to a system is

not supported in classical B and its B provers². However, we have provided a variant here to help illustrate the validity of *mc1*.

4 Refinement for Standard Model Checking

The B machines *mc0* and *mc1* given in the previous sections are specified at a high level: certain details are not included that would be required for an implementation of the algorithm. This section addresses this issue through a single refinement of *mc1* that specifies more closely the standard model checking algorithm, and as a consequence, highlights several key properties. Figure 6 shows the variables, invariant and initialisation clauses of this machine.

<p>REFINEMENT <i>mc2</i></p> <p>REFINES <i>mc1</i></p> <p>VARIABLES</p> <p><i>unex</i>, /* reached not fully explored */</p> <p><i>rac</i>, /* reached and checked */</p> <p><i>err</i> /* reached errors */</p>	<p>INVARIANT</p> <p>$unex \subseteq rac \wedge$</p> <p>$tr[rac \setminus unex] \subseteq rac \cup err$</p> <p>INITIALISATION</p> <p>$unex := \{i\} \parallel$</p> <p>$rac := \{i\} \parallel err := \emptyset$</p>
---	--

Fig. 6. The Variables, Invariant and Initialisation of *mc2*

As can be seen, *mc2* introduces a single variable, *unex*. The purpose of this variable is to store all states reached by model checking so far, which satisfy the correctness conditions, but whose successors are yet to be determined. Moreover, it is defined as a subset of *rac*, since each state it stores will be reached via the transition relation from the root state *i*, and subsequently checked.

In addition, note that a new invariant condition is added: $tr[rac \setminus unex] \subseteq rac \cup err$. This constitutes the basis of proving when model checking can terminate, given that no violations exist. To clarify its use, we first present the behaviour of the operations in this machine, given in Figure 7.

We introduce the *remove* operation to remove a state from *unex* whenever all of its successors have been reached, and therefore are elements of *rac*. The repeated application of *remove* will cause *unex* to diminish in size, indicating that fewer transitions remain to be explored. This can be expressed formally as a simple variant, $|unex|$, whose size decreases upon the action of *remove*.

The *add_inv* event of *mc1* is refined to select a single state from *unex* (a state whose transitions have not yet all been traversed), and computes a single successor of it (*s2*) that satisfies the correctness conditions. The successor is added to both *unex* and *rac*. In the case where the successor is an invariant violation, it is added to only *err* in the *add_err* operation. Addition to either *unex* or *rac* would, otherwise, break the invariant, $unex \subseteq rac \wedge rac \subseteq inv$.

² Event-B and its associated provers provide support for variants.

```

add_inv ≐
ANY s1,s2 WHERE
  s1 ∈ unex ∧
  s2 ∈ inv ∧
  s1 ↦ s2 ∈ tr ∧
  s2 ∉ rac ∧
  err = ∅
THEN
  unex := unex ∪ {s2} ||
  rac := rac ∪ {s2}
END;

add_err ≐
ANY s1,s2 WHERE
  s1 ∈ unex ∧
  s2 ∉ inv ∧
  s1 ↦ s2 ∈ tr ∧
  err = ∅
THEN
  err := err ∪ {s2}
END;

remove ≐ /* new event */
ANY s1 WHERE
  s1 ∈ unex ∧
  /* all s1's successors checked */
  tr[{s1}] ⊆ rac ∧
  err = ∅
THEN
  unex := unex \ {s1}
END;
ok ← pass ≐
WHEN
  unex = ∅ ∧
  err = ∅
THEN
  ok := Pass
END;
ok ← fail ≐
WHEN err ≠ ∅
THEN
  ok := Fail
END

```

Fig. 7. The Operations of *mc2*

A number of assertions are also specified in *mc2*, to verify the preservation of responsiveness of the specified model checking algorithm³. We do not show them because they simply consist of a disjunction of the guards of each operation. Their proof with B4Free guarantees that there is always at least one enabled operation, e.g., model checking has not yet finished, so one can perform either *add_inv*, *add_err* or *remove*, or conversely, state space exploration has terminated and either *pass* or *fail* is enabled.

Given the responsiveness of this machine, in addition to the previous variants specified for the *add_inv*, *add_err* and *remove* operations, which show that eventually these operations are all blocked, we can deduce that either *pass* or *fail* will eventually be enabled. This relies on *pass* and *fail* being valid refinements of their abstract specification. This is trivial for the *fail* operation, since it remains unchanged from *mc1*. The goal for the *pass* operation is to show that $Inv \wedge unex = \emptyset \wedge err = \emptyset \Rightarrow reach \subseteq rac$. By choosing the appropriate invariant, we have:

$$\begin{aligned}
 & tr[rac \setminus unex] \subseteq rac \cup err \\
 & = tr[rac] \subseteq rac \qquad \text{Given } unex = \emptyset \text{ and } err = \emptyset
 \end{aligned}$$

That is, *rac* is a fix-point of *tr*. Since *reach* is the least fix-point, we can conclude that $reach \subseteq rac$.

³ An assertion in B is an expression over the sets, constants, properties, variables or invariant clauses of a B machine. They enable one to form corollaries in B. By proving an assertion, it is made available for use inside other proof activities.

The overall chain of refinement developed for standard model checking consists of: $mc0 \sqsubseteq mc1 \sqsubseteq mc2$. That is, $mc2$ is a valid refinement of $mc0$. Therefore, the model checking algorithm specified in $mc2$ is sound with respect to the abstract specification of model checking. In the next section, we introduce the notion of symmetry reduction into our specifications.

5 Refinements for Symmetry Reduced Model Checking

This section presents two refinement machines that specify symmetry reduced model checking through the refinement of $mc1$ (Section 3), namely $rmc1$ and $rmc2$. These refinements follow closely the specification of $mc2$, except they introduce the concept of symmetry between states of a system.

5.1 Level 1

The primary purpose of the first refinement machine for symmetry reduced model checking is to provide the first step towards integrating symmetry reduction into the B specification of standard model checking, whilst linking the variables used by the standard and reduced approaches. Through this machine, we also show that our original work in symmetry reduction [6] is sound with respect to the abstract specification of model checking. In this particular strategy, called *permutation flooding*, each unexplored state encountered is first checked against the invariant. Then, all states symmetric to it (which we have proved satisfy the same predicates) are computed and are added to the state space: these states are marked as explored so that model checking need not explicitly check them. The concept of state symmetries is specified using constants and properties, and is given in Figure 8.

The symmetries of a system are defined over the transition relation in terms of sets of special permutations (called *automorphisms*), denoted *aut*. We also specify two key properties of automorphisms, as given in [13, Chapter 14]:

- an inverse of an automorphism is itself an automorphism, and
- automorphisms preserve the transition relation (a result also shown in [6]).

In the context of this specification, we define that the special state i (representing the uninitialised machine) is symmetric only to itself. In addition, we specify a consequence of a result in [6, Corollary 1], which proves that symmetric states satisfy the same predicates. That is, a state satisfies the invariant, *iff* states symmetric to it also satisfy the invariant.

A valid automorphism p for the example from Figure 3 is shown in Figure 9 (dashed lines represent the transition relation), where $S2$ and $S4$ are permuted for each other and all other states are kept unchanged. In terms of a B machine, a state comprises the values of its variables. Intuitively, two states, such as $S2$ and $S4$, are symmetric if the values of one state can be transformed into those of the other. In addition, a sequence of state transitions (i.e., operations) possible from one state will also be possible from the other; this is also depicted in Figure 9.

REFINEMENT <i>rmc1</i>	<i>/* P1: automorphisms preserve tr */</i>
REFINES <i>mc1</i>	$\forall(p,s1,s2).(p \in aut \wedge s1 \in S \wedge$
CONSTANTS	$s2 \in S \Rightarrow$
<i>aut, /* automorphisms on tr */</i>	$(s1 \mapsto s2 \in tr) \Leftrightarrow$
<i>rep /* representative function */</i>	$(p(s1) \mapsto p(s2) \in tr)) \wedge$
PROPERTIES	
$aut \in \mathbb{P}(S \twoheadrightarrow S) \wedge$	<i>/* symmetries have same rep. */</i>
$id(S) \in aut \wedge$	$\forall(p,s1,s2).(p \in aut \wedge$
$\forall(p).(p \in aut \Rightarrow p^{-1} \in aut) \wedge$	$s1 \mapsto s2 \in p \Rightarrow$
$\forall(p).(p \in aut \Rightarrow i \mapsto i \in p) \wedge$	$rep(s1) = rep(s2)) \wedge$
<i>/* automorphisms preserve invar. */</i>	<i>/* s and rep(s) implies auto. */</i>
$\forall(p,s1,s2).(p \in aut \wedge$	$\forall(s1,s2).(s1 \mapsto s2 \in rep \Rightarrow$
$s1 \mapsto s2 \in p \Rightarrow$	$\exists(p).(p \in aut \wedge s1 \mapsto s2 \in p)) \wedge$
$(s1 \in inv) \Leftrightarrow (s2 \in inv)) \wedge$	
$rep \in S \rightarrow S \wedge$	<i>/* representatives are fix-points */</i>
	$\forall(s).(s \in ran(rep) \Rightarrow rep(s) = s)$

Fig. 8. The Constants and Properties of the Machine, *rmc1*

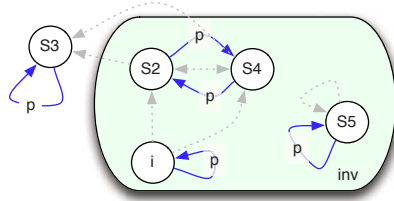


Fig. 9. An automorphism for Figure 3

The constant, *rep*, is introduced to model an algorithm that computes a unique representative for some given state, and is defined over the set of states *S*. We have implemented this function in PROB, which determines a representative state from the set of states symmetric to it [6].

It follows that checking one state during the reduced search, corresponds to checking all symmetric states in the standard search. The *rep* function in this refinement is constrained accordingly (the first 3 properties involving *rep*). Further, we specify representatives as fix-points. Assertions for *rmc1* are given in Figure 10, whose proof simplifies later proof activities required to guarantee its consistency and show that it is a valid refinement of *mc1*.

There are five assertions defined for this machine, of which the first four are relatively simple and follow from the properties of *aut* and *rep*. The last assertion requires proof that for any reachable state its representative state is also reachable. To show this it is instructive to present a fix-point proof over automorphisms, upon which *rep* is based. Using the property of automorphisms

ASSERTIONS

$$\begin{array}{ll}
/* \text{ representatives preserve invar. } */ & \forall (s1, s2). (s1 \mapsto s2 \in tr \Rightarrow \\
\forall (s1, s2). (s1 \in S \wedge & \exists (ss2). (rep(s1) \mapsto ss2 \in tr \wedge \\
s2 \in S \wedge & rep(s2) = rep(ss2))) \wedge \\
s1 \mapsto s2 \in rep \Rightarrow & \\
((s1 \in inv) \Leftrightarrow (s2 \in inv)) \wedge & /* s is reachable iff \\
& rep(s) is reachable */ \\
rep(i) = i \wedge & \forall (s). (s \in S \Rightarrow \\
rep^{-1}[\{i\}] = \{i\} \wedge & ((s \in reach) \Leftrightarrow (rep(s) \in reach)))
\end{array}$$

Fig. 10. The Assertions of *rmc1*

marked *P1* in Figure 8, we begin by proving for any automorphism p , that $p[reach]$ is a fix-point of tr :

$$\begin{array}{ll}
tr[p[reach]] \subseteq p[reach] & \text{(A)} \\
\Leftrightarrow \forall y \cdot y \in tr[p[reach]] \Rightarrow y \in p[reach] & \text{inclusion is universal} \\
\Leftrightarrow (\exists x \cdot x \in p[reach] \wedge x \mapsto y \in tr) & \text{quantify on } p \\
\Rightarrow y \in p[reach] & \\
\Leftrightarrow (\exists x \cdot p^{-1}(x) \in reach \wedge x \mapsto y \in tr) & p \text{ is injective} \\
\Rightarrow p^{-1}(y) \in reach & \\
\Leftrightarrow (\exists x \cdot p^{-1}(x) \in reach \wedge p^{-1}(x) \mapsto p^{-1}(y) \in tr) & \text{property } P1 \\
\Rightarrow p^{-1}(y) \in reach & \\
\Leftrightarrow true &
\end{array}$$

Equation (A) implies $p[reach]$ is a fix-point of tr . Thus, for an automorphism q :

$$reach \subseteq q[reach] \quad \text{(B)}$$

By monotonicity, from (B) we get:

$$\begin{array}{ll}
q^{-1}[reach] \subseteq q^{-1}[q[reach]] & \\
\Leftrightarrow q^{-1}[reach] \subseteq reach & q \text{ is injective} \quad \text{(C)}
\end{array}$$

Instantiate q with p in (B) to get:

$$reach \subseteq p[reach] \quad \text{(D)}$$

Instantiate q with p^{-1} in (C) to get:

$$\begin{array}{ll}
(p^{-1})^{-1}[reach] \subseteq reach & \\
\Leftrightarrow p[reach] \subseteq reach & p \text{ is injective} \quad \text{(E)}
\end{array}$$

Finally, from (D) and (E) we obtain the result $p[reach] = reach$. That is, all automorphisms preserve the reachable states.

VARIABLES

```
/* vars for standard checking */
rac, unex, err,
/* vars for reduced approach */
rrac, runex, rerr
```

INVARIANT

```
unex  $\subseteq$  rac  $\wedge$ 
rrac  $\subseteq$  ran(rep)  $\wedge$ 
rrac  $\subseteq$  rac  $\wedge$ 
runex  $\subseteq$  rrac  $\wedge$ 
```

```
rerr  $\subseteq$  err  $\wedge$ 
rep-1[rrac] = rac  $\wedge$ 
rep-1[runex] = unex  $\wedge$ 
rep-1[rerr] = err  $\wedge$ |[rac  $\setminus$  unex]  $\subseteq$  rac  $\cup$  err

```

INITIALISATION

```
rac := {i} || rrac := {i} ||
unex := {i} || runex := {i} ||
err :=  $\emptyset$  || rerr :=  $\emptyset$ 
```

Fig. 11. The Variables, Invariant and Initialisation of *rmc1*

Six variables are used by this machine, and are shown in Figure 11. Intuitively, they can be split into three pairs, where each pair consists of a variable used in the B specification of standard model checking (*rac*, *unex* or *err*), and a corresponding variable introduced to specify reduced checking (*rrac*, *runex* or *rerr*). The key premise is to link each pair with some set of constraints, so that properties that apply to standard checking also apply to the reduced approach.

As with the standard approach to checking, the set of states reached during checking whose successors have not yet all been explored (*unex*), is a subset of the states encountered by model checking (*rac*); $unex \subseteq rac$. To link *rac* and *rrac*, we specify that $rrac \subseteq rac$ and $rep^{-1}[rrac] = rac$; the states symmetric to those of *rrac* are members of *rac*. We specify corresponding constraints for variables *unex*, *runex*, *err*, and *rerr*. In addition, $tr[rac \setminus unex] \subseteq rac \cup err$ is specified to simplify the detection of model checking termination when no counterexamples are found (i.e., when $unex = \emptyset$ and $err = \emptyset$, see *mc2* in Section 4). This will be proved correct in the next refinement using only *rrac*, *runex*, and *rerr*. The operations of *rmc1* are given in Figure 12.

Notice that this machine behaves in a similar way to *mc2*, which also refines *mc1*. The difference regarding the *add_inv* or *add_err* events, is that for each newly encountered state *s* we add its *representative* to *runex* (if *s* satisfies the invariant) or *rerr* (if *s* violates the invariant); while adding all symmetric states, $rep^{-1}[\{s\}]$ to *unex* or *err*. The *remove* operation follows this pattern, and removes a state from *runex* whenever the representatives of all of its successors have been encountered; while all symmetric states are then removed from *unex*.

Justification of the correctness of this refinement is similar to the standard case, presented in Section 4. This involved proving the enabledness preservation of operations, the validity of the refinement and that eventually *pass* or *fail* becomes enabled.

Soundness Result 1: The important observation of this refinement machine is that the style of state space traversal provided by the operations *add_inv*, *add_err* and *remove*, reflects the algorithm we used in our initial work on symmetry reduction in PROB, i.e., permutation flooding. For example, $rep^{-1}[\{rep(s2)\}]$ in the *add_inv* operation in Figure 12 represents all symmetric states of *s2*, which are used to *flood* the variables, *unex* and *rac*. We obtain the assurance that

```

add_inv ≐
ANY s1,s2 WHERE
  s1 ∈ runex ∧
  s2 ∈ inv ∧
  s1 ↦ s2 ∈ tr ∧
  rep(s2) ∉ rrac ∧
  rerr = ∅
THEN
  runex := runex ∪ {rep(s2)} ||
  unex := unex ∪ rep-1[{rep(s2)}] ||
  rrac := rrac ∪ {rep(s2)} ||
  rac := rac ∪ rep-1[{rep(s2)}]
END;
add_err ≐
ANY s1,s2 WHERE
  s1 ∈ runex ∧
  s2 ∉ inv ∧
  s1 ↦ s2 ∈ tr ∧
  rep(s2) ∉ rrac ∧
  rerr = ∅
THEN
  rerr := rerr ∪ {rep(s2)} ||
  err := err ∪ rep-1[{rep(s2)}]
END;

remove ≐
ANY s1 WHERE
  s1 ∈ runex ∧
  /* all s1's successors checked */
  rep[tr[{s1}]] ⊆ rrac ∧
  rerr = ∅
THEN
  runex := runex \ {s1} ||
  unex := unex \ rep-1[{s1}]
END;

ok ← pass ≐
WHEN
  rerr = ∅ ∧
  runex = ∅
THEN
  ok := Pass
END;
ok ← fail ≐
WHEN
  rerr ≠ ∅
THEN
  ok := Fail
END

```

Fig. 12. The Operations of *rmc1*

permutation flooding is sound with respect to the abstract specification of standard model checking, since $mc0 \sqsubseteq mc1 \sqsubseteq rmc1$.

5.2 Level 2

In the final refinement for symmetry reduction, we retain only three variables, *rrac*, *runex* and *rerr*, from the relatively detailed *rmc1*, upon which we specify a minimal set of constraints, as shown in Figure 13.

REFINEMENT <i>rmc2</i>	INVARIANT	INITIALISATION
REFINES <i>rmc1</i>	$i \in rrac \wedge$ $rrac \subseteq \text{ran}(\text{rep}) \wedge$ $rrac \subseteq \text{rac} \wedge$ $\text{runex} \subseteq \text{rrac} \wedge$ $\text{rerr} \subseteq \text{err}$	$\text{rrac} := \{i\} \parallel$ $\text{runex} := \{i\} \parallel$ $\text{rerr} := \emptyset$
VARIABLES <i>rrac, runex, rerr</i>		

Fig. 13. The Variables, Invariant and Initialisation of *rmc2*

Observe that the specification of the variables remains the same as that given in *rmc1*, while all details of *rac*, *unex*, and *err* have been removed. The same applies for the operations of this machine: *add_inv*, *add_err* and *remove* are identical to those in *rmc1*, except that there are no assignments to *rac*, *unex*, and *err*. For this reason, we do not show the operations of this refinement.

We note that the style of state space traversal specified contrasts with that of *rmc1* and instead reflects more closely a classical symmetry reduction algorithm, which we used in [7,8]. Therefore, upon encountering an unexplored state (e.g., *s1* in *add_inv*), we compute and store only the unique representatives of its successors that have not yet been checked (*rep(s2)*); the model checking algorithm will never store two symmetric states, and it has less of a demand for memory. The disadvantage of implementing such a *rep* function is that it can be computationally expensive⁴. We also note that the proof of correctness for *rmc1* is echoed by this machine.

Soundness Result 2: The chain of refinement for our classical approach to symmetry reduced model checking consists of: $mc0 \sqsubseteq mc1 \sqsubseteq rmc1 \sqsubseteq rmc2$. Therefore, by the transitivity of refinement, our augmented algorithm is sound with respect to the abstract specification of model checking.

6 Concluding

We have presented a B development that shows through refinement the soundness of our previous methods for symmetry reduction in PROB, with respect to standard model checking. That is, if standard model checking exhausts its search space without finding a counterexample then symmetry reduced checking must also exhaust its quotient search space without finding a counterexample.

An abstract specification for model checking, *mc0*, is given in Section 2, which is refined by *mc1* in Section 3. From here, two separate two chains of refinement specify details of algorithms that implement the standard and reduced approaches. The refinement branch for the reduced approach includes *rmc1*, which reflects the style of model checking we adopted in [6], and *rmc2*, that reflects the style we used in [7] and [8]. Given that both chains refine the abstract specification, we obtain our desired soundness result.

The system was specified using B and the Click'n Prove tool, although it would have been possible to use Event-B. Indeed, our use of guarded B operations is characteristic of events in Event-B. In addition, we could have utilised the tool support of Event-B when guaranteeing the model checking algorithms eventually terminate having found a counterexample (*fail*) or without finding a counterexample (*pass*), after exploring the reachable state space. This task involved using variants to ensure the *add_inv*, *add_err* and *remove* operations eventually relinquish control (giving *pass* and *fail* an opportunity to be enabled), and proving the preservation of operation enabledness for the system. Despite this, we did not find using B impeded the development process. We recognise though, that if our development had become more complex (e.g., requiring decomposition), it would have been beneficial to use Event-B and its tools.

The B development presented captures properties of model checking that are sufficient to show the overall soundness of our approaches to symmetry reduction.

⁴ This *rep* function is based upon algorithms for determining isomorphic graphs, for which there is currently no known polynomial time algorithm.

In these specifications, we have removed the details of the algorithms that select a unique representative from a class of symmetric states; as modelled by the *rep* function. Proving that our implementations correctly compute representatives currently remains as future work and would require developing additional formal models. We do not believe this would be difficult for our permutation flooding approach, since the implementation relies on a simple, but effective, permutation function. However, we do think this would be challenging for our two other implementations, which use complex algorithms for determining graph isomorphism, and were inspired by the work of McKay [14]. Additional future work is to extend our B development by adding labels to *tr* so that properties can be proved over paths of the system. This would provide a basis for proving the soundness of refinement via model checking. Finally, it would also be valuable to prove that symmetry reduced model checking preserves LTL properties.

References

1. Clearys: B4Free tool (2009), <http://www.b4free.com>
2. Steria, Aix-en-Provence, France: Atelier B, User and Reference Manuals (2009), <http://www.atelierb.eu/index-en.php>
3. B-Core (UK) Limited: B-Toolkit manuals (2002), <http://www.b-core.com/btoolkit.html>
4. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
5. Leuschel, M., Butler, M.J.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
6. Leuschel, M., Butler, M.J., Spermann, C., Turner, E.: Symmetry Reduction for B by Permutation Flooding. In: Julliard, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 79–93. Springer, Heidelberg (2006)
7. Turner, E., Leuschel, M., Spermann, C., Butler, M.J.: Symmetry Reduced Model Checking for B. In: TASE, pp. 25–34. IEEE Computer Society, Los Alamitos (2007)
8. Spermann, C., Leuschel, M.: ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In: TASE, pp. 15–22. IEEE Computer Society, Los Alamitos (2008)
9. Abrial, J.R., Cansell, D.: Click’n Prove: Interactive Proofs within Set Theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 1–24. Springer, Heidelberg (2003)
10. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language, RODIN, D7 (2005)
11. Leuschel, M., Plagge, D.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In: Ameer, Y.A., Boniol, F., Wiels, V. (eds.) ISO/LA. RNTI-SM-1 of Revue des Nouvelles Technologies de l’Information, Cépaduès-Éditions, pp. 73–84 (2007)
12. Abrial, J.R.: The B Book: Assigning programs to meanings. Cambridge University Press, New York (1996)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
14. McKay, B.D.: Practical Graph Isomorphism. *Congressus Numerantium* 30, 45–87 (1981)

Development of a Synchronous Subset of AADL^{*}

Mamoun Filali-Amine¹ and Julia Lawall²

¹ IRIT-CNRS, Université de Toulouse, 118 route de Narbonne, F-31062 Toulouse, France

² DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen, Denmark

Abstract. We study the definition and the mapping of an AADL subset: the so called synchronous subset. We show that the data port protocol used for delayed and immediate connections between periodic threads can be interpreted in a synchronous way. In this paper, we formalize this interpretation and study the development of its mapping such that the original synchronous semantics is preserved. For that purpose, we use refinements through the Event B method.

1 Introduction

Model-based design has emerged as one of the most important design paradigms in recent years. High level models allow the developer to concentrate on the functionality to be offered rather than implementation details. The Architecture Analysis and Design Language (AADL) [11] is by now considered as a mature alternative for modeling embedded and real time systems. AADL has been standardized by the SAE [19], and features of AADL have influenced the MARTE OMG standard [10]. As a successor of the MetaH language [16] developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications, AADL capitalizes on more than 10 years of experience. AADL also builds on the experience acquired during the development of Architecture Description Languages (ADLs) such as ACME and Wright [3].

In this paper, we study the AADL data port protocol, which defines the semantics of delayed and immediate connections between periodic threads. This is a fundamental protocol that lies at the heart of any embedded AADL-based platform. We show that the data port protocol can be interpreted in a synchronous way [6]. Nevertheless, this interpretation does not provide a satisfactory basis for implementation in embedded systems, as the stack depth entailed by recursive calls is only bounded by the least common multiple of the periods of all of the threads, which can be very large. We thus present the development of a mapping of the synchronous semantics of the AADL data port protocol into an iterative implementation, such that the original synchronous semantics is preserved. For this purpose, we use refinements through the Event B method [2].

After a brief overview of AADL and Event B in Sections 2 and 3, we motivate the proposed development in Section 4. Section 5 presents the successive refinements of the development. Section 6 outlines the validation of the development. Before concluding, we review some related work in Section 7.

^{*} This work was partly supported by the French AESE project Topcased and by the region Midi-Pyrénées.

2 AADL

AADL includes all the standard concepts of any ADL, e.g., components, connectors to describe the interface of components, and connections to link components. AADL distinguishes between three kinds of components: software components (process, thread, thread group, subprogram, and data), hardware components (processor, bus, memory, device), and system components.

2.1 AADL Threads

In AADL, threads are the only components that have an execution semantics. AADL supports the classic types of thread dispatch protocols: a thread can be declared to be periodic, aperiodic, sporadic or background. All of the standard properties (worst case execution time (WCET), deadline, etc.) used to describe a real-time system exist in AADL. In the following, we consider periodic threads only. A periodic thread is dispatched within every period.¹ When a thread *completes* its execution, it goes to the “awaiting_dispatch” state until its next period. The thread’s actual execution time is bounded by its WCET and must end by its deadline.

2.2 AADL Data Port Protocol

AADL defines three types of ports: data, event and event data ports. Data ports allow communication via a single word (a register). Event and event data ports represent buffered communications. In this paper, we consider only data ports.

A data port connection can be declared as *delayed* or *immediate*. If the connection is delayed, data is available at the deadline of the sending thread. If the connection is immediate the receiving thread must wait for the sending thread to complete to start its execution. Figure 1 illustrates the instants where execution and these communications take place. One key aspect of the AADL data port protocol is that communications between a thread and its environment occur at well defined instants:

- In general, a message is copied at the dispatch (data event in Fig. 1); in the case of an immediate connection, a message is copied at the start of execution (immediate data in Fig. 1).
- A message is actually sent at the completion, in the case of an immediate connection, or at the deadline, in the case of a delayed connection.

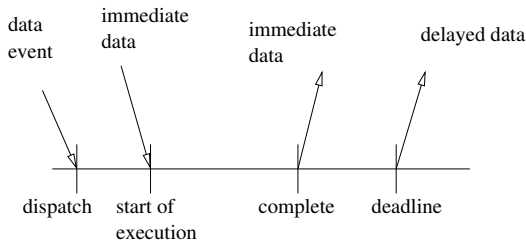


Fig. 1. Communication through data ports in AADL

¹ We consider here AADL V1 which does not take into account the phase of a periodic thread.

3 A Brief Overview of Event B

Event B stems from the B method [11]. One of the goals of Event B is to reason about so called reactive systems [15]. Like B, Event B is based on set theory together with first-order logic. It proposes refinements as the main software development concept. However, instead of B operations, Event B proposes events, which are simpler.

3.1 Basic Principles

The basic structuring concepts of Event B are the *context* and the *machine*. A context contains *sets*, *constants* and their properties (*axioms* and *theorems*). A machine contains a system specification. A system specification *sees* contexts and defines a static and a dynamic part. The static part defines a state space through *variables*. These variables are “typed” and more generally specified by *invariants*. The dynamic part defines a behavior through an initialization event and a set of events. Each event can be considered as a non deterministic guarded command [9]. The guard is specified by a conjunction of predicates and the command is specified as a set of substitutions.

3.2 Notation

For the most part, Event B uses standard set notation. Some notation that is specific to Event B is as follows:

- **pair construction:** Pairs are constructed using the maplet operator \mapsto . A pair is thus denoted $(a \mapsto b)$ instead of (a, b) .
- **restriction to the domain:** $F \triangleleft R = \{x \mapsto y \mid (x \mapsto y) \in R \wedge x \in F\}$
- **overwrite:** $Q \triangleleft R = ((\text{dom}(Q) \setminus \text{dom}(R)) \triangleleft Q) \cup R$

4 Motivation of the Development

In this section, we motivate our development by presenting the specification view of the data port protocol and some features of the operational view. The specification view is intended to be used when reasoning over the protocol. The operational view is intended to be used for an actual implementation. For instance, the operational view takes into account the times where:

- the computations take place: at the period;
- the outputs are made available: at the completion or at the deadline.

The goal of the development is to establish that the operational view refines the specification view.

First, we illustrate the two views through a toy AADL architecture, shown in Figure 2. In this architecture, we have three periodic threads: t_1 , t_2 and t_3 . The period and deadline of t_1 are both 10. That of t_2 are 10 and 5, respectively. That of t_3 are 15 and 5, respectively. Thread t_1 has two output ports o_1, o_2 and two input ports i_4 and i_5 . Thread t_2 has one output port o_5 and two input ports i_1 and i_3 . Thread t_3 has two output ports o_3 and o_4 and one input port i_2 . In each case, output port o_i is connected to input port i_i , via either a delayed (d) or an immediate (i) connection. We furthermore adopt the convention that inside a thread, input and output ports are linked through implicit immediate connections (an output can be linked to any subset of inputs).

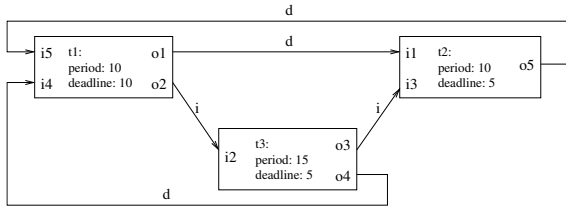


Fig. 2. A toy AADL architecture

4.1 The Specification View

In the specification view, computations are assumed to occur at precise time instants. In this study, we assume that computations occur at the beginning of their period, and do not take time. Still, their results are available only at the deadline. This ensures conformity with an effective implementation in which computations do take time but respect the deadlines. Computations can depend on each other in:

- either a *delayed* way: at time t , the computation for port p depends on the computation that occurred for port p' at its most recent deadline.
- Or in an *immediate* way: at time t , the computation for port p depends on the computation that occurred for port p' at the same time t , if any. The result of this computation is then buffered in p for use in subsequent computations until a new result is available from p' .

These two causality relations are given through the following relations:

$Pred_D \in \mathbb{P}(\text{Port} \times \text{Port})$ // delayed port predecessor relation
 $Pred_I \in \mathbb{P}(\text{Port} \times \text{Port})$ // immediate port predecessor relation

The $Pred_D$ and $Pred_I$ port predecessor relations for the example shown in Figure 2 are the following:

Delayed port predecessor relation	Immediate port predecessor relation
$\{(i1 \mapsto o1), (i4 \mapsto o4), (i5 \mapsto o5)\}$	$\{(i2 \mapsto o2), (i3 \mapsto o3)\}$

Remark. The “hidden” immediate relations between each input port and each output port of a thread are not shown.

The constants C_D (resp. C_I) are used to define the computation tasks for each delayed (resp. immediate port). A computation task is parameterized by

- the identity of the port at the end of a connection,
- the values of the ports at the beginning of a connection, at preceding times for delayed ports or at the current time for immediate ports.

As delayed connections only appear between threads, C_D only transfers values from a single output port to a single input port. C_I does the same for connections between threads, and carries out the thread’s computation for the “hidden” connections within

threads. Then, the recursive `Compute` function is defined for delayed and immediate ports as follows:²

	$\text{Compute}(t)(p)$
$\text{delayed}(p)$ $\wedge t \% \text{period}(p) = 0$	$C_D(p)(\{q \mapsto \text{Compute}(\text{Deadline}(t \mapsto q))(q) \mid q \mapsto p \in \text{Pred_D}\})$
$\text{immediate}(p)$ $\wedge t \% \text{period}(p) = 0$ $\wedge \forall q \in \text{Pred_I}^{-1}(p).$ $t \% \text{period}(q) = 0$	$C_I(p)(\{q \mapsto \text{Compute}(t)(q) \mid q \mapsto p \in \text{Pred_I}\})$
else	$\text{Compute}(t - 1)(p)$

where t is not equal to 0.

Remark: In the preceding table, we have adopted the usual mathematical notation $\{\text{exp} \mid \text{boolean_exp}\}$ for set comprehensions. In event B, the quantified variables would be made explicit and the order of the terms changed as follows: $\{\text{vars. boolean_exp} \mid \text{exp}\}$.

The expression $\text{Compute}(t)(p)$ deserves some explanation. For a delayed port it is computed over the predecessor ports q (according to the `Pred_D` relation) at their respective deadlines: $\text{Compute}(\text{Deadline}(t \mapsto q))(q)$. For an immediate port, it is computed over the predecessor ports q (according to the `Pred_I` relation) at the current time t : i.e., $\text{Compute}(t)(q)$. In general, the value of a port p is computed at its period: $\text{period}(p)$; for an immediate port, it is actually computed if its period aligns with those of its predecessors; otherwise it remains unchanged.

Discussion. This initial specification is functional. It follows that, assuming the termination of the `Compute` function, the specification can be considered as executable. However, we remark that such code cannot be considered as executable in the context of an embedded system. In particular, the memory resources needed for executing such a code are not a priori bounded, i.e., the code does not have the constant space property. Indeed, the depth of the required stack to handle recursive calls depends on the least common multiple of the periods of the various threads. Then, from a technical point of view, the goal of the refinements that will be introduced in Sections 5.3 and 5.4 can be seen as the implementation of this recursivity through bounded memory independently of parameter values.

4.2 The Operational View

The operational view introduces a scheduler that manages the data port architecture. The information needed by this scheduler is given by the `RealTime` context. `Idle` and `Deadline` are the basic structures used by such a scheduler.

`Idle` $\in \mathbb{N} \rightarrow \mathbb{P}(\text{Port})$: for a time t , the list of idle ports.

`Deadline` $\in \text{Port} \times \mathbb{N} \rightarrow \mathbb{N}$: for a delayed port p and a time t , the (time) value of the most recent deadline

We illustrate these static structures by instantiating them according to the architecture shown in Figure 2.

² $\%$ denotes the infix modulo function.

Data structures of the architecture example

- Idle returns the list of idle ports at time t .

Idle	1 – 9	Port
Idle	10	{ $i2, o3, o4$ }
Idle	11 – 14	Port
Idle	15	{ $i1, i3, i4, i5, o1, o2, o5$ }
Idle	16 – 19	Port
Idle	20	{ $i2, o3, o4$ }
Idle	21 – 29	Port

- Deadline is a function which gives for a delayed port p and a time t , the (time) value of the most recent deadline.

Deadline($o1, 0 - 9$)	0	Deadline($o4, 0 - 4$)	0	Deadline($o5, 0 - 4$)	0
Deadline($o1, 10 - 19$)	9	Deadline($o4, 5 - 19$)	4	Deadline($o5, 5 - 14$)	4
Deadline($o1, 20 - 29$)	19	Deadline($o4, 20 - 29$)	19	Deadline($o5, 15 - 24$)	14
				Deadline($o5, 25 - 29$)	24

Algorithm of the scheduler. B does not offer mechanisms for real-time programming, such as dedicated primitives for awaiting clock interrupts. In the proposed Event B machines, guards model the real-time clock triggers. Once an event is triggered, all the enabled events at that time are executed until none of them is still enabled. Then, the processor idles until the next clock tick. In this paper, we consider that the time between two clock ticks is sufficient to handle all the enabled events. In that way, we do not lose any clock tick and all the events take place in zero time according to the synchronous abstraction. The main loop consists of time-triggered iterations. For our toy example, iterations are triggered at 0, 10, 15, 20, 30, etc., which correspond to the periods of the threads of our example. Each iteration first handles the ports that should run at that time and then prepares the next iteration. Handling ports is done through the `ComputeDelayed` and `ComputeImmediate` events. Preparing the next iteration is done through the `Tick` event. In the last refinement, these events are exclusive and deterministic.

Synthesis. The operational view of the considered AADL subset is a mix between a time-triggered machine and a data-flow machine. Ports are updated according to their period. Immediate connections enforce a send-receive synchronization.

5 Abstracting and Refining the AADL Data Port Protocol

In this section, we formalize the fact that if we restrict AADL to connections with the data port protocol, we have a synchronous computation model. For that purpose, we first exhibit a model of the protocol from which we derive another model, based on

histories, close to the description given in the previous section. A third model is derived for considering implementation related issues with respect to the boundedness of the used memory and the time for evaluating new port values. To summarize, we consider the following refinement-based development, where \sqsubseteq is B notation for “refined by”:

MACHINE	Spec	\sqsubseteq	I_Spec	\sqsubseteq	P_Spec	\sqsubseteq	M_Spec	\sqsubseteq	Scheduler
CONTEXT	Ports		I_Ports		M_Ports		S_Compute		

- Spec is the initial specification representing the abstraction of the AADL protocol.
- I_Spec is the refinement where *Idle* ports are introduced.
- P_Spec is the refinement where a *Partition* of ports is introduced. Immediate ports are computed.
- M_Spec is the refinement where port buffering through a memory is introduced. Delayed ports are computed.
- Scheduler is the final refinement where port updates are scheduled according to a total order.

5.1 The Specification

This is the initial specification for the AADL data port protocol as the *time* parameterized function *Compute*. The variable *ports* records the value of this function at each point in time through the *Initialisation* and *Tick* events. This recording is done atomically so that no value returned by *Compute* is lost: between two events, *Compute* stutters.

The static description. We have three variables:

- *t* is the current time,
- *ports* maps ports to their current value,
- *b* is a previous time, such that *ports* has not changed since *b* until *t* (excluded). *inv5* states that we have not missed any value in the interval $b..t - 1$: the range of the *Compute* function over this interval is a singleton.

MACHINE Spec

SEES Ports

VARIABLES t ports b

INVARIANTS

inv1 : $t \in \mathbb{N} \wedge 0 < t$

inv2 : $ports \in Port \rightarrow Val$

inv4 : $b \in \mathbb{N} \wedge b < t$

inv3 : $ports = Compute(b)$

inv5 : $Compute[b..t - 1] = \{Compute(b)\}$

The dynamic description. The basic idea is that, in order to preserve our basic invariant inv5 , the time t is advanced to a new value t' such that ports remain constant from t to $t' - 1$.

Initialisation

begin

act1: $t : | t' \in \mathbb{N} \wedge 0 < t' \wedge \text{Compute}[0..(t' - 1)] = \{\text{Compute}(0)\}$

act2: $\text{ports} := \text{Compute}(0)$

act3: $b := 0$

end

Event Tick $\hat{=}$

begin

act1: $t : | t' \in \mathbb{N} \wedge t < t' \wedge \text{Compute}[t..(t' - 1)] = \{\text{Compute}(t)\}$

act2: $\text{ports} := \text{Compute}(t)$

act3: $b := t$

end

5.2 Introducing Idle Ports and Atomicity Breaking through Silent Steps

In this refinement, we introduce *Idle* ports: a port is *Idle* at time t if it has the same value as at time $t - 1$ (see the definition of the *Compute* function in Section 4.1). Intuitively, a port is idle when the thread to which it belongs is not active, i.e., after the deadline. Moreover, non idle ports are now not updated atomically: we introduce a silent *Step* event for updating them incrementally through the variable *compute*.

Basic sets. We introduce the constant time parameterized function *Idle*:

CONTEXT I_Ports

EXTENDS Ports

CONSTANTS Idle

AXIOMS

axm1: $\text{Idle} \in \mathbb{N} \rightarrow \mathbb{P}(\text{Port})$

axm3: $\forall t. (t \in \mathbb{N} \Rightarrow (\forall p. p \in \text{Port} \Rightarrow (p \in \text{Idle}(t+1) \Rightarrow \text{Compute}(t+1)(p) = \text{Compute}(t)(p))))$

END

The static description. We introduce the variable *compute* to incrementally record port updates: recorded ports define the domain of the *compute* function (array). The invariant inv2 states the correctness of this recording; any recorded slice is equal to the range of the *Compute* function over the same slice. The invariant inv4 states that idle ports are implicitly recorded.

VARIABLES t ports compute b

INVARIANTS

inv1: $\text{compute} \in \text{Port} \leftrightarrow \text{Val}$

inv2: $\forall d. (d \subseteq \text{dom}(\text{compute}) \Rightarrow d \triangleleft \text{compute} = d \triangleleft \text{Compute}(t))$

inv4: $\text{Idle}(t) \subseteq \text{dom}(\text{compute})$

The dynamic description.

Initialisation

begin

act3: $t, compute : | t' \in \mathbb{N} \wedge 0 < t'$
 $\wedge (t' \neq 1 \Rightarrow Idle[1 .. (t' - 1)] = \{Port\})$
 $\wedge compute' = Idle(t') \triangleleft Compute(0)$

act1: $ports := Compute(0)$

act4: $b := 0$

end

A silent step can occur if there exists some port not yet recorded:

Event *Step* $\hat{=}$

any p

where

grd1: $p \in Port$

grd2: $p \notin dom(compute)$

then

act1: $compute(p) := Compute(t)(p)$

end

A tick can occur if all the ports have been recorded:

Event *Tick* $\hat{=}$ **refines** *Tick*

when **grd1**: $dom(compute) = Port$

then

act1: $t, compute : | t' \in \mathbb{N} \wedge t < t'$
 $\wedge (t' \neq t + 1 \Rightarrow Idle[(t + 1) .. (t' - 1)] = \{Port\})$
 $\wedge compute' = Idle(t') \triangleleft compute$

act2: $ports := compute$

act3: $b := t$

end

5.3 Partitioning the Ports

In this refinement, we partition ports into delayed and immediate ports. We also introduce the computation pattern for immediate ports. At time t , the computation function for an immediate port takes into account the values of other ports at the same time t . It follows that the value of such predecessor ports should have been computed before and more generally that the predecessor relation should be acyclic.

Basic sets.

CONTEXT P_Ports

EXTENDS I_Ports

CONSTANTS Delayed Immediate

AXIOMS

axm1: $Delayed \subseteq Port$

axm2: $Immediate \subseteq Port$

axm3: $partition(Port, Delayed, Immediate)$

END

The static description.

VARIABLES t ports compute b

The dynamic description.

Event *ComputeImmediate* $\hat{=}$ **refines** *Step*
any p
where
 grd1: $p \in \text{Immediate}$
 grd2: $p \notin \text{dom}(\text{compute})$
 grd3: $\text{Pred}_{I^{-1}}[\{p\}] \subseteq \text{dom}(\text{compute})$
then
 act1: $\text{compute}(p) := C_I(p)(\text{Pred}_{I^{-1}}[\{p\}] \triangleleft \text{compute})$
end

5.4 Introducing Port Buffering

In this refinement, we make precise the computation pattern for delayed ports. At time t , the computation function for a delayed port takes into account the values of other ports at their last deadline. In order to give access to such *past* values, we use a buffering mechanism. The boundedness of such a buffering is ensured thanks to the properties of the Deadline function (see properties (II) of Section 6.2).

Event *ComputeDelayed* $\hat{=}$ **refines** *ComputeDelayed*
any p **where**
 grd1: $p \in \text{Delayed}$
 grd2: $p \notin \text{dom}(\text{compute})$
then
 act1: $\text{compute}(p) := C_D(p)(\text{Pred}_{D^{-1}}[\{p\}] \triangleleft \text{mem})$
end

5.5 Port Update Scheduling

This is our last refinement step. As already discussed in Section 4, the events of this refinement are deterministic and the choice between them is exclusive. Although, unlike classic B, an implementation refinement is not supported by Event B, we believe that this refinement is significant with respect to a true implementation of an AADL data port scheduler. In fact, the only data structure that remains as non implementable with respect to classic B, is the compute partial function. The implementation of partial functions can be considered now as part of the folklore and could be done by automatic refinements as proposed by [18].

6 Development Validation

In this section, we relate some facts about the proposed development. The first one concerns the development proofs and the second one concerns a technical aspect about the resources needed to handle recursive calls.

6.1 Proof Obligations

Most of the development has been done with the Rodin platform. There remain, however, some proofs that cannot be done with Rodin mainly related to the last refinement. This refinement relies on lists (B sequences) which are not yet supported by Rodin. It was thus easier for us to translate (manually) the development to Isabelle [17] and carry out all of the proofs within its proof environment. In fact, thanks to the locale mechanism of Isabelle it is easy to simulate Event B context extensions and machine refinements. However, since Isabelle is a general purpose theorem prover and not a method dedicated prover like Rodin, proof obligations related to invariant preservation, refinement and event feasibility had to be generated by hand. Fortunately, Isabelle decision procedures are very powerful and most of the proofs were straightforward.

6.2 Recursive Function Patterns

In this section, we present the basic ideas underlying the proposed implementation of recursive calls with bounded memory. In our representation of the AADL data port protocol, we have essentially two patterns:

- **well-founded recursion:** This pattern was used for the computation of immediate ports (see Section 4.1). Let us recall that the values of these ports depend on other immediate port values.

$$\text{Compute}(t)(p) = \text{C_I}(p)(\{q \mapsto \text{Compute}(t)(q) \mid q \mapsto p \in \text{Pred_I}\})$$

Such a computation is possible because we assume that that Pred_I is an acyclic relation. Thus, the computation proceeds according to a total order compatible with that acyclic relation. It follows that when an element is processed, all the lower elements have been processed already. Thus, the computation uses a finite number of finite resources: the number of port buffers. We note that each element is processed once. Such a property is not provided by a basic implementation of recursivity. Memoization could have been used; but, since all the elements are, a priori, known and have to be processed, the proposed order-based evaluation strategy is more efficient since it avoids testing if an element has already been processed.

- **past recursion:** This pattern was used for the computation of delayed ports:

$$\text{Compute}(t)(p) = \text{C_D}(p)(\{q \mapsto \text{Compute}(\text{Deadline}(t \mapsto q))(q) \mid q \mapsto p \in \text{Pred_D}\})$$

where Deadline has the following properties:

$$\text{Deadline}(0) = 0 \wedge \text{Deadline}(t + 1) \neq \text{Deadline}(t) \Rightarrow \text{Deadline}(t + 1) = t \quad (1)$$

In fact, thanks to these properties of the Deadline function, such a pattern can be implemented through the following primitive recursive pattern:

$$f(0) = c \wedge f(n + 1) = g(f(n), n + 1)$$

Actually, for such a pattern, the value of $f(n)$ can be computed with one *register* and one *counter*: initializing the register with c and the counter with 0, we compute

the successive values of $f(i)$ until the counter values reaches n . Correctness is ensured by the invariant $counter \leq n \wedge register = f(counter)$ and termination by the variant $n - counter$.

The underlying idea of the preceding proposed implementation (Section 5.4) can be summarized as follows: in order to compute $f(Deadline(t))$ without recursion, we define an auxiliary primitive recursive function a such that:

$$a(0) = Deadline(0) \wedge a(n+1) = \mathbf{if} \text{Deadline}(n+1) = \text{Deadline}(n) \mathbf{then} a(n) \mathbf{else} f(n)$$

We show by induction on n that $\forall n. a(n) = f(Deadline(n))$. Then, since a is primitive recursive, $f(Deadline(n))$ can be computed in an iterative way with finite memory resources. It follows that the computation of $f_i(n)$ which requires the knowledge of $f_{k \in I}(Deadline(n))$ also requires finite memory resources since the set I is a priori known.

Remarks.

- We have given here one *underlying* idea of the proposed implementation. It can be reused as a pattern for implementing a recursive function with an unknown recursion depth with bounded memory resources. In a similar way, the other idea concerns the implementation of well-founded recursion.
- The proposed implementation (Section 5.4), does not recompute the result of iterations from one call to another.

7 Related Work

It is becoming acknowledged that one way to make things abstract is to consider them at a level where we have a coarse grain of atomicity. Implementation details are then introduced progressively while maintaining the properties of the coarse grain events. For instance, along these ideas, bus protocols have been developed starting from a synchronous view [12]. In these protocols, the concern is to ensure the correct behavior of the devices with respect the bus lines while establishing basic properties like *mutual exclusion* between the connected devices. We have been concerned by another safety property: the preservation of a *precedence relation* given by a functional specification.

With respect to the specific domain we have been concerned with: computation scheduling, we can cite the work of Stoddard et al. [20] about interrupt scheduling. We note that they are especially concerned by interrupt handling and not by communication aspects. Their work is also concerned by making proofs for an unknown number of tasks. Scheduling aspects have also been dealt with in [13]. Here, the main concern was to provide a constructive specification of the problem such that certified code could be extracted by the Coq system [5].

The work of [14] has also a semantic concern with respect to AADL. Its aim is to provide a synchronous execution platform for AADL. A Lustre [8] translation semantics of the basic mechanisms is proposed. Thanks to this approach, a model of the whole system is obtained. This model is executable and its properties can be expressed by means of synchronous observers; also, it can be validated or simulated thanks to the

specification of the environment and the automatic generation of input sequences. As we have said, this is a translation semantics approach, whereas our work is concerned by the validation of an operational semantics with respect to a denotational semantics.

8 Conclusions

In this paper, we have been concerned by the formalization of an existing protocol offered by the AADL architecture description language. Although the protocol description was precise, we believe that the proposed abstraction through a functional specification is interesting since it is compact and allows to reason about the protocol without going to the intricacies of the implementation. Moreover, although, an implementation can be obtained directly from such a functional description, our proposed implementation relies on, a priori known, finite resources, as is mandatory for an embedded environment. Technically, we have shown that, in some cases, the memory resources needed to handle recursive calls can be, a priori, bounded even if the recursion depth depends on the parameters. It would be worth studying how to facilitate the reuse of such techniques through specification patterns as proposed by [4] and [7].

Concerning future work, we envision to introduce quantitative timing aspects. In this paper, we have made the assumption that computations take *zero* time, or more concretely, that all the required computations take place between two ticks and respect the real time specification deadlines. More generally, we are interested in providing patterns for implementing abstract functional synchronous languages [6] or subsets of AADL [11] on top of concrete asynchronous architectures.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Cansell, D., Méry, D.: Refinement and reachability in Event_B. In: Treharne, H., King, S., Henson, M.C., Schneider, S.A. (eds.) ZB 2005. LNCS, vol. 3455, pp. 222–241. Springer, Heidelberg (2005)
3. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology (July 1997)
4. Ball, E., Butler, M.: Event-B patterns for specifying fault-tolerance in multi-agent interaction. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 104–129. Springer, Heidelberg (2009)
5. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Giménez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin, C., Saïbi, A., Werner, B.: The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA (August 1997), <http://coq.inria.fr>
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
7. Blazy, S., Gervais, F., Laleau, R.: Reuse of specification patterns with the B method. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 40–57. Springer, Heidelberg (2003)

8. Caspi, P., Halbwachs, N., Pilaud, P.: Lustre: a declarative language for programming synchronous systems. In: Proceedings of the 14th annual symposium on principles of programming languages, January 1987, pp. 178–188 (1987)
9. Dijkstra, E.: *A Discipline of Programming*. Prentice Hall, Englewood Cliffs (1976)
10. Faugère, M., Bourbeau, T., de Simone, R., Gérard, S.: MARTE: Also an UML profile for modeling AADL applications. In: ICECCS, pp. 359–364. IEEE Computer Society, Los Alamitos (2007)
11. Feiler, P.H., Lewis, B., Vestal, S.: The SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In: RTAS Workshop 2003, May 2003, pp. 1–10 (2003)
12. Franca, R.B., Buss Becker, L., Bodeveix, J.-P., Farines, J.-M., Filali, M.: Towards safe design of synchronous bus protocols in Event_B. In: Brazilian Symposium on Formal Methods, Gramado Brazil. LNCS, vol. 5902. Springer, Heidelberg (2009)
13. Izerrouken, N., Pantel, M., Thirioux, X.: Machine checked sequencer for critical embedded code generator. In: Cavalcanti, A. (ed.) ICFEM 2009. LNCS, vol. 5885, pp. 521–540. Springer, Heidelberg (2009)
14. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proceedings of the 7th ACM & IEEE international conference on Embedded software EMSOFT 2007, Salzburg, Austria, pp. 134–143. ASSERT (2007)
15. Manna, Z., Pnueli, A.: *The temporal logic of reactive and concurrent systems: specification*. Springer, Heidelberg (1991)
16. MetaH (1997), <http://www.htc.honeywell.com/metah/>
17. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Requet, A.: Bart: A tool for automatic refinement. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 345–345. Springer, Heidelberg (2008)
19. SAE. Aerospace information report. avionics architecture description language. Technical Report AS5506, SAE (March 2002)
20. Stoddart, B., Cansell, D., Zeyda, F.: Modelling and proof analysis of interrupt driven scheduling. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 155–170. Springer, Heidelberg (2006)

Matelas: A Predicate Calculus Common Formal Definition for Social Networking

Nestor Catano¹ and Camilo Rueda²

¹ Madeira ITI
Funchal, Portugal
ncatano@uma.pt

² Pontificia Universidad Javeriana
Cali, Colombia
crueda@cic.puj.edu.co

Abstract. This paper presents *Matelas*, a B predicate calculus definition for social networking, modelling social-network content, privacy policies, social-networks friendship relations, and how these relations effect users' policies. The work presented in this paper is part of an ongoing work that aims at using several formal methods tools and techniques to develop a full-fledged social-network service implementing stipulated policies. Although we employed Atelier B to write *Matelas*, plans are to port it to Event B and to use Rodin to implement the social-network application.

1 Introduction

Over the past years we have experienced a huge development in Internet and communication systems. Internet and technology have changed our lives. They have changed the way we perceive the world, the way we build social relations, the way we approach people, the way we are. Today, many people find easier to share interests with people on the opposite side of the world, people who they have never personally met, than with the neighbour from the opposite house. Social-networks services in the form of web-sites, e.g., Facebook, Sapo, MySpace, LinkedIn, Hi5, have revolutionised the way people socialise. They have become popular tools to allow people to share common interests, and keep-up with friends, family and business connections. Facebook, currently the dominant service, reports 250 million active user accounts, roughly half of which include daily activity [14]. A typical social network user profile features personal information (e.g., gender, birthday, family situation), a continuous stream of activity logged from actions taken on the site (such as messages sent, status updated, games played) and media content (e.g., personal photos and videos). The privacy and security of this information is therefore a significant concern [16]. For example, users may upload media (such as photographs) they wish to share with specific friends, but do not wish to be widely distributed to their network as a whole. However, social network services have conflicting goals. Although respecting the privacy of their client base is important, they must also grow and expand the connections between their users in order to be successful. This is

typically achieved by exposing content to users through links such as friends-of-friends, in which content relating to individuals known to a user’s friends (but not the user) is revealed. Examples of this behaviour include gaining access to a photo album of an unknown user simply because a friend is tagged in one of the images. Back-doors also exist to facilitate casual connections such as allowing an unknown user to gain access to profile information simply by replying to a message he or she has sent.

We argue that mechanisms for users to enforce restricted access to content in social network applications are urgently needed, and propose the use of formal method techniques to build a core social network application enforcing these policies. Formal methods are based on mathematical formalisms whereby social-networks policies can be expressed in logic unambiguously. Formal methods make possible the use of mathematically-based machinery to support the precise reasoning about the logical description of properties. The work presented in this paper is part of an ongoing research work [12] in which social networking web-sites (e.g. Facebook, Twitter, and Hi5) are used as a living testbed in which formal methods [23] coupled with graph theory and Human Computer Interaction (HCI) techniques are employed to develop more dependable, secure, and crucially trustworthy social network systems. In this paper, we present *Matelas*¹, a predicate calculus abstract specification layer definition for social networking, modelling social-network content, privacy policies, social-network friendship relations, and how these effect the policies with regards to content and other users in the network. Our work builds on Jean-Raymond Abrial’s “parachute strategy” of building systems [1] in which a system is first considered from a very abstract and simple point of view, with broad fundamental observations, and then details are added to describe more precise behaviour of the system. As future work, we envision to refine *Matelas* into a social-network core application that adheres to stipulated policies and definitions. Hence, from our predicate calculus model definition of social networks, a code-level model will be attained while applying successive refinement steps.

In the following, Section 2.1 presents the context of the work presented in this paper. Section 2.2 gives a brief introduction to the B method for software development. Section 3 presents *Matelas*. Section 4 discusses related work on the use of formal methods for social-networking, and Section 5 presents conclusions and discusses future work and underlying challenges.

2 Preliminaries

2.1 A Formal Framework for Social Networking

The work presented in this paper is part of an ongoing research work in which social networking web-sites are used as a living testbed in which formal methods coupled with graph theory and Human Computer Interaction (HCI) techniques are employed to develop more dependable, secure, and crucially trustworthy

¹ *Matelas* is the French word for the English word mattress.

social network systems. This ongoing work builds on the correct definition of *Matelas*. We plan to *refine* *Matelas* to a social network core application that adheres to stipulated policies [17,18]. The refined core application will serve as a common trunk to which social network features will be plugged-in. While this core is minimal in functionality, it will be considerably extended by incorporating plug-ins. This will be achieved by developing a framework where the plug-ins, written in popular programming languages such as Java or C, can demonstrate their adherence to the policies stipulated by *Matelas*. This will be achieved by using Proof Carrying Code (PCC) [20]. PCC is a technique in which a code consumer (the social network core application) establishes a set of rules (privacy and security policies) that guarantee that externally produced programs (the plug-ins) can safely be run by the consumer. In addition to the code to be executed, the code producer must provide a proof of adherence to the set of rules defined by the code consumer. This proof is run by the code consumer once, using a proof validator, to check whether the proof is valid and therefore the external program is safe to execute. Hence, the problem of extending our social network core application with plug-ins can be regarded as a producer-consumer problem in which the code producer (the plug-in) must adhere to security and privacy policies specified by *Matelas*, and as a consequence to the policies of the social network core application.

Furthermore, while a social network application or plug-in may adhere to stipulated policies, these policies might be insufficient to avoid human error. While a plug-in may not access users date of birth without explicit authorisation, it is still possible for users to inadvertently give such authorisation. This may happen either by accident or, most likely, due to the complexity of the settings and preferences interface that the user is asked to interact with. Hence, as part of our whole work on social networking, we will augment our correct social network core and plug-ins with understandable human interfaces that enable end users to express their privacy policies and preferences, as well as to review and modify them.

2.2 The B Method for Software Development

In the *refinement* calculus strategy for software development, the process of going from a system specification to its implementation in a machine goes through a series of stages. Each stage adds more details to the description of a system. Each stage can thus be seen as a model of the system at a particular level of abstraction. Models at each level serve different purposes. At higher levels models are used to state and verify key system properties. At lower levels models are used to implement the system behaviour. It is crucial that models at each stage are coherent with the system specification, *i.e.*, that the simulation obeys the specification properties. A model M_{i+1} at stage $i + 1$ is said to be a refinement of a model M_i at stage i when the states computed by M_i and M_{i+1} at each given step obey a so-called “gluing invariant” stating properties for the joint behaviour of both models. A refinement step generates *proof obligations* that must be formally verified in order to assert that a model M_{i+1} is indeed a

refinement of a model M_i . These are sufficient conditions to guarantee that, although at different levels of abstraction, both are models of the same system. Correctness of the whole development process is thus ensured (3).

In the B method ([1], [25]) models are so-called *machines* composed of a static part defining observations (variables, constants, parameters, etc) of the system and their invariant properties, and a dynamic part defining operations changing the state of the system. Each operation must maintain the invariant property. In B, the language for stating properties, essentially predicate logic plus set theory, and the language for specifying dynamic behaviour (*i.e.* programs) are seamlessly integrated. A significant feature of the B system modelling approach is the availability of automatic verification tools such as B-Tools [10], or Atelier B [5], and model-checking simulators such as ProB [22].

A derivative of the B method is Event B [3]. Event B models are developments of discrete transition systems. They are composed of *machines* and *contexts*. These correspond, roughly, to a B method *machine* whose static part (except variables and their invariants) is transferred to a different module (the context). B method operations are replaced in Event B machines by *events*. In B method machines, operations are *invoked*, either by a user or by another machine, whereas in Event B, an event can be fired some condition (its *guard*) holds. Three basic relations are used to structure a model. A machine *sees* a context and can *refine* another machine. A context can *extend* another context. Events have two forms, as shown in Table 1. The “when” form of event executes the action A_1 when the current value of the system variables v satisfies the guard G_1 . The “any” form of event executes action A_2 when there exists some value of x satisfying the guard G_2 . Proof obligations require invariants to hold after executing the actions.

Table 1. Events

any x where $G_2(x, v)$ then $A_2(x, v)$ end	when $G_1(v)$ then $A_1(v)$ end
---	--

3 Matelas

Matelas is a B abstract specification for social networking that models social-network content, social networks friendship relations, and privacy on content. Privacy issues have generated a bunch of theories, and approaches [26]. Nonetheless, as stated by Anita L. Allen in [4], “while a no universally accepted definition of privacy exists, definitions in which the concept of access plays a central role have become increasingly commonplace”. Following Allen’s approach, we model

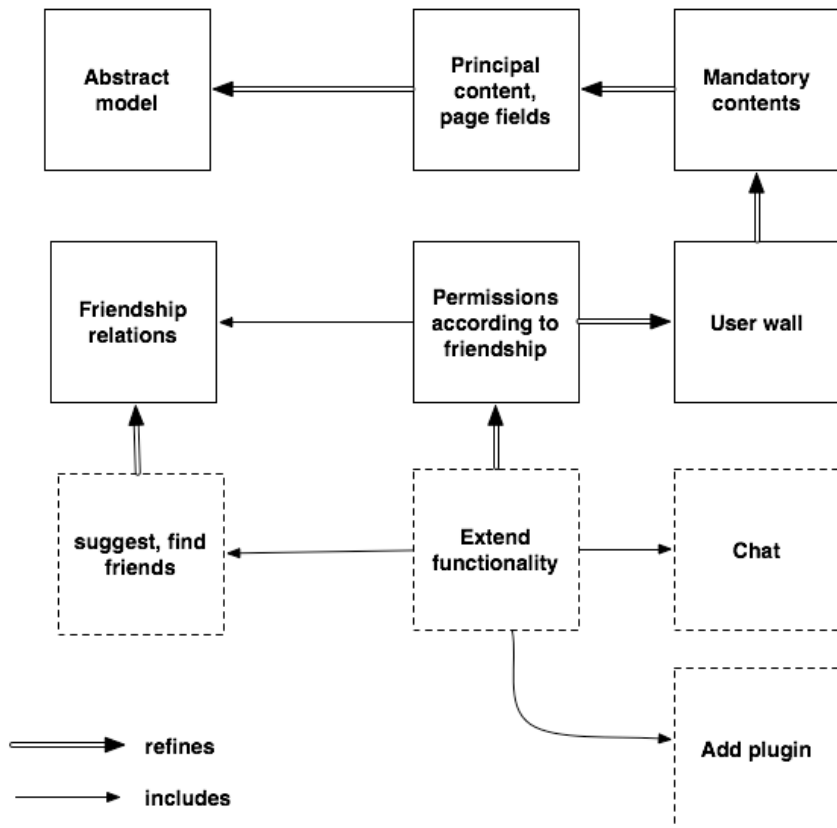


Fig. 1. System architecture. Dashed boxes are components not yet defined

privacy with the aid of a relation that registers users’ access privileges on social-network resources, and a content ownership relation.

Matelas distinguishes five rather independent aspects of social networks, namely, user content and privacy issues, friendship relation in social-networks, user content and how it is affected by friendship relations, external plug-ins, and the user interface. At the present stage our model comprises six B (implemented) components: an abstract machine, four refinements and an included machine. We plan to refine Matelas to a social-networking core system. What each implemented machine observes of the system is shown in Table 2. The architecture of the core system is shown in Figure 1, with dashed boxes representing components not yet defined. The fourth refinement in Table 2 includes the *social_friends* machine. A first abstract model views the system as composed of users and “raw content”, representing photos, videos, or text that a person has in his personal page. Four relations concerning raw contents are modelled at this level: content, visibility, ownership, and access privileges. The “content” relation associates a person with all raw contents currently in the person’s page.

Table 2. System architecture

Machine	Observations
Abstraction	Page content, content visibility, content ownership, access privileges
Refinement 1	Principal content, page fields
Refinement 2	Mandatory content
Refinement 3	User wall, wall visible content, wall access privileges
Social_friends	Friendship relations
Refinement 4	Relations between friendship, visibility and privileges

Each user owns some of the content in his page. The “visible” relation associates a person with visible raw content. Visible raw contents are those raw contents a user is allowed to view at some point. Only those raw contents for which a user has “view” privilege can be visible. The “content” relation contains the “visible” relation. The “view” privilege and other types of privileges (e.g., edition of a particular content) are defined in the access privileges relation *act*. Elements in *act* are triplets (rc, op, pe) stating that person *pe* has *op* privilege on raw content *rc*. In B language notation a triplet (a, b, c) is written $a \mapsto b \mapsto c$.

The owner $owner(rc)$ of a raw content *rc* is unique. The following invariant properties of the abstract model state that, (1) $owner(rc)$ has all privileges over *rc*², (2) each raw content owned by a user is in the user’s page content, (3) a raw content is visible for a user only when the user has “view” privilege over it, and (4) all user’s visible raw contents are in the user’s page.

- (1) $\forall rc.(rc \in rawcontent \Rightarrow (\forall op.op \in OPS \Rightarrow (rc \mapsto op \mapsto owner(rc)) \in act))$
- (2) $owner^{-1} \subseteq content$
- (3) $\forall (rc, pe).(rc \in rawcontent \wedge pe : person \Rightarrow ((pe \mapsto rc) \in visible \Rightarrow (rc \mapsto view \mapsto pe) \in act))$
- (4) $visible \subseteq content$

The abstract model defines actions (so-called “operations”) for creating, transmitting, making visible, hiding, editing, commenting and removing a raw content. All these, of course, are defined so as to maintain all the invariant properties. Code for the operation representing a user removing from his page a raw content owned by some other user is shown in Table 3. The pre-conditions requires the user in question not being the owner. Upper case items refer to types. Lower case, to variables of the system. A user can only remove visible raw contents. The SELECT clause has two cases. The first one is when the *rc* to be eliminated is not the only one present in *pe*’s page. The second one is the opposite. Since the web page of each person in the system must have at least one content, *pe* must be deleted from the system in this case. In B notation, $C \triangleleft r$ and $r \triangleright C$ denote restriction of a relation *r* to a subset *C* of its domain and its range respectively. Similarly, $C \triangleleft r$ and $r \triangleright C$ denote the restriction of the domain and the range of *r* to elements *not* belonging to *C*.

² *OPS* is the set of privilege types in the system.

Table 3. Operation for removing a raw content

```

remove_rc ( rc , pe ) =
PRE
  rc ∈ RAWCONTENT ∧ rc ∈ rawcontent ∧
  pe ∈ person ∧
  pe ↦ rc ∈ visible ∧ pe ≠ owner(rc)
THEN
  SELECT pe ∈ dom(content - {pe ↦ rc}) THEN
    visible := visible - {pe ↦ rc} ||
    content := content - {pe ↦ rc} ||
    act := act - {rc ↦ view ↦ pe} ||
  WHEN pe ∉ dom(content - {pe ↦ rc}) THEN
    visible := {pe} ⋈ visible ||
    content := {pe} ⋈ content ||
    act := act ▷ {pe} ||
    person := person - {pe}
  END
END

```

Table 4. Operation for removing an owned raw content

```

remove_owned_rc ( rc ) =
PRE
  rc ∈ RAWCONTENT ∧ rc ∈ rawcontent
THEN
  visible := visible ▷ {rc} ||
  content := content ▷ {rc} ||
  act := ({rc} × OPS) ⋈ act ▷ dom(content ▷ {rc}) ||
  owner := {rc} ⋈ owner ||
  person := dom(content ▷ {rc})
  END
END;

```

The operation for a user removing an owned raw content is shown in Table 4. Notice that in this case the removed content must also be removed from all other user's pages ($content ▷ \{rc\}$). This might leave some users with no raw contents in their pages. The persons remaining in the system must thus be recomputed ($person := dom(content ▷ \{rc\})$).

The first refinement mainly adds the observation of page fields. Each content belongs to some field. The notion of field models the fact that users perform different actions, such as commenting, dealing with some given content. Page fields are defined as $field ∈ (rawcontent - principal) → principal$. The various raw contents of a given field are thus thought to be related (e.g. as a comment, or as being part of a photo album) to a unique *principal* raw content. Removing a principal raw content entails removing all its “comment” contents in all user

pages. A principal raw content can only be removed by its owner. The following selected actions from the *remove_owned_rc* operation show this behaviour (for the case $rc \in principal$). Expression $field^{-1}[\{rc\}]$ gives all secondary raw contents whose primary is rc . These have to be removed together with rc .

$$\begin{aligned} rawcontent &:= rawcontent - (field^{-1}[\{rc\}] \cup \{rc\}) \parallel \\ content &:= content \triangleright (field^{-1}[\{rc\}] \cup \{rc\}) \parallel \\ act &:= (\{rc\} \times OPS) \triangleleft act \triangleright dom(content \triangleright \{rc\}) \end{aligned}$$

The second refinement models the fact that each user page must always keep some predefined minimum information. This is represented as a set of special predefined contents (referred to as *prawcontent*) in each page that cannot be removed. This predefined information must be present in a page before any other content is added, as stated in the invariant property, $prawcontent \subseteq rawcontent \Rightarrow prawcontent \neq \emptyset$. Remove operations are refined to ensure that all these special raw contents are always kept in a user's page.

The third refinement models the notion of *wall*, common in social network systems. A wall is modelled as a relation associating a user with some raw contents *different* from those in her web page: $wall \in person \leftrightarrow rawwall \wedge (rawwall \cap rawcontent = \emptyset)$. Each wall owner gives others some particular visibility and access privileges to his wall. Operations for adding/removing/hiding comments to/from the wall are included at this level.

Machine *Social_friends* provides definitions for types of friendship relations in social networks. The machine models *acquaintance*, *social* and *best friend* relations, with operations to add/remove users to/from each type of friendship relation of a given user. This machine is parametrised with a set modelling a type (that of "friends"). Some invariant properties of this machine are shown below, where $id(friend)$ is the identity relation over *friend*, and $ran(friendship)$ is the range of the *friendship* relation. The third property states that a user is not a friend of himself. The fourth one states that all friends are involved in some friendship relation. Other friendship types are defined similarly. Notice that friendship relations are *not* defined to be transitive.

$$\begin{aligned} friend &\subseteq FRIEND \\ friendship &\in friend \leftrightarrow friend \\ id(friend) \cap friendship &= \emptyset \\ friend &= dom(friendship) \cup ran(friendship) \\ best_friends &\in friend \leftrightarrow friend \\ best_friends &\subseteq friendship \end{aligned}$$

The fourth refinement includes the *Social_friends* machine. It models how access privileges relate to friendship relations, namely, *best_friends*, *social_friends*, and *acquaintances*. The relation *best_friends* models the highest level of friendship of people in the social network, and *acquaintances* the lowest. In general, a

lower friendship level cannot have any access privilege a higher level does not also have, as stated in the following invariant properties:

$\begin{aligned} \forall pe. pe \in \text{dom}(\text{friendship}) &\Rightarrow \\ \forall bs. bs \in \text{best_friends}\{\{pe\}\} &\Rightarrow \\ (\text{owner}^{-1}\{\{pe\}\} \times OPS) \cap \text{act}^{-1}\{\text{social_friends}\{\{pe\}\}\} & \\ \subseteq & \\ (\text{owner}^{-1}\{\{pe\}\} \times OPS) \cap \text{act}^{-1}\{\{bs\}\} & \end{aligned}$
$\begin{aligned} \forall pe. pe \in \text{dom}(\text{friendship}) &\Rightarrow \\ \forall sf. sf \in \text{social_friends}\{\{pe\}\} &\Rightarrow \\ (\text{owner}^{-1}\{\{pe\}\} \times OPS) \cap \text{act}^{-1}\{\text{acquaintances}\{\{pe\}\}\} & \\ \subseteq & \\ (\text{owner}^{-1}\{\{pe\}\} \times OPS) \cap \text{act}^{-1}\{\{sf\}\} & \end{aligned}$

Similar properties are stated for wall access privileges. All these properties only relate to each user's raw contents. That is, for any rc of a given user pe (i.e. $\text{owner}^{-1}\{\{pe\}\}$), the privileges of her social friends with respect to rc cannot include something that any pe 's best friend does not also have. In this fourth refinement, the *remove_owned_rc* operation adds the action

$$\text{restrict_friends}(\text{dom}(\text{content} \triangleright \{rc\}))$$

where *restrict_friends* is an operation of the *Social_friends* machine restricting the friendship relation to the supplied set (see Table 5).

Table 5. Restricting friendship relations to a supplied set

<pre> restrict_friends(frs) = PRE frs ⊆ FRIEND THEN friendship := frs ◁ friendship ▷ frs best_friends := frs ◁ best_friends ▷ frs social_friends := frs ◁ social_friends ▷ frs acquaintances := frs ◁ acquaintances ▷ frs friend := friend ∩ frs END </pre>

The operations distinguish between commenting a particular raw content in some user's page or doing so in the wall. Commenting a wall is done as shown in table 6. Variable *wall* records all contents present in each person's wall. Variable *vinwall* \subseteq *wall* keeps track of visible wall contents for each person, *wallowner* the owner of each content in a wall and *wallaccess* defines, for each wall owner, the persons allowed to comment her wall. In the operation in table 6, when a comment is added to the wall of person *ow*, the added comment is put in the wall of each person having access to the wall of *ow* (expression $(\text{wallaccess}\{\{ow\}\} \times \{cmt\})$) and is also defined to be visible in those walls.

Table 6. Operation for commenting in a wall

```

comment_wall ( cmt, ow, pe ) =
PRE
  pe ∈ person ∧ ow ∈ person
  ∧ cmt ∈ RAWCONTENT ∧ cmt ∉ rawcontent
THEN
  SELECT ow ↦ pe ∈ wallaccess ∧ cmt ∉ rawcanvas
  THEN
    rawwall := rawwall ∪ {cmt} ||
    rawcanvas := rawcanvas ∪ {cmt} ||
    vinwall := vinwall ∪ (wallaccess[{ow}] × {cmt}) ||
    wall := wall ∪ (wallaccess[{ow}] × {cmt}) ||
    canvas := canvas ∪ (wallaccess[ow] × {cmt}) ||
    wallowner := wallowner ∪ {cmt ↦ ow}
  END
END

```

3.1 Publishing Content

A common operation to social-networking web-sites is publishing content to people in the network. Publishing a social-network content rc to a user pe can be regarded as a process of transmitting rc from the page of $owner(rc)$ to the page of pe . The abstract machine code for transmitting a raw content in a social-network is shown in Table 7. The pre-condition of $transmit_rc$ requires that pe is different than ow , and rc is not already in the page of pe . To transmit raw content rc , the triplet $rc \mapsto view \mapsto pe$ is added to act so as to grant the $view$ permission on raw content rc to user pe , and raw content rc is made visible to pe by adding $pe \mapsto rc$ to $visible$.

In a complementary direction, user pe can request permission to operate raw-content rc . The abstract machine code for requesting a particular permission on a raw content rc is shown in Table 8, where op is the permission being requested.

Table 7. Transmitting page content

```

transmit_rc ( rc , ow , pe ) =
PRE
  rc ∈ RAWCONTENT ∧ rc ∈ rawcontent ∧ ow ∈ person ∧
  pe ∈ person ∧ ow = owner(rc) ∧
  ow ≠ pe ∧ pe ↦ rc ∉ content ∧ rc ↦ view ↦ pe ∉ act
THEN
  visible := visible ∪ pe ↦ rc ||
  content := content ∪ pe ↦ rc ||
  act := act ∪ rc ↦ view ↦ pe
END

```

Operation *request_permission* can either grant *pe* permission *op* over raw content *rc*, or deny the permission. If the permission is granted, $rc \mapsto op \mapsto pe$ is added to *act*, *rc* is added to *content(pe)*, and the result variable *res* is set to *TRUE* so as to communicate the success of the operation. Otherwise, when the permission is denied, *res* is set to *FALSE*. The pre-condition of requesting a permission requires that *pe* is different than *owner(rc)*.

Table 8. Requesting Content Permissions

```

res ← request_permission (rc , pe , op) =
PRE
  rc ∈ RAWCONTENT ∧ rc ∈ rawcontent ∧ pe ∈ person ∧
  op ∈ OPS ∧ pe ≠ owner(rc)
THEN
  CHOICE
    act := act ∪ rc ↦ op ↦ pe ||
    content := content ∪ pe ↦ rc ||
    res := TRUE
  OR
    res := FALSE
  END
END

```

4 Related Work

P3P, the Platform for Privacy Preferences (<http://www.w3.org/P3P/>), an effort of the World Wide Web Consortium (W3C), encompasses a standard XML markup language for expressing privacy policies so as to enable user agent tools (e.g. Web browsers, electronic wallets, mobile phones, stand-alone applications, or social network applications) to read them and take appropriate actions. A P3P Policy is primary a set of boolean answers to multiple-choice questions about name and contact information, the kind of access that is provided, the kind of data collected, the way the collected data will be used, and whether the data will be shared with third parties or not. Though P3P policies are precisely scoped [13], they are not expressive enough to model general privacy properties on content. They are not based on mathematical formalisms either, e.g., predicate calculus, so that it is not possible to reason about the truths derivable from policies expressed in P3P standard language.

In [7], N. Sadeh et al. develop a theory that relates expressiveness and efficiency in a domain-independent manner. Authors derive an upper bound on the expected efficiency of a given mechanism. The expected efficiency depends on the mechanism's expressiveness only. Using predicate calculus to write users' privacy policies on content improves the expressiveness of mechanisms modelling policies. We plan to build on Sadeh et al.'s work to study how this higher expressiveness of predicate calculus based privacy policies comes down to a higher

efficiency of the agent mechanisms allowing social-network users to set their privacy preferences.

In B language the expression of temporal logic constraints is notably missing. In [15], J. Gros Lambert proposes a method to verify temporal logical properties of Event B systems. We will build on Gros Lambert’s work, and J-R Abrial’s work in [2], to verify temporal logic properties about *Matelas*.

In [19], Vijay Saraswat et al. propose a policy language for access control, and a policy algebra in the timed constraint programming paradigm. Based on Saraswat’s work, we plan to extend our work on modelling privacy on content with a relation that registers users’ access privileges on social-network content with a relation that registers role-based access permissions on content.

5 Conclusion

We presented *Matelas*, a B model for social networking, describing social-network content, privacy policies, social-networks friendship relations, and how these effect the policies with regards to content and other users in the network. We used Atelier B [5] to write *Matelas*. We found the B method particularly useful in two aspects. One is the expressivity of the generalised substitution language that makes it possible to construct a very simple abstract model of the system, yet containing all fundamental security and privacy properties. The second is that proof obligations are easy to interpret as “before-after” predicates of each operation assignments which makes it easy to discover possible errors and their correction by just looking at the statement of the proofs. A minor drawback, at least for this application, is that some useful operations are discovered as the refinement process leads to more detailed components which requires to change all previous models to include these operations as empty statements. This inconvenience could, of course, be circumvented by using Event B. Our decision of using the Atelier B tool to undertake the development of the social-network core was based on the authors’ previous experience with the tool. We envision to port *Matelas* machines to Event B models and to use Rodin [24] to refine *Matelas* so as to produced the social-network core system.

We have a positive impression on the use of Atelier B as tool to develop relatively complex software systems, yet have some recommendations on how the tool can be improved. For *Matelas*’ abstraction, the four refinements and the *social_friends* machine, the B method software tool Atelier B generates 658 proof obligations. About 60% of them are discharged automatically. Most of the others (handling up to 90% of all obligations) are discharged in *Atelier B* by just doing *modus ponens* followed by invocation of one of the available provers. Some proofs, especially those involving equality of assignments in abstract and concrete machines, are somewhat tricky. We found this might be due to some limitations of the provers for handling predicates of the form $A \vee B$, for A, B complex expressions with A false and B true, even when B is supplied as a hypothesis and proved first or, similarly, $\neg A$ is added as hypothesis and proved first.

The work presented in this paper is part of an ongoing work that aims at developing a full-fledged social network core that implements stipulated privacy

policies. To the best of our knowledge, this is the first effort on using the B method to formally develop a social-network web-site. We plan to refine Matelas to a social-network core, and use Proof Carrying Code [20,21] to build Java plug-ins that extend its features. The policies for Java plug-ins can be written in JML, which allows the use of different formal methods tools to check program correctness [9,11]. JML specifications have the advantage over predicate calculus based models in that they are close to Java, and thus are closer to average programmers. We envisage to investigate on systematic ways B Machines can be translated into JML specifications. Work has already been done in the other direction [8], that is, to transform JML specifications into B machines to check the specifications for flaws. Alternatively, plug-ins can be written in C language, and formal specifications using the ACSL (ANSI/ISO C Specification Language) specification language [6], a JML-like specification language for C programs. Altogether, our work falls within the scope of the Grand Challenge in “Dependable System Evolution” (<http://vsr.sourceforge.net/introduction.htm>) set forth by the U.K, and Tony Hoare’s Grand Challenge in Verified Software. The challenge is to create a toolset that would guarantee that programs meet given specifications.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica* 77(1,2), 1–24 (2007)
4. Allen, A.L.: *Uneasy Access: Privacy for Women in a Free Society*. Rowman and Littlefield (1988)
5. Atelier b, http://www.atelierb.eu/index_en.html
6. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language, <http://frama-c.cea.fr/download/-plug-indevelopmentguide.pdf>
7. Benisch, M., Sadeh, N., Sandholm, T.: A theory of expressiveness in mechanisms. In: *Proceeding of the 23rd Conference on Artificial Intelligence (July 2008)*
8. Bouquet, F., Dadeau, F., Julien, J.: JML2B: Checking JML specifications with B machines. In: *The 7th International B Conference*, pp. 285–288 (2007)
9. Breunese, C., Catano, N., Huisman, M., Jacobs, B.: Formal methods for smart cards: An experience report. *Science of Computer Programming* 55(1-3), 53–80 (2005)
10. B Tools, <http://www.b-core.com/btool.html>
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)* 7(3), 212–232 (2005)
12. Catano, N., Kostakos, V., Oakley, I.: Poporo: A formal framework for social networking. In: *3rd International Workshop on Formal Methods for Interactive Systems (FMIS)*, Eindhoven, The Netherlands (November 2009) (to appear)

13. Cranor, L., Lessig, L.: *Web Privacy with P3p*. O'Reilly & Associates, Inc., Sebastopol (2002)
14. Facebook's statistics, <http://www.facebook.com/press/info.php?statistics>
15. Gros Lambert, J.: Verification of LTL on B event systems. In: Julliand, J., Kouchnarenko, O. (eds.) *B 2007*. LNCS, vol. 4355, pp. 109–124. Springer, Heidelberg (2006)
16. Gross, R., Acquisti, A.: Information revelation and privacy in online social networks. In: *Workshop on Privacy in the Electronic Society (WPES)*, pp. 71–80 (2005)
17. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) *ESOP 1986*. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
18. Hoare, C.A.R.: Proof of correctness of data representations. *Acta Informatica* 1, 271–281 (1972)
19. Jagadeesan, R., Marrero, W., Pitcher, C., Saraswat, V.A.: Timed constraint programming: a declarative approach to usage control. In: *Proceeding of Principles and Practice of Declarative Programming (PPDP)*, pp. 164–175 (2005)
20. Necula, G.C.: Proof-carrying code. In: *Symposium on Principles of Programming Languages (POPL)*, Paris, January 1997, p. 106119 (1997)
21. Necula, G., Lee, P.: Research on proof-carrying code for untrusted-code security. In: *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, p. 204 (1997)
22. ProB, <http://users.ecs.soton.ac.uk/mal/systems/prob.html>
23. Robinson, A., Voronkov, A.: *Handbook of Automated Reasoning*. MIT Press, Cambridge (2001)
24. Rodin, <http://www.event-b.org/platform.html>
25. Schneider, S.: *The B-Method: An Introduction*. Palgrave (2001)
26. Schoeman, F.D.: *Philosophical Dimensions of Privacy: An Anthology*. Cambridge University Press, Cambridge (1984)

Structured Event-B Models and Proofs

Stefan Hallerstede

University of Düsseldorf
Germany

`stefan.hallerstede@wanadoo.fr`

Abstract. Event-B does not provide specific support for the modelling of problems that require some structuring, such as, local variables or sequential ordering of events. All variables need to be declared globally and sequential ordering of events can only be achieved by abstract program counters. This has two unfortunate consequences: such models become less comprehensible — we have to infer sequential ordering from the use of program counters; proof obligation generation does not consider ordering — generating too many proof obligations (although these are usually trivially discharged).

In this article we propose a method for specifying structured models avoiding, in particular, the use of abstract program counters. It uses a notation that mainly serves to drive proof obligation generation. However, the notation also describes the structure of a model explicitly. A corresponding graphical notation is introduced that visualises the structure of a model.

1 Introduction

Recently, we have argued that the benefits of the minimalist approach of Event-B [1] to formal modelling are sometimes balanced by complications that result, in particular, from more complicated invariants [11]. The reason for this was that it was necessary to introduce abstract program counters when dealing with models that require (sequential) ordering of some events. However, we have argued in an earlier article [9] that, specifically, structuring constructs like sequential composition or if-statements lead to complications. Thus, the problem we face is keeping the simplicity resulting from the minimalism while providing some means to structure Event-B models. The solution we propose is to move more information about what is to be proved into the models — a solution we have already chosen before by introducing witnesses to Event-B. We do not introduce sequential composition or if-statements but a notation that allows us to state properties to prove about them. The usual approach in program verification would be to derive proof obligations from a program following its structure. We do not have a program but work exclusively with the proof obligations.

We need to specify control flow in Event-B models without having to resort to implementing abstract program counters. In principle proof outlines [16] can accomplish this. However, similarly to [14], we want to avoid introducing a concrete syntax of a programming notation.

Proof outlines are a compact representation of correctness proofs using Hoare triples [12]. A Hoare Logic states what is to be proved about a program S . For predicates p and q we write $\{p\} S \{q\}$ to state that “starting from a state satisfying p program S leads to a state satisfying q ”. Sequential composition of programs S and T is proved to satisfy $\{p\} S; T \{q\}$ by a rule

$$\frac{\{p\} S \{r\} \quad \{r\} T \{q\}}{\{p\} S; T \{q\}}$$

Proof outlines [16] represent this more succinctly,

$$\{p\} S; \{r\} T \{q\}$$

annotating the program $S; T$ with all predicates involved in the proof. This notation is used extensively in [4] to present correctness proofs of programs.

Similarly to proof outlines, temporal verification diagrams [14] specify alternating sequences of assignments and assertions. In addition, they provide hierarchical structuring based on state charts. Certain “patterns” of diagrams are identified that are instrumental in proofs of temporal properties of reactive systems. We combine ideas of [16] and [14] in our proposal for structured Event-B.

Overview. We introduce structured Event-B in Section 2. A small example in Section 3 describes the relationship between Event-B and structured Event-B. In Section 4 we develop a simple sequential program to show how the method could be used in practice. Section 5 points to related and future work and Section 6 contains a conclusion.

2 Event-B with Structure

We introduce a structuring notation for Event-B that maintains the simplicity of the original Event-B proof obligations. In this article, we identify two concepts that are missing from Event-B: sequentiality and locality. Event-B is strongest at proving properties of highly concurrent systems that mostly use global variables. Our structured notation supports sequentiality and locality while keeping the ease of use of Event-B. Concurrency can be modelled explicitly in the style of [16]. Here, we focus on sequentiality as this is at the moment difficult to model in Event-B.

2.1 Notation

Before introducing the notation in detail, we provide some small examples of terms of the notation together with a graphical notation that we use for illustration. (The graphical notation is not an exact representation. It’s main purpose is to clarify a model. See also [6].) The structure notation is based on assertions p , q and r , and events e and f . We write $p \rightarrow e \rightarrow q$ for “starting from assertion p event e establishes q ”, Figure 1a; we write $p \rightarrow e -$ for “assertion p is an

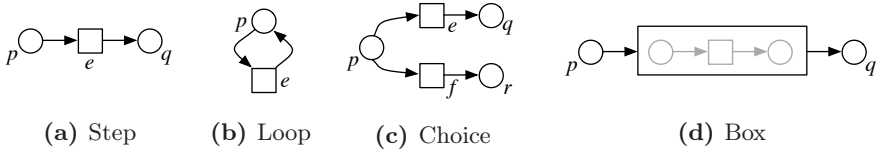


Fig. 1. Graphical representation of the structure notation

invariant of event e ”, Figure 1b, and $p \rightarrow e \leftarrow$ if e is convergent using the same graphical representation; we write $p \rightarrow (e \rightarrow q \parallel f \rightarrow r)$ for “starting from p event e establishes q or event f establishes r ”, Figure 1c, and we write $p \rightarrow [S] \rightarrow q$ for “starting from p box S establishes q ” where S is any term, Figure 1d. In Section 5 we describe briefly a construct for concurrency that we are considering. For the purpose of this article the notation outlined above is sufficient.

In order to define some operators on structure terms, we need to know about their possible shapes. The syntax of the (sequential) structure notation (p, q are predicates, and e is an event that may be decorated with a “!”, see below) is:

$$\begin{aligned}
 S & ::= p \rightarrow T \\
 T & ::= U \rightarrow S \mid U \rightarrow q \mid U - \mid e \leftarrow \mid T_1 \parallel T_2 \\
 U & ::= e \mid [S]
 \end{aligned}$$

We define two operators \mathcal{I} and \mathcal{F} on the syntax of the structure notation yielding the *initial assertion* and the *final assertions* of a term, respectively. The initial assertion is defined by $\mathcal{I}(p \rightarrow T) \hat{=} p$. The final assertion of a term is the disjunction of the “end points” of the term,

$$\begin{aligned}
 \mathcal{F}(p \rightarrow U \rightarrow S) & \hat{=} \mathcal{F}(S) & \mathcal{F}(p \rightarrow U -) & \hat{=} \text{false} \\
 \mathcal{F}(p \rightarrow U \rightarrow q) & \hat{=} q & \mathcal{F}(p \rightarrow e \leftarrow) & \hat{=} \text{false} \\
 \mathcal{F}(p \rightarrow (T_1 \parallel T_2)) & \hat{=} \mathcal{F}(p \rightarrow T_1) \vee \mathcal{F}(p \rightarrow T_2)
 \end{aligned}$$

The definition of the final assertion is consistent with the axiomatic semantics of non-deterministic choice [4]. This will become apparent in the description of assertions below. The intuition behind the definition of $\mathcal{F}(p \rightarrow U -)$ and $\mathcal{F}(p \rightarrow e \leftarrow)$ is that the event “returns to p ”; and in absence of an exiting choice, for example, $p \rightarrow (U_1 - \parallel U_2 \rightarrow q)$, it does not have an “end point”.

2.2 Proof Obligations

Proof obligations are defined following the structure notation. In fact, the main purpose of the structure notation is to drive the generation of proof obligations in a more evident manner.

Assertions. To state the proof obligations for assertions we consider all suitable sub-terms of a structure term. For instance, given the following structure term $p(v) \rightarrow e(v) \rightarrow q(v) \rightarrow f(v) \rightarrow r(v)$, we consider the sub-terms $p(v) \rightarrow e(v) \rightarrow q(v)$ and $q(v) \rightarrow f(v) \rightarrow r(v)$.

Let $e(v)$ be an event with guard $g(v)$ and action $v :| a(v, v')$. For the sub-term $p(v) \rightarrow e(v) \rightarrow q(v)$ we prove

$$p(v) \wedge g(v) \wedge a(v, v') \Rightarrow q(v') \qquad \text{assertion preservation}$$

We also prove *action feasibility*, $p(v) \wedge g(v) \Rightarrow \exists v' \cdot a(v, v')$, if an event is not refined further. Loops and choices are treated similarly to steps. Terms $p(v) \rightarrow e(v) -$ and $p(v) \rightarrow e(v) \leftarrow$ correspond to $p(v) \rightarrow e(v) \rightarrow p(v)$ but make explicit that $p(v)$ is invariant. The term $p(v) \rightarrow (e_1(v) \rightarrow q_1(v) \parallel e_2(v) \rightarrow q_2(v))$ corresponds to two terms $p(v) \rightarrow e_1(v) \rightarrow q_1(v)$ and $p(v) \rightarrow e_2(v) \rightarrow q_2(v)$.

For a box $p(v) \rightarrow [S(v)] \rightarrow q(v)$ we have to prove

$$\begin{array}{ll} p(v) \Rightarrow \mathcal{I}(S(v)) & \text{box entry} \\ \mathcal{F}(S(v)) \Rightarrow q(v) & \text{box exit} \end{array}$$

Convergence. The term $p(v) \rightarrow e(v) \leftarrow$ states that p is an invariant of $e(v)$ and that $e(v)$ is *convergent*, that is, it decreases a variant. If a variant $t(v)$ is specified for $e(v)$ or a refinement of $e(v)$, denoted by $e(v) \leftarrow t(v)$, we prove convergence of $e(v)$ in terms of *variant boundedness*, $p(v) \wedge g(v) \Rightarrow t(v) \geq 0$, and *variant progress*, $p(v) \wedge g(v) \wedge a(v, v') \Rightarrow t(v') < t(v)$. In unstructured Event-B events can be declared to be anticipated in order to delay a convergence proof to some refinement. In structured Event-B convergence is proved only when a variant is stated which may happen in a refinement. So the distinction between convergent and anticipated events disappears.

Refinement. We consider three forms of refinement, *structure* refinement, *event* refinement, and *box* refinement. A structure refinement replaces an event in a refined model by a structure. Event refinement relates two events, box refinement two boxes. Structure refinement is defined in terms of event and box refinement. An event $e(v)$, occurring in a term $p(v) \rightarrow e(v) \rightarrow q(v)$, is structure refined by a term $R(v, w)$, denoted by $e(v) \sim R(v, w)$, where the term $R(v, w)$ must contain at least one event decorated with an exclamation mark. The assertions $p(v)$ and $q(v)$ are associated with gluing assertions $p^*(v, w)$ and $q^*(v, w)$. We prove

$$\begin{array}{ll} p(v) \wedge p^*(v, w) \Rightarrow \mathcal{I}(R(v, w)) & \text{box entry} \\ \mathcal{F}(R(v, w)) \Rightarrow q^*(v, w) & \text{box exit} \end{array}$$

Note that $p^*(v, w)$ needs to be established by the event that refines the event preceding $e(v)$ in the abstract term. We do not allow strengthening of assertions in any other case.

Two kinds of events occur in $R(v, w)$, decorated events $f(w)!$ that refine event $e(v)$ and, undecorated, new events $f(w)$ that refine *skip*. Let $h(w)$ be the guard of $f(w)$ and $w :| b(w, w')$ its action. The predicate $m(v, w, v', w')$ denotes *witnesses* for the abstract variables v' linking abstract variables to concrete variables. Witnesses describe for each event separately how the refinement is achieved [9]. For decorated events $f(w)!$ occurring in a term $r(v, w) \rightarrow f(w)! \rightarrow s(v, w)$, let $\phi(v, w, w') = r(v, w) \wedge h(w) \wedge b(w, w')$; we prove

$\phi(v, w, w') \Rightarrow \exists v' \cdot m(v, w, v', w')$	<i>witness feasibility</i>
$\phi(v, w, w') \Rightarrow g(v)$	<i>guard strengthening</i>
$\phi(v, w, w') \wedge m(v, w, v', w') \Rightarrow a(v, v')$	<i>action simulation</i>
$\phi(v, w, w') \wedge m(v, w, v', w') \Rightarrow s(v', w')$	<i>assertion preservation</i>

For undecorated events $f(w)$ occurring in a term $r(v, w) \rightarrow f(w) \rightarrow s(v, w)$ we prove that they refine *skip*; we prove *assertion preservation* $\phi(v, w, w') \Rightarrow s(v, w')$.

Box refinement maintains the box-entry property once proved. A box $[S(v)]$, occurring in a term $p(v) \rightarrow [S(v)] \rightarrow q(v)$, is refined by a box $[R(v, w)]$ where $S(v)$ and $R(v, w)$ are identical terms except for assertions contained in $R(v, w)$ that may be strengthened. Box refinement is established by *box entry* and *box exit* proof obligations with respect to the gluing assertions $p^*(v, w)$ and $q^*(v, w)$.

Enabledness. Enabledness proof obligations in Event-B can be used to verify deadlock-freeness or precondition weakening [10], for instance. In structured Event-B the large disjunctions that would appear in Event-B enabledness proof obligations can be smaller depending on the structure term $R(v, w)$ of a structure refinement $e(v) \sim R(v, w)$. For terms $r(v, w) \rightarrow f(w) \rightarrow s(v, w)$, $r(v, w) \rightarrow f(w) -$, and $r(v, w) \rightarrow f(w) \leftarrow$ contained in $R(v, w)$ we prove $r(v, w) \Rightarrow h(w)$. For a choice term $r(v, w) \rightarrow (f_1(w) \rightarrow s_1(v, w)) \parallel f_2(w) \rightarrow s_2(v, w)$ we prove $r(v, w) \Rightarrow h_1(w) \vee h_2(w)$. If $r(v, w)$ is the initial assertion of $R(v, w)$, the abstract guard $g(v)$ is added to the premise.

3 Event-B With and without Structure

The Event-B method as described in [3] has a certain structure that is not made formally explicit. However, it is mentioned in the informal description in [3]. Not taking into account convergence, this would correspond to

$$true \rightarrow \mathbf{initialisation} \rightarrow inv \rightarrow (\mathbf{event}_1 - \parallel \dots \parallel \mathbf{event}_n -)$$

where *inv* is the invariant. The correspondence described in this section is not intended to suggest such a definition. It serves merely to explain the structured notation in terms of the unstructured notation.

3.1 Without Structure

We give a very simple example of a structured model and a corresponding unstructured model. Being very simple, too, proofs are omitted. We model an abstract program that sets y to 2. In order to represent structure in unstructured Event-B we have to introduce an abstract program counter *apc*, say, with values *aini*, *aend*, yielding a model with invariant

$$\begin{aligned} apc = aini &\Rightarrow y = 0 \\ apc = aend &\Rightarrow y = 2 \end{aligned}$$

and events

initialisation	convergent <i>inc2</i>
<i>apc</i> := <i>aini</i>	when <i>apc</i> = <i>aini</i> then
<i>y</i> := 0	<i>apc</i> := <i>aend</i>
	<i>y</i> := <i>y</i> + 2

We would show convergence of *inc2* to show that the abstract program counter is modelled correctly (using the variant $\{apc\} \cap \{aini\}$, for instance).

We refine the abstract model by one that increments a variable *x* in two steps. We use two events *incx1* and *incx2*,

initialisation	convergent <i>incx1</i>	convergent <i>incx2</i>
<i>cpc</i> := <i>aini</i>	when <i>cpc</i> = <i>aini</i> then	when <i>cpc</i> = <i>amid</i> then
<i>x</i> := 0	<i>cpc</i> := <i>amid</i>	<i>cpc</i> := <i>aend</i>
	<i>x</i> := <i>x</i> + 1	<i>x</i> := <i>x</i> + 1

and a gluing invariant

$$\begin{aligned} cpc \in \{aini, aend\} &\Rightarrow x = y \\ cpc \in \{aini, amid\} &\Rightarrow apc = aini \\ cpc = amid &\Rightarrow x = y + 1 \end{aligned}$$

that relates concrete variables *x* to abstract variables *y* depending on the value of the program counter. It is also necessary to relate the program counters *apc* and *cpc* by $cpc \in \{aini, amid\} \Rightarrow apc = aini$. Control flow is modelled explicitly.

3.2 With Structure

Using the structure notation, there is no need to model program counters. Assertions *aini* and *aend*

$$\begin{aligned} @aini \quad &y = 0 \\ @aend \quad &y = 2 \end{aligned}$$

(read: at *aini* “*y* = 0”) are stated at those locations where they hold

$$true \rightarrow iniy \rightarrow aini \rightarrow inc2 \rightarrow aend$$

The control flow is modelled by the structure term. It is not represented in the formal text otherwise. The model contains events *iniy* and *inc2*

<i>iniy</i>	<i>inc2</i>
<i>y</i> := 0	<i>y</i> := <i>y</i> + 2

Similarly to the unstructured model, we refine the abstract model by incrementing twice. Event *inc2* is structure refined by the *incx1* and *incx2*!

$$inc2 \sim aini \rightarrow incx1 \rightarrow amid \rightarrow incx2! \rightarrow aend$$

¹ We could also have used the same event twice. But in this article we want to keep the convention that each event appears only once. The structure refinement notation $e \sim R$ used in this article does not consider the position of *e* in the abstract term.

and the abstract event $iniy$ by the concrete event $inix$ which is stated formally $iniy \sim true \rightarrow inix! \rightarrow aini$.

$$\begin{array}{ccc} inix & incx1 & incx2 \\ x := 0 & x := x + 1 & x := x + 1 \end{array}$$

With the gluing assertions

$$\begin{array}{l} @aini \quad x = y \\ @amid \quad x = y + 1 \\ @aend \quad x = y \end{array}$$

we have to prove, for instance, that $incx1$ event refines $skip$ and $incx2$ event refines abstract event $inc2$

$$\begin{array}{l} x = 0 \wedge x = y \Rightarrow x + 1 = y + 1 \\ x = y + 1 \Rightarrow x + 1 = y + 2 \end{array}$$

These proof obligations correspond closely to the proof rule of the refinement calculus [15] for sequential composition.

3.3 Remarks

With the structure made explicit we can immediately see the sequencing in the model whereas in the unstructured model we have to look closely to see it. This becomes more convincing in larger examples like the one of Section 4, for example. In addition, the assertions of the structured model are simpler than the invariants of the unstructured model. In particular, it is not necessary to specify sequencing information relating only abstract program counters. The refinement proofs are not more difficult in the structured model although the formal definition is more complex mostly due to the box proof obligations. Note, however, that usually we do not have to prove anything at all for boxes. This is because we reuse assertions already declared, trivially satisfying the implications of box entry and box exit, for instance, $aini$ and $aend$ in the model above. We have removed one source of complexity: the choices to code the assertions in the invariant are no longer available. We believe this makes the method easier to use. The main drawback of the structured approach is that the refinement notion is more restrictive, being defined per event and no longer per model.

4 Development of a Sequential Program

We demonstrate the use of structured Event-B by means of a sequential program development, the extended Euclidian algorithm (Figure 2). The refinement steps are illustrated using the graphical notation of Figure 1. We believe it to be very useful for understanding the model more easily. For instance, the collapsed representation in Figure 3 of the final model of the sequential program shows nicely the control flow of the program. The example is large enough to illustrate the use of structured Event-B and small enough to fit fully into this article.

```

when  $b \neq 0$  then
upini:  $f, s, t, q, r := 0, \{0 \mapsto a\}, \{0 \mapsto b\}, \{0 \mapsto a \div b\}, \{0 \mapsto a \bmod b\}$ ;
        while  $r(f) \neq 0$  do
up:  $f, s(f+1), t(f+1), q(f+1), r(f+1) := f+1, t(f), r(f), t(f) \div r(f), t(f) \bmod r(f)$ 
        end;
dnini:  $D, U, V := t(f), 1, 1 - q(f)$ ;
        while  $f > 0$  do
dn:  $f, U, V := f-1, V, U - q(f-1) * V$ 
        end;
gcd:  $d, u, v := D, U, V$ 
end
    
```

Fig. 2. The extended Euclidian algorithm

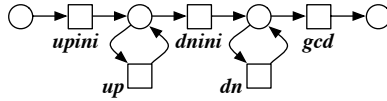


Fig. 3. Collapsed graphical representation of the final gcd model

In Section 4.1 we model the program to be developed in models $g0$ and $g1$. We refine model $g0$ into model $g1$ applying Bézout’s identity. We introduce the first loop creating a stack of divisions in the second refinement $g2$ in Section 4.2, and the second loop in $g3$ in Section 4.3. Finally, we data-refine two separate stack pointers used in the two loops into one in $g4$ in Section 4.4.

We use the following definitions of *divides*, denoted by $|$, of *GCD*, and of *abs*:

$$\begin{aligned}
 x|y &\Leftrightarrow x \neq 0 \wedge (\exists m \cdot y = x * m) \\
 z \in GCD[\{x \mapsto y\}] &\Leftrightarrow z|x \wedge z|y \wedge (\forall d \cdot d|x \wedge d|y \Rightarrow d|z) \\
 y = abs(x) &\Leftrightarrow (x \geq 0 \Rightarrow y = x) \wedge (x < 0 \Rightarrow y = -x)
 \end{aligned}$$

4.1 GCD by Way of a Linear Equation

The initial model consists of a single event *gcd*.

```

g0.gcd
when  $b \neq 0$  then
   $d := GCD[\{a \mapsto b\}]$ 
    
```

We assume that the variables a , b , and d cannot be data-refined. Similarly to Event-B, we require that variables that are kept in a refinement are implicitly linked by an equality (in all assertions). The initial structure term only states that *gcd* establishes *true* starting from *true* in one step,

$$true \rightarrow g0.gcd \rightarrow true$$

There is nothing to prove. Figure 4a shows a graphical representation of the initial model.



Fig. 4. The first two models of the development of the algorithm

The first refinement replaces the GCD relation by a linear diophantine equation with coefficients u and v :

$$\begin{aligned}
 & \mathbf{g1.gcd} \\
 & \text{when } b \neq 0 \text{ then} \\
 & \quad d, u, v : | d' = a * u' + b * v' \wedge d' | a \wedge d' | b
 \end{aligned}$$

The structure of the refinement is the same as that of the abstraction

$$\mathbf{g0.gcd} \sim \text{true} \rightarrow \mathbf{g1.gcd!} \rightarrow \text{true}$$

In the graphical representation we expand the square for event $\mathbf{g0.gcd}$ into a box containing the graphical representation of $\text{true} \rightarrow \mathbf{g1.gcd!} \rightarrow \text{true}$, see Figure 4b. The action simulation proof obligation of the two events $\mathbf{g0.gcd}$ and $\mathbf{g1.gcd}$, $d' = a * u' + b * v' \wedge d' | a \wedge d' | b \Rightarrow d' \in GCD[\{a \mapsto b\}]$ (Bézout's identity), for assertion preservation is easily discharged.

4.2 Creation of a Stack of Divisions

In the second refinement we build up a stack of divisions. Variable h points to the top of the stack that is described by assertion upinv :

$$\begin{aligned}
 @\text{upinv} \quad & s \in 0..h \rightarrow \mathbb{Z} \wedge t \in 0..h \rightarrow \mathbb{Z} \\
 @\text{upinv} \quad & q \in 0..h \rightarrow \mathbb{Z} \wedge r \in 0..h \rightarrow \mathbb{Z} \\
 @\text{upinv} \quad & h \geq 0 \wedge s(0) = a \wedge t(0) = b \\
 @\text{upinv} \quad & \forall i \cdot i \in 0..h \Rightarrow t(i) \neq 0 \wedge s(i) = t(i) * q(i) + r(i) \\
 @\text{upinv} \quad & \forall i \cdot i \in 1..h \Rightarrow t(i-1) = s(i) \wedge r(i-1) = t(i)
 \end{aligned}$$

Two new events are introduced. Event $\mathbf{g2.upini}$ initialises the loop that computes the stack and event $\mathbf{g2.up}$ models the loop body.

$$\begin{array}{ll}
 \mathbf{g2.upini} & \mathbf{g2.up} \\
 \text{when } b \neq 0 \text{ then} & \text{when } r(h) \neq 0 \text{ then} \\
 \quad h := 0 & \quad h := h+1 \\
 \quad s := \{0 \mapsto a\} & \quad s(h+1) := t(h) \\
 \quad t := \{0 \mapsto b\} & \quad t(h+1) := r(h) \\
 \quad q := \{0 \mapsto a \div b\} & \quad q(h+1) := t(h) \div r(h) \\
 \quad r := \{0 \mapsto a \bmod b\} & \quad r(h+1) := t(h) \bmod r(h)
 \end{array}$$

In the refined event $g2.gcd$ only the guard is strengthened. The action is unchanged. In fact, the result of the computation is ignored except for the termination condition $r(h) = 0$.

$$\begin{aligned}
 &g2.gcd \\
 &\text{when } r(h) = 0 \text{ then} \\
 &\quad d, u, v :| d' = a * u' + b * v' \wedge d'|a \wedge d'|b
 \end{aligned}$$

The introduction of the loop is expressed by the term

$$g1.gcd \sim true \rightarrow g2.upini \rightarrow upinv \rightarrow (g2.up \leftarrow [] g2.gcd! \rightarrow true)$$

Figure 5 shows the graphical representation of the model; the square for event $g1.gcd$ is replaced by a box representing the refined term. We believe that already this simple case demonstrates the value of the graphical representation. The picture is quite easy to comprehend.

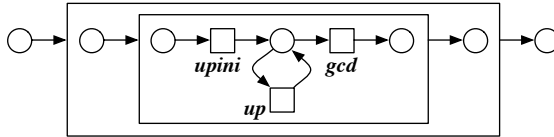


Fig. 5. Second refinement of the gcd model

Aside. We are not using the graphical representation to specify structure though: the textual representation is richer and feeding all information into the graphical representation would complicate it. The purpose of the graphical representation is to visualise an important aspect of the model. The syntax of structure terms is designed to resemble the graphical representation.

Assertion preservation is easily proved, for example, for preservation of assertion $\forall i \cdot i \in 0 .. h \Rightarrow t(i) \neq 0$ by events $g2.upini$ and $g2.up$,

$$\begin{aligned}
 b \neq 0 &\Rightarrow \forall i \cdot i \in 0 .. 0 \Rightarrow \{0 \mapsto b\}(i) \neq 0 \\
 upinv \wedge r(h) \neq 0 &\Rightarrow \forall i \cdot i \in 0 .. h+1 \Rightarrow t \Leftarrow \{h+1 \mapsto r(h)\}(i) \neq 0
 \end{aligned}$$

Convergence and Enabledness. In the term refining event $g1.gcd$ we have indicated that event $g2.up$ terminates. The ring of \mathbb{Z} is a Euclidian domain with abs as a Euclidian function: $\forall x, y \cdot y \neq 0 \Rightarrow (\exists q, r \cdot x = y * q + r \wedge abs(r) < abs(y))$. Hence, the expression $abs(r(h))$ is a variant for event $g2.up$, that is, $g2.up \leftarrow abs(r(h))$.

The proof obligations for enabledness (showing weakening of the precondition) are $b \neq 0 \Rightarrow b \neq 0$ and $upinv \Rightarrow r(h) \neq 0 \vee r(h) = 0$. Both are easily discharged.

4.3 Calculation of the Coefficients

In the third refinement we calculate the Bézout coefficients u and v and the gcd d by means of the variables D , U , and V . This refinement step is structurally

very similar to the second one, except that the stack pointer is decreased during the calculation.

$@dninv \text{ upinv}$
 $@dninv k \in 0..h \wedge r(h) = 0 \wedge dk|s(k) \wedge dk|t(k)$
 $@dninv dk = s(k) * uk + t(k) * vk$

The proof of the structure refinement

$g2.gcd \sim \text{upinv} \rightarrow g3.dnini \rightarrow dninv \rightarrow (g3.dn \leftarrow [] g3.gcd! \rightarrow true)$

makes use of the properties of the stack described by upinv and requires some arithmetic.

$g3.dnini$	$g3.dn$	$g3.gcd$
when $r(h) = 0$ then	when $k > 0$ then	when $k = 0$ then
$k, D := h, t(h)$	$k := k-1$	$d := D$
$U := 1$	$U := V$	$u := U$
$V := 1 - q(h)$	$V := U - q(k-1) * V$	$v := V$

In this step we also refine the gcd event to use the result of the preceding computation of the coefficients and the gcd. Figure 6 shows the graphical representation illustrating the control flow of the algorithm consisting of two consecutive loops preceded by an initialisation each.

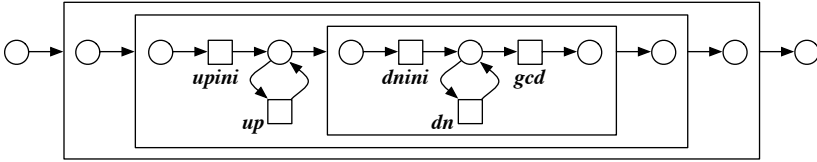


Fig. 6. Third refinement of the gcd model

Convergence and Enabledness. Convergence can be verified by means of the decreasing stack pointer k : $g3.dn \leftarrow k$. The enabledness proof obligations are $\text{upinv} \wedge r(h) = 0 \Rightarrow r(h) = 0$ and $dninv \Rightarrow k > 0 \vee k = 0$, both proved easily.

4.4 Implementation of the Stack Pointer

We sketch the fourth refinement in order to demonstrate the use of data refinement. In all events of model $g3$ we textually replace h and k by f . (In refined event $g4.dnini$ we can remove the resulting assignment $f := f$.)

$g2.upini \sim true \rightarrow g4.upini! \rightarrow \text{upinv}$
 $g2.up \sim \text{upinv} \rightarrow g4.up! \rightarrow \text{upinv}$
 $g3.dnini \sim \text{upinv} \rightarrow g4.dnini! \rightarrow dnini$
 $g3.dn \sim dninv \rightarrow g4.dn! \rightarrow \text{upinv}$
 $g3.gcd \sim dninv \rightarrow g4.gcd! \rightarrow true$

The assertions $upinv$ and $dninv$ are extended by gluing assertions relating model $g3$ to model $g4$

$$\begin{aligned} @upinv \quad f &= h \\ @dninv \quad f &= k \end{aligned}$$

Aside. In the algorithm shown in Figure 2 all variables are global. We could as well have inferred from the structure of the model (Figure 6) local variables f , s , t , q , r for the two loops and D , U , V for the second loop.

5 Related and Future Work

The two verification approaches presented in [16] and [14] are lacking a notion of refinement. In [8] a restricted form of refinement for temporal verification diagrams is presented that permits splitting vertices and removing edges. This is generalised in [7] by considering the transitive closure of edges and matching notion of data-refinement. In our approach, we have incorporated refinement based on the corresponding notion of Event-B that appears simpler to handle. We also preserve structure information during refinement which is particularly important for obtaining the intended algorithmic structure by the end of a development.

Alternative ways of expressing structure of Event-B models that have been proposed are the CSP-based approach of [13], JSD-based approach of [6] and UML-B [17]. In [13] events are annotated with events that are to be enabled *next*. Corresponding enabledness proof obligations are shown but refinement is not considered. In [6] JSD-like diagrams are used to illustrate concurrent Event-B models. The notation can also be used to illustrate refinement of Event-B models. However, the notation is not exploited for proof obligation generation; the suggested (still) informal mapping to Event-B introduces abstract program counters. UML-B [17] focuses more on states as its central concept. UML-B models are translated into Event-B by introducing abstract program counters to represent those states. The notion of refinement is centred around state decomposition and gluing invariants are generated from the emerging nesting structure of state machines. State machines of UML-B are not exploited for proof obligation generation.

Abstract State Machines (ASM) [5] also provide to ways to introduce structure into a model. Control State ASMs use abstract program counters to model control structures. We could also identify such a class of models in Event-B. But this would not solve our problem. One of the reasons for introducing a structure notation in Event-B is that proof obligations involving program counters can get quite involved. Apart from that invariants of Event-B get cluttered with properties involving abstract program counters. A second way to structure Abstract State Machines is to use Turbo ASMs [5]. Turbo ASMs provide programming constructs to compose ASMs to model computations. Such a reconstruction of programming constructs would not solve our problem either. We would again have to deal with sequential composition, if-statements, and so on, when generating proof obligations.

We are investigating modelling concurrency ($p_1 \rightarrow e_1 \rightarrow q_1 \parallel p_2 \rightarrow e_2 \rightarrow q_2$) using structured Event-B. In fact, the proof outlines in [16] were first developed for proving properties of concurrent programs. We are more interested to look into possibilities for support by a tool such as the Rodin tool [2]. The proof obligations for enabledness are difficult to handle. However, this difficulty is also present in unstructured Event-B models: in structured Event-B it will surface during proof obligation generation, whereas in unstructured Event-B it will show up during proof.

We also think about changes concerning the way proof obligations are generated. In the new scheme of proof obligation generation we would no longer get a list of all proof obligations that must be discharged. Instead, we would get a todo-list that tells us what still needs to be proved. We have realised that this is the way we should proceed with action feasibility and convergence proof obligations. Action feasibility can be proved showing the existence of a post-state directly or by implementing the action in a refinement. Convergence proofs can be delayed by using anticipation. This approach would make the Event-B method more flexible without sacrificing its strong tool support.

We are also investigating verification of temporal properties. The temporal verification diagrams in [14] are used to prove temporal logic properties of reactive systems. A similar approach should also work for structured Event-B. Refinement should provide a way to master more complex temporal properties. The structured Event-B approach shares with temporal verification diagrams the strength of only generating first-order proof obligations.

6 Conclusion

We have introduced a structure notation for Event-B together with the necessary proof obligations. We have demonstrated how it can be used practically in a sequential program development. The structure notation is *not* a programming notation but a notation that describes a theory about a formal model. For instance, the formula $p \rightarrow e \rightarrow q \rightarrow f \rightarrow r$ does not mean: first e is executed then f ; it describes theorems about e and f . Effectively, we have moved some concepts of proof into modelling. We think this is very attractive, in particular, for implementation in a software tool such as Rodin. There would be no need to configure the proof obligation generator for different applications. Everything about the proof obligations would be said in the model; fully transparent for the user of the tool. An additional benefit would be that only proof obligations need to be generated that are specified in structure terms. Hence, usually, fewer proof obligations would be generated.

In our opinion, the notation also improves legibility of more complex structured models. The associated graphical notation helps to grasp quickly the structure of a model.

Acknowledgement. The original (unstructured) Event-B model of the extended Euclidian algorithm was developed by Christophe Métayer. Jens Bendisposto provided useful comments on earlier versions of this article.

References

1. Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2009) (to appear)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform* 77(1-2), 1–28 (2007)
4. Apt, K.R., de Boer, F.S., Olderog, E.-R.: *Verification of Sequential and Concurrent Programs*. Springer, Heidelberg (2009)
5. Börger, E., Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003)
6. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) *IFM 2009*. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
7. Cansell, D., Méry, D., Merz, S.: Diagram refinements for the design of reactive systems. *J. UCS* 7(2), 159–174 (2001)
8. de Alfaro, L., Manna, Z., Sipma, H.B., Uribe, T.E.: Visual verification of reactive systems. In: Brinksma, E. (ed.) *TACAS 1997*. LNCS, vol. 1217, pp. 334–350. Springer, Heidelberg (1997)
9. Hallerstede, S.: Justifications for the Event-B Modelling Notation. In: Julliand, J., Kouchnarenko, O. (eds.) *B 2007*. LNCS, vol. 4355, pp. 49–63. Springer, Heidelberg (2006)
10. Hallerstede, S.: On the Purpose of Event-B Proof Obligations. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 125–138. Springer, Heidelberg (2008)
11. Hallerstede, S.: Proving Quicksort Correct in Event-B. In: *Refine 2009*, 16 pages (2009)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* 12, 576–580 (1969)
13. Ifill, W., Schneider, S.A., Treharne, H.: Augmenting B with control annotations. In: Julliand, J., Kouchnarenko, O. (eds.) *B 2007*. LNCS, vol. 4355, pp. 34–48. Springer, Heidelberg (2006)
14. Manna, Z., Pnueli, A.: Temporal verification diagrams. In: Hagiya, M., Mitchell, J.C. (eds.) *TACS 1994*. LNCS, vol. 789, pp. 726–765. Springer, Heidelberg (1994)
15. Morgan, C.: *Programming from Specifications*, 2nd edn. Prentice Hall, Englewood Cliffs (1994)
16. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. *Acta Informatica* 6(4), 319–340 (1976)
17. Said, M.Y., Butler, M.J., Snook, C.F.: Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009*. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009)

Refinement-Animation for Event-B — Towards a Method of Validation^{*}

Stefan Hallerstede, Michael Leuschel, and Daniel Plagge

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

Abstract. We provide a detailed description of refinement in Event-B, both as a contribution in itself and as a foundation for the approach to simultaneous animation of multiple levels of refinement that we propose. We present an algorithm for simultaneous multi-level animation of refinement, and show how it can be used to detect a variety of errors that occur frequently when using refinement. The algorithm has been implemented in PROB and we applied it to several case studies, showing that multi-level animation is tractable also on larger models.

Keywords: Refinement, Model Checking, Constraint-Solving, Tools, Industrial Applications.

1 Introduction and Motivation

The Event-B modelling method [1] has been designed to be complemented by a software tool such as Rodin [2]. The core of the Rodin tool provides automatic generation of proof obligations that can be analysed to improve understanding of a model. Often proof obligations give good indications of how to make an improvement in case of inconsistencies in a model. However, there are also many occasions where proof obligations do not point directly to a problem or where a model does not contain inconsistencies but is still “incorrect” (see, e.g., the Earley parser example discussed in [7]). In these cases animation is a useful tool to gain further insight into a model. The Rodin plugins PROB [9,11], Brama [13], and AnimB [12] provide animation facilities for Event-B.

When dealing with complex models, refinement can be used to introduce the many details gradually, achieving a reduced complexity at each refinement level. It can be difficult to analyse a refinement relationship only by means of associated proof obligations. All three animation plugins mentioned above provide some means to animate refinements. In this article we investigate their relative capabilities and how to advance refinement animation in order to turn it into a tool for refinement validation. This serves as a blueprint for the evolution of PROB in terms of animation support.

^{*} Part of this research has been EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

¹ Brama requires an older version (0.9.2.x) of Rodin at the time of writing.

Before starting the investigation we should be clear on the objectives of animation when used for validation. What is the purpose of animating a model across multiple levels of refinement? In Event-B several concepts play a rôle in refinement. Most prominently, these are invariants, guards, actions, and witnesses. If a refinement fails, any combination of those concepts may be involved. Animation should help to locate the cause of a problem in the model pointing to specific invariants, guards, and so on, if possible. However, even if a refinement is formally correct, there can still be problems with the model. This concerns, in particular, properties that have not been formalised. Animation should make it easy to experiment with a model, visualising potential problems. We try to integrate this aspect of animation with the first one as far as possible. Otherwise consistent tool support for both would be difficult to realise.

In Section 2 we give a concise description of the fundamentals of Event-B refinement. As far as we are aware there is no single place where all the essential aspects are described and motivated in such detail. All of this is needed in order to present the basic refinement-animation algorithm in Section 3. The presentation of the algorithm is interspersed with methodical remarks on validation. In Section 4 we present some concrete examples on how to use PROB for refinement validation and some brief description of case studies to which it has been applied. Finally, Sections 5 and 6 contain a discussion of related work and a conclusion.

2 Modelling and Refinement in Event-B

Event-B can be used to model complex intricate systems. To understand the system and the model of the system we need to reason thoroughly about the model. Such reasoning is the principal purpose of Event-B. The basic concepts of Event-B are characterised by means of proof obligations; they are the core of the Event-B method. However, they are not an exclusive means of reasoning. Based on an operational interpretation of a model we can also animate it to gain deeper understanding. In this section we parallel the presentation of Event-B proof obligations, in particular, refinement, with ideas of animation. This demonstrates well how animation complements proof. Because there is no single software tool for animation that provides all that is needed, we use the three tools PROB, Brama, and AnimB at the same time.

2.1 Contexts

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts specify static parts of a model, that is, carrier sets and constants that are constrained by axioms. Usually, these are quite simple formulas. Contexts are intended to be used to parametrise machines. We mention contexts here because of the rôle they play in animation. For any particular animation specific values for all constants have to be found. PROB does this automatically using constraint-solving techniques to find proper values that satisfy all axioms. The constraint-solving also determines whether the axioms contain a contradiction.

2.2 Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* of the event. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by one of the following forms:

any t when	or	when	or	begin
$G(t, v)$		$G(v)$		$S(v)$
then		then		end
$S(t, v)$		$S(v)$		
end		end		

The second form is used if event $E(v)$ does not have parameters, and the third form if in addition the guard equals *true*. A dedicated event of the third form is used for *INITIALISATION*. In the formal exposition below, we assume without loss of generality that the most general first form is used.

The action of an event is composed of several *assignments* of the form: $x := E(t, v)$ or $x := E(t, v)$ or $x :| Q(t, v, x')$, where x are some variables, $E(t, v)$ expressions, and $Q(t, v, x')$ a predicate. The second form assigns x to an element of a set, and the third form assigns to x a value satisfying a predicate. Without loss of generality, the first two can be formally defined in terms of the third form: $x := E(t, v) \hat{=} x :| x' = E(t, v)$ and $x := E(t, v) \hat{=} x :| x' \in E(t, v)$. The effect of an assignment is described by a before-after predicate:

$$\text{before-after predicate of “} x :| Q(t, v, x') \text{”} \hat{=} Q(t, v, x')$$

A before-after predicate describes the relationship between the state just before an assignment has occurred, x , and the state just after the assignment has occurred, x' . All assignments of an action $S(t, v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(t, v, x')$. Variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $A(t, v, x')$ with $y' = y$, yielding the predicate:

$$S(t, v, v') \hat{=} A(t, v, x') \wedge y' = y \quad .$$

Running Example. We use the coffee dispenser model in Fig. [□](#) for illustration of refinement-animation. In the abstract machine *CoffeeM* the dispenser can fill a mug half or fully; the state of the mug is represented by the variable *alvl* (abstract level). As a special service the dispenser can also drink the coffee. In the first refined machine *CoffeeR1* a feature is introduced for inserting an arbitrary number of coins into the dispenser. A coin is consumed each time a mug is filled. In the second refined machine *CoffeeR2*, the number of coins maximally accepted

```

machine CoffeeM sees CofCtx
variables abvl
invariants @inv1 abvl ∈ FILL
variant ({full ↦ 2, half ↦ 1, empty ↦ 0})(abvl)
events
event INITIALISATION
begin
  @mf abvl := empty
end
event fill_mug
any x when
  @g0 abvl = empty
  @g1 x ≠ abvl
then
  @a1 abvl := x
end
convergent event drink
when @g1 abvl ≠ empty then
  @a1 abvl ∈ {empty, half} \ {abvl}
end
end

machine CoffeeR1 refines CoffeeM sees CofCtx
variables abvl coins
invariants @ci coins ∈ N
events
event INITIALISATION extends INITIALISATION
begin
  @ai coins := 0
end
event fill_mug extends fill_mug
when @gc coins > 0 then
  @delc coins := coins - 1
end
convergent event drink extends drink
end
anticipated event insert_coin
begin
  @insc coins := coins + 1
end
end

context CofCtx
constants full empty half level
sets FILL
axioms
  @Fthe partition(FILL, {full}, {half}, {empty})
  @lvl level = (0..2 × {empty}) ∪ (3..7 × {half}) ∪ (8..11 × {full})
end

machine CoffeeR2 refines CoffeeR1 sees CofCtx
variables clvl coins maxc
invariants
  @imc maxc ∈ N1
  @iff abvl ∈ 0..11
  @lvl abvl = level(clvl)
variant maxc - coins
events
event INITIALISATION
begin
  @mc maxc := 4
  @cci coins := 0
  @ffi clvl := 0
end
event fill_mug refines fill_mug
when
  @gc2 coins > 0
  @ml clvl ∈ level-1{empty}
with
  @x x = level(clvl)
then
  @delc2 coins := coins - 1
  @ffl clvl ∈ level-1{full}
end
convergent event drink refines drink
when
  @dgfl clvl ∉ level-1{empty}
with
  @alvl' abvl = level(clvl)
then
  @dff clvl ∈ level-1{empty, half} \ {level(clvl)}
end
convergent event insert_coin extends insert_coin
when
  @gmc coins < maxc
end
end
    
```

Fig. 1. Coffee dispenser model (using syntax of the Event-B text editor “Camille”)

is limited and the amount of coffee contained in a mug is represented numerically by the variable *clvl* (concrete level).

Animation in ProB. The before-after predicate can be used to compute the state space of a machine, a graph where each node represents a state of the machine and each arc the execution of an event. Fig. 2 contains the state space of the *CoffeeM* machine from Fig. 1, as computed by the ProB tool. The triangle represent a special root node, where the variables and constants of a machine have not yet been set. An animator lets the user navigate the state space by choosing the events to be fired. A model checker will systematically explore the state space, looking for various errors in the machine.

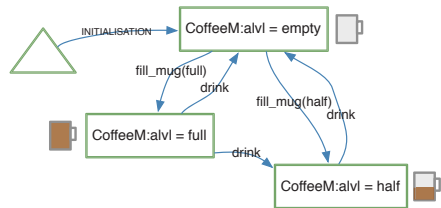


Fig. 2. State space for *CoffeeM* (with additional mugshots)

2.3 Machine Consistency

Invariants are supposed to hold initially and whenever variable values are changed by an event. Obviously, this does not hold a priori and, thus, needs to be proved. The corresponding proof obligation for every event is called *invariant preservation*, formally, $I(v) \wedge G(t, v) \wedge \mathbf{S}(t, v, v') \Rightarrow I(v')$. There is a special form of this proof obligation, without invariant and guard in the hypothesis, for the *INITIALISATION*. By proving *action feasibility* for an event, $I(v) \wedge G(t, v) \Rightarrow (\exists v'. \mathbf{S}(t, v, v'))$, as well, we achieve that $\mathbf{S}(t, v, v')$ provides an after state whenever $G(t, v)$ holds. This means that the guard indeed represents the enabling condition of the event.

2.4 Machine Refinement

A machine N can refine at most one other machine M . We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$ associated with the concrete machine N , where v are the variables of the abstract machine and w the variables of the concrete machine.

Each event $E(v)$ of the abstract machine is *refined* by one or more concrete events $F(w)$. Let abstract event $E(v)$ and concrete event $F(w)$ be:

$$\begin{aligned} E(v) &\hat{=} \text{ any } t \text{ when } G(t, v) \text{ then } S(t, v) \text{ end} \\ F(w) &\hat{=} \text{ any } u \text{ when } H(u, w) \text{ with } W(t, v', u, v, w, w') \text{ then } T(u, w) \text{ end} \end{aligned}$$

Informally, concrete event $F(w)$ refines abstract event $E(v)$ if, whenever the gluing invariant $J(v, w)$ is true: (i) the guard of $F(w)$ is stronger than the guard of $E(v)$, and (ii) for every possible execution of $F(w)$ there is a corresponding execution of $E(v)$ which simulates $F(w)$ such that the gluing invariant remains true after execution of both events. In Fig. 11 some events carry the attribute “extended”. This means that all parameters, guards, and actions are copied literally from the abstract event.² Note that the event $F(w)$ contains one more component $W(t, v', u, v, w, w')$ following the keyword *with*, called the *witnesses*. We return to its rôle in Section 2.6 below.

Refinement Animation. To check whether the guard of a concrete event is stronger, we also need to animate the corresponding abstract machine. Fig. 3 shows a graphical visualisation (created with Brama) of an animation of the coffee dispenser model described earlier. Green boxes signal enabled events, red boxes disabled events. As can be seen,

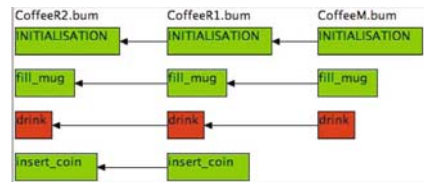


Fig. 3. Coffee dispenser refinement-animation in Brama

² In Event-B it is also possible for an event to refine more than one abstract event, merging these events into one concrete event [3]. Such events cannot be extended.

all the refinement levels are animated concurrently. Brama’s representation also shows at a glance that whenever a refined event is enabled, then all of its ancestor events are also enabled.

The view underlying Fig. 3 is operational. It focuses solely on event execution. If we wanted to use it for analysing a formal model, we would need to add information. In particular, information about gluing invariants would be useful (Fig. 4). Note, that this is not a purely cosmetic change: the animator must supply all necessary information.

CoffeeR2	Inv	CoffeeR1	Inv	CoffeeM
INITIALSATION		INITIALSATION		INITIALSATION
fill_mug	⊖	fill_mug	⊖	fill_mug
drink	↓	drink	↓	drink
insert_coin	↓	insert_coin	→	

Fig. 4. Improved coffee dispenser refinement-animation

2.5 Common Variables and Common Parameters

As far as animation and model checking are concerned, refinement introduces a new challenge: we no longer have just a single machine that needs to be animated as in Fig. 2, but a series of machines, each with its own state.³

In order to check the gluing invariant, we need to access variables from various machines. This raises a new issue. In Section 2.4 we have simply assumed that all variables v are refined by new variables w , and all parameters t are refined by new parameters u . The variables v and parameters t “disappear” in the refinement. In practice, variables and parameters can be repeated in a refinement. Abstract machines and concrete machines can have variables in common, and abstract events and concrete events parameters. By convention, when repeating variables and parameters abstract and concrete counterparts are assumed to be equal.⁴

Animation. For refinement animation of machines this means that variables must be renamed in each machine and gluing invariants generated. If the animation would operate on variables shared between different machines, it would not be possible to visualise machines with deviating behaviour. This also affects the witnesses described in the following section.

Example. In the example from Fig. 1, the machine *CoffeeM* and *CoffeeR1* have the variable *alvl* in common. Fig. 5 shows PROB animating the Coffee example. As can be seen in the newly developed hierarchical “State View”, the variable *alvl* occurs twice, once in *CoffeeM* and once in *CoffeeR1*. We can also see that the variable *alvl* disappears when going to *CoffeeR2*. In Fig. 6 we show how the AnimB animator displays a state of multiple refinement-levels; each refinement level is given its own tab.

³ Earlier versions of PROB avoided this problem by animating each refinement level separately, at the cost of not being able to check the gluing invariant and of less user feedback.

⁴ Once a variable has disappeared in the course of several refinements it cannot reappear. The reason for this is that the equality cannot be established by means of a machine that does not contain the variable. Furthermore, invariants are accumulated in Event-B. So it is not possible to reintroduce a variable with a different meaning.

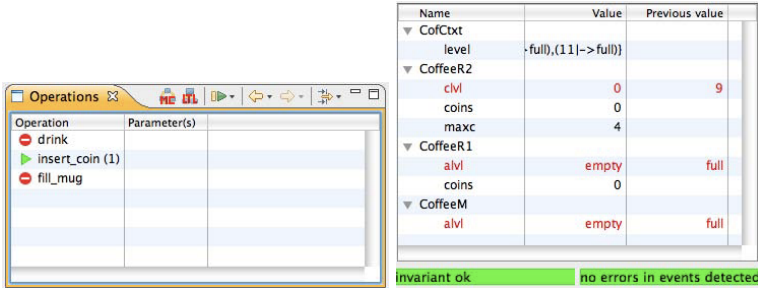


Fig. 5. PROB Operations and State View after the trace *insert_coin, fill_mug, drink*

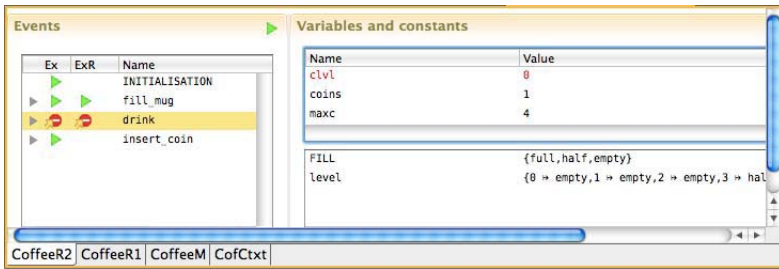


Fig. 6. AnimB View after *insert_coin, insert_coin, fill_mug, drink*

2.6 Refined Events and Witnesses

The predicate $W(t, v', u, v, w')$ denotes *witnesses*. Somewhat simplified, they link the abstract parameters t and the abstract variables v' to concrete parameters u and variables and w' (see also Fig. 7). Witnesses describe for each event separately how the refinement is achieved. Let $K(v, w) \hat{=} I(v) \wedge J(v, w)$.

Aside. As described in [4], in order to verify that $F(w)$ refines $E(v)$ we have to prove $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \Rightarrow \exists t, v' \cdot G(t, v) \wedge \mathbf{S}(t, v, v') \wedge J(v', w')$. In a proof of this statement we prefer to instantiate the quantified parameters and variables t and v' by expressions that can in some way inferred from the premises. This idea is generalised to the witnesses used in Event-B. Witnesses are predicates that provide values to satisfy the conclusion of the statement.

The proof obligations for concrete machines are called *guard strengthening*: $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow G(t, v)$, *action simulation*: $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow \mathbf{S}(t, v, v')$, and *invariant preservation*: $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \wedge W(t, v', u, v, w, w') \Rightarrow J(v', w')$. We have to prove *witness feasibility* in order to be able to add the witness predicate $W(t, v', u, v, w, w')$ to the premises in the proof obligations above: $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \Rightarrow (\exists t, v' \cdot W(t, v', u, v, w, w'))$.

In general, witnesses would be required for all parameters p of an event but when a parameter is repeated in a refined event, by convention, it is assumed to be equal to the corresponding abstract parameter. If a parameter is not repeated

an explicit witness is required. (The Rodin tool creates the *default witness* “true” if none is specified in the latter case. This witness does not constrain the relationship between abstract and concrete parameters and variables. Hence, usually default witnesses are not sufficient to establish the refinement relationship.) For variables the rule when witnesses are needed is more complicated: whenever a variable x that disappears occurs in a non-deterministic assignment in the abstract event, in the refined event a witness for the post-state variable v' is required.

Animation. For animation we have to take care that as a consequence of variable and parameter renaming (resulting from repeated variables), some witnesses may have to be generated. Combined with the generated gluing invariant (as described in Section 2.5) they provide an opportunity to locate refinement mismatches and provide meaningful feedback to the user.

Example. Event *fill_mug* in *CoffeeR2* contains the witness $x = level(clvl')$ for the abstract parameter x of *fill_mug* in *CoffeeR1*. This means intuitively, that every execution of *fill_mug* in *CoffeeR2* corresponds to an execution of *fill_mug* in *CoffeeR1* with parameter x set to $level(clvl')$. Event *fill_mug* in *CoffeeR1* must be enabled for $x = level(clvl')$ and the gluing invariant $alvl = level(clvl)$ must hold after executing the abstract and concrete event. Similarly, *drink* in *CoffeeR2* contains a witness $alvl' = level(clvl')$ for the abstract variable $alvl$. (Note, that it is just invariant @lvl in Fig. 1 with all variables primed.)

Animation in PROB. Witnesses are the key concept that makes refinement animation possible. Indeed, refinement animation and refinement checking in classical B require for every concrete state to keep track of the set of all abstract states for which the gluing invariant holds. Only if this set becomes empty, have we found an error in the refinement. The size of state space necessary for simulation grows exponentially. In Event-B, by contrast, the witnesses pinpoint the states which have to satisfy the refinement relationship (see Fig. 7).

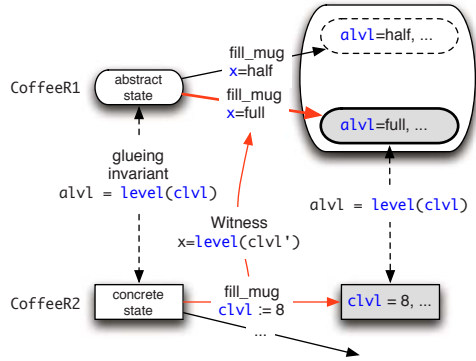


Fig. 7. Witnesses and Multi-Level Animation

2.7 New Events and Convergence

In the course of refinement, often *new events* $F(w)$ are introduced into a model. New events must be proved to refine the implicit abstract event *skip* that does nothing. Moreover, it may be proved that new events do not collectively diverge by proving that a specified *variant* $V(w)$ is bounded from below: $K(v, w) \wedge$

$H(u, w) \Rightarrow V(w) \geq 0$ and is decreased by each new event $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \Rightarrow V(w') < V(w)$ where we assume that the variant is an integer expression. (Instead of an integer expression also a finite set expression can be used.) We call events that satisfy these two proof obligations *convergent*. *Anticipated* events can be used to prove convergence on a lexicographic order or just to delay convergence proofs. Anticipated events can be refined by anticipated or convergent events, but must ultimately be refined by a convergent event. For an anticipated event the second proof obligation is replaced by $K(v, w) \wedge H(u, w) \wedge \mathbf{T}(u, w, w') \Rightarrow V(w') \leq V(w)$.

Example. Event `insert_coin` in Fig. 1 is anticipated in `CoffeeR1` and is then proven convergent in `CoffeeR2` by introducing an upper bound on the number of inserted coins.

2.8 Enabledness of Refined and New Events

We may prove that whenever the abstract machine may continue by means of event $E(v)$ with guard $G(t, v)$ then the concrete machine may continue by means of concrete event $F(w)$ or some other events $F_1(w), \dots, F_k(w), K(v, w) \wedge G(t, v) \Rightarrow (\exists u \cdot H(u, w)) \vee (\exists u_1 \cdot H_1(u_1, w)) \vee \dots \vee (\exists u_k \cdot H_k(u_k, w))$. The Rodin tool does not support enabledness proof obligations at the moment. But PROB supports analysis of liveness properties and animation can show a deadlock (where all events except for the initialisation are disabled).

3 Description of the Multi-level Animation Algorithm

In this section we describe the validation and animation algorithm in detail. We point out in the presentation of the algorithm how it indicates problems with particular proof obligations. We also show how feedback to the user needs to be considered. Producing informative output from an animation with good performance is a challenge. For this reason the algorithm makes heavy use of PROB's existing functionality. In particular, PROB provides methods to find values for variables that satisfy predicates occurring in Event-B models.

Below we limit discussion to animation; but the algorithm is identical for model checking: the model checker uses the same technique to determine the state space.

3.1 Preprocessing

The algorithm is applied to a particular refinement machine M_i of a model. In a pre-processing step, all ancestor machines M_0, \dots, M_{i-1} of M_i are loaded and all contexts seen by M_0, \dots, M_i are merged by collecting the declared constants and joining the axioms. All variables and constants are tagged according to the model or context where they are defined. The invariant is obtained by conjoining all invariants of M_0, \dots, M_i .

We transform each event of M_i to an internal representation. The representation is outlined on the right hand side of Fig. 8. Usually the list of abstract events contains just one entry. If an event refines *skip* or belongs to the most abstract machine M_0 , the list of abstract events is empty. If the event refines several events, it will contain all of those events.

3.2 The Animation Algorithm

The animator executes events depending on the current state of a model. It maintains a state consisting of all constants of the seen contexts as well as all variables of the machines M_0, \dots, M_i .

In a first step the animator tries to find values for the constants that satisfy all axioms. Subsequently, the animator executes in each step an event of the most concrete machine M_i ; then it executes the corresponding abstract events from the concrete event to the most abstract event.

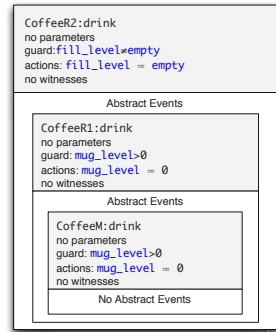
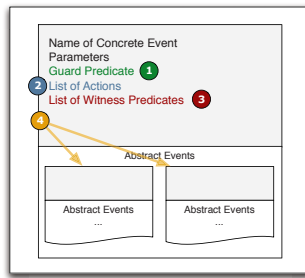


Fig. 8. Illustration of the algorithm and one particular event structure

When all of this has been done, the animator is ready for the next step.

The algorithm to animate a particular event works as follows (item numbers correspond to those of Fig. 8, left hand side):⁵

1. Search for possible values for the parameters by evaluating its guard. If no values are found, the event is disabled.
2. Execute each action by evaluating the respective before-after predicate. If no solution is found, report an error. The possible reasons for a failing action are:
 - a. The predicate P of an action $v :| P$ is not satisfiable or the set S of an action $v : \in S$ is empty. Both cases show violations of the event feasibility proof obligation.
 - b. The new value v' of a variable v was previously determined by a witness of a refined event (see step 3), but the abstract action cannot assign the same value to v' . This indicates a violation of the action simulation proof obligation.

⁵ With respect to animation *INITIALISATION* is not treated differently from any other events (see left of Fig. 8) except that it is enforced to occur once upon start of an animation.

3. For each witness evaluate its predicate and try to find values for the witnessed variable. If no value is found for a witness, report an error, because a witness should have at least one solution (by the witness feasibility proof obligation).
 4. a. If the list of abstract events is empty, a complete solution has been found for this event that leads to a new state. It consists of the values newly assigned by the actions plus the variables unchanged by the actions.
 - b. If there are one or more abstract events, choose one nondeterministically and evaluate its guard like in step 1. If it evaluates to true, continue recursively with step 2, otherwise try the next event.
- If no guard evaluates to true, report an error, because the guard of the refinement is weaker than that of the abstract event (violation of the guard strengthening proof obligation).

All four steps can be nondeterministic and we generate all solutions (limited to a maximum number) with backtracking.

Animation of convergent and anticipated events. If we have successfully found a possible event leading from one state to another, we can easily check if the convergence criteria are satisfied. The principle is quite simple: for each convergent event of an animated model, we check if the variant V is decreased and non-negative by the predicates $V > V'$ and $V \geq 0$ resp. $V \supset V'$ when the variant is a set. If the event refines another convergent event, we omit the test because in the lexicographic order constructed by refinement, events that have been shown to decrease a variant in an abstraction of some concrete machine may increase the variant of the concrete machine. Similarly, we can check anticipated events (with $V \geq V'$ and $V \geq 0$ resp. $V \supseteq V'$), but we cannot omit the test if an event is a refinement of an anticipated event.

Animating only a part of the refinement chain. Above we presumed that the user wants to animate a refinement M_i and all its ancestors M_0, \dots, M_{i-1} . But we also permit the user to limit the animation to the refinements between M_i and an “upper” refinement M_k with $0 \leq k \leq i$ instead of M_0 . Then variables of not animated models and predicates that contain references to those variables will be removed.

4 Refinement-Validation with PROB

Refinement animation can be used to validate models. We present some specific problems that can be analysed by animation and discuss a selection of case studies to which it has been applied to.

4.1 Detection of Specific Problems

Below we show on various modified versions of the coffee model (Fig. 1), how the new multi-level animation algorithm allows PROB to detect a variety of refinement errors. Note that in contrast to AnimB and Brama, PROB can be driven by a model checker so as to systematically detect refinement errors.

Guard Weakening. If we remove the guard `@ml` from the event `fill_mug` in *CoffeeR2*, we violate the guard strengthening proof obligation. As can be seen in Fig. 9, PROB’s model checker using our new algorithm detects this problem straightaway (case 4a) of our algorithm), leading us to a state where `fill_mug` is enabled in *CoffeeR2* but not in the abstract machines.

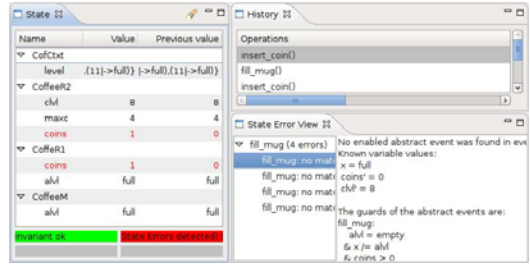


Fig. 9. Violation of Guard Strengthening (PROB)

Witness disables abstract guard. A similar error message appears if we keep the guards as they are, but inject an error in the witness. E.g, when using $x = empty$ as witness for `fill_mug`, PROB detects that there is a solution for the witness, but that the witness does not enable the abstract event.

Witness not feasible. Next, let us use the witness $x = level(clvl) \wedge x = empty$ for event `fill_mug`. Here case (3) of our algorithm detects an error for `fill_mug` (after executing `insert_coin`), and PROB displays the error message: “No solution found for witness of the abstract parameter x in event CoffeeR2:fill_mug”. The animator AnimB does not detect this error (but it did detect the previous two errors).

Witness violates invariant. Finally, we try to specify the witness $alvl \in \{empty, half\} - \{alvl\}$ for the event `drink`, which does not guarantee that the abstract event will satisfy the gluing invariant. As can be seen in Fig. 10, PROB finds an invariant violation error ($alvl = level(clvl)$ is false) directly after the drink event.

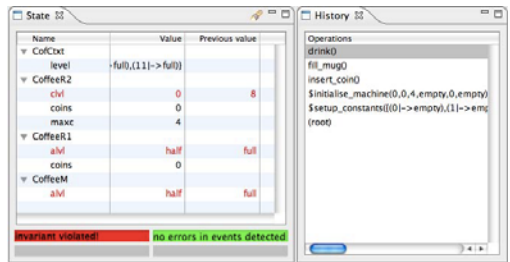


Fig. 10. Violation of Gluing Invariant (PROB)

Note that, AnimB detects an error in the model, but only later when trying to execute the `fill_mug` event after the erroneous `drink` event.

In practice, validation by animation complements the proof-based methodology of core Event-B. Corresponding methodological benefits of using animation of Event-B models are discussed in more detail in [8].

4.2 Application to Case Studies

We have successfully applied the new multi-level animation of PROB on a two-level model of SAP service choreographies [14]. We have also tested the tool on

the CDIS air traffic control case study carried out in the EU project Rodin. Figure 11 contains a screenshot of the first two levels; we have successfully animated all 7 levels of the full model concurrently.

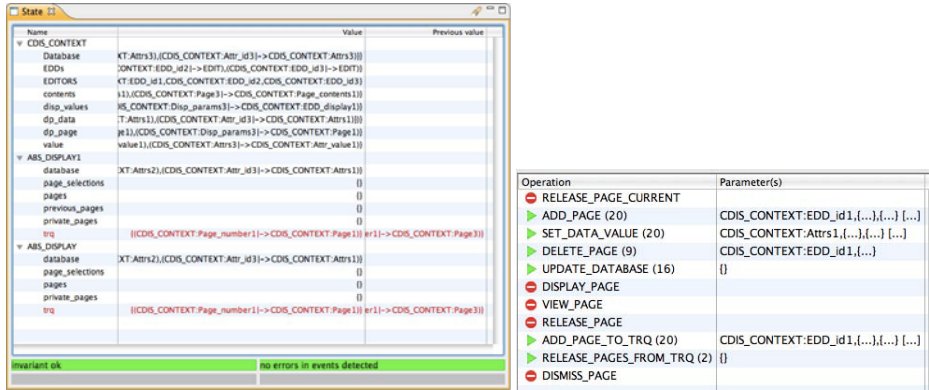


Fig. 11. Animating the Rodin CDIS case study

Another case study was a complete development of the quicksort algorithm in Event-B, consisting of ten machines and two contexts. We have successfully animated and model checked all the ten levels concurrently. Animation showed how the algorithm “works” at different abstraction levels. This is valuable for explaining an otherwise static model of an algorithm.

We have also successfully animated concurrently 14 levels of an elevator model solution by ETH Zürich. This has uncovered a potential problem in the model, namely that starting at a certain refinement level, the lift is no longer able to move (but the doors can be opened and closed and the buttons can be pressed; so there is no deadlock in the conventional sense).

5 Related Work

As already indicated above, the tools Brama and AnimB are also capable of performing multi-level animation of Event-B models, and have partially inspired this work. Unfortunately there is little scientific or technical documentation available for both of these tools. A few notable differences are

- Both Brama and AnimB require to specify explicitly values for constants; i.e., we had to “calculate” the cartesian products for the level constant in Fig. 11 by hand.
- PROB can be driven by a model checker to systematically search for errors, and to validate LTL formulas.
- PROB uses a constraint solving approach to find solutions for predicates, while AnimB and Brama seem to rely on pure enumeration. As such, PROB can evaluate much more complicated guards and predicates than AnimB or Brama.

Another animator for Event-B is [5]; but it does not yet seem to support refinement animation. The same is true for the animator in [6] for classical B. Another related work is the refinement checking algorithm in [10]. This algorithm does not have access to Event-B’s witnesses and hence has to keep track of sets of states in the abstract model (and does not check the gluing invariant as the traces of the abstract and refined model are computed separately).

6 Conclusion

We have presented a description of refinement in Event-B and have shown how a suitable animation and validation algorithm can be developed. The key ingredient that makes the algorithm tractable are the witnesses of Event-B. We have implemented the algorithm within PROB, and have shown how a variety of refinement errors can now be detected effectively. We have applied the technique to various case studies, and have animated up to 14 levels simultaneously.

In future work, we plan combining the graphical representation of Brama from Fig. 3 with the validation features of PROB. As we have sketched in Fig. 4, we also would like to be able to see when an event is disabled in a concrete machine but enabled in an abstract machine (Brama does not compute this information), and also to visualize the gluing invariant of each refinement level individually. We would also like to visualise the errors found by PROB inside the Rodin models, e.g., so that the offending proof obligations can be marked as “not provable.”

References

1. Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2009) (to appear)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Abrial, J.-R., Cansell, D., Méry, D.: Refinement and Reachability in EventB. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) *ZB 2005*. LNCS, vol. 3455, pp. 222–241. Springer, Heidelberg (2005)
4. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-B. *Fundam. Inform.* 77(1-2), 1–28 (2007)
5. Aït-Sadoune, I., Ameur, Y.A.: Animating event b models by formal data models. In: Margaria, T., Steffen, B. (eds.) *ISoLA. Communications in Computer and Information Science*, vol. 17, pp. 37–55. Springer, Heidelberg (2008)
6. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Utting, M., Vacelet, N.: BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In: *Proceedings of FATES 2002*, August 2002, pp. 105–120 (2002); Technical Report, INRIA
7. Bendisposto, J., Leuschel, M., Ligoit, O., Samia, M.: La validation de modèles Event-B avec le plug-in Prob pour RODIN. *Technique et Science Informatiques* 27(8), 1065–1084 (2008)
8. Hallerstede, S., Leuschel, M.: How to explain mistakes. In: Gibbons, J., Oliveira, J.N. (eds.) *TFM 2009*. LNCS, vol. 5846, pp. 105–124. Springer, Heidelberg (2009)

9. 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)
10. Leuschel, M., Butler, M.: Automatic refinement checking for B. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 345–359. Springer, Heidelberg (2005)
11. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. *STTT* 10(2), 185–203 (2008)
12. Métayer, C.: AnimB Homepage, <http://www.animb.org/index.xml>
13. Servat, T.: Brama: A new graphic animation tool for B models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 274–276. Springer, Heidelberg (2006)
14. Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., Schieferdecker, I.: Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. In: Núñez, M. (ed.) TESTCOM/FATES 2009. LNCS, vol. 5826, pp. 179–194. Springer, Heidelberg (2009)

Reactivising Classical B

Steve Dunne¹ and Frank Zeyda²

¹ School of Computing
University of Teesside, Middlesbrough, UK
s.e.dunne@tees.ac.uk

² Department of Computer Science
University of York, York, UK

Abstract. We propose what is essentially a recasting of *Circus*, the Z-and-CSP-based concurrent language for refinement, into a B context by means of a modest extension of classical B which introduces a new structural component called a reactive-B *process*. This specifies the *channels* via which the process can communicate with its environment, and *actions* by which its behaviour is specified. Such actions are expressed in a new action notation in the same syntactic spirit as B's Abstract Machine Notation, but with a similar Unifying Theories of Programming relational semantics to that of the actions of *Circus*. Crucially, by including ordinary abstract machines these reactive-B processes can also acquire persistent state, which their actions can manipulate by invoking the operations of those included machines.

1 Introduction

For almost as long as formal methods have been utilised in software engineering there has been interest in combining state-based and behavioural formalisms. In particular during the last decade or more the B Method [1] has been variously adapted for expressing behavioural aspects of systems, as in Event-B [8], or combined with an explicit behavioural formalism such as CSP [6,12,13], as in CSP||B [19,14,15] or csp2B [3]. However, in this paper we draw our inspiration primarily from a parallel line of work developed over the same period which has successfully combined the Z formalism [17,20] with CSP to develop the powerful concurrent language for refinement called *Circus* [21,22,11].

In Event-B the behaviour of a system is not represented explicitly, but rather is indirectly encoded in terms of declared behavioural state which the system's events have to test to determine their *enabledness*, and update when they *fire* to influence the further behaviour of the system. While such an approach can be perfectly general in its expressivity it doesn't greatly engender behavioural transparency, so that Event-B specifications can often be relatively opaque from a behavioural perspective. On the other hand CSP||B and csp2B both reconcile state manipulation with concurrent interaction by identifying each state-updating operation with its own individual corresponding concurrent interaction event, that event typically either sharing the same name as the operation concerned or at

least acquiring a name which is uniquely associated with the operation, so that the behavioural characteristics of the system can then be explicitly described in CSP. This certainly yields greater behavioural transparency, but there is a price to be paid in terms of the relative inflexibility during a system's development, in that the system's concurrent architecture must essentially be determined at the outset of that development by its abstract specification.

Interestingly, *Circus* is relatively unusual among state-enriched concurrent specification languages in having no such one-to-one correspondence between operations and events. This gives it valuable flexibility as a concurrent development formalism, in that new events can be introduced at any refinement stage, allowing the concurrent architecture of an implementation to emerge during development rather than being set at the abstract specification stage [4]. The price to be paid for this is a significantly more complicated semantic reconciliation of two quite different sorts of actions –namely, *Z*-described state-manipulation operations and CSP-described behavioural agents– which is necessary when these are combined within the body of a higher-level *Circus* action, since such an operation cannot simply be identified with an event with which it shares its name. The designers of *Circus* have met this challenge by interpreting both its operations and its actions in a uniform way as binary relations in the Unified Theories of Programming (UTP) [7]. In this way the UTP provides a firm semantic foundation for *Circus*.

The starting point for the work described in this paper was our speculation that what *Circus* has already done for *Z* by incorporating it in a wider language with explicit behaviourally expressive features, ought to be equally possible for *B*. We therefore sought an effective means by which *B* might be endowed with a corresponding set of features, ideally without compromising any of its existing conceptual integrity and clarity. The result is our syntactically modest but conceptually significant extension of classical *B* providing a recasting of *Circus* within a *B* setting, which we call reactive *B* (or just *rB* for brevity).

One might reasonably ask why we have chosen classical *B* rather than Event-*B* as the basis for our reactive extension. The reason is that, in the context of our extended *B* language, we want each *B* abstract machine play the passive role of an encapsulated data type incorporated within a separately-described process, rather than an active role as a behavioural entity in its own right which communicates with its environment via shared events. Specifically, our *B* abstract machines need to offer operations with input and output parameters, and with a contractual interpretation which implies, *inter alia*, that those operations must be called within their preconditions if divergence is to be avoided. Currently, only classical *B* provides these features.

The rest of the paper is organised as follows. In section 2 we preview our *rB* by means of a small illustrative example. In section 3 we informally describe all the features of our new *rB* action notation used to specify *rB* processes. In section 4, in order further to illustrate the applicability of reactive *B*, we present two more reactive specifications which use more of those features. In section 5 we give an overview of the UTP and its application to reactive programs. In section 6 we

show how we apply the UTP to give our rB actions a precise relational semantics. In section 7 we present a more extensive illustrative example to demonstrate the versatility of reactive B. In section 8 we conclude by reviewing what we have done so far, and then looking ahead to chart how we believe further development of reactive B might proceed.

2 A Reactive-B Preview

Our reactive B extends classical B simply by introducing a new structural component called a *Process* alongside the existing Abstract Machine Specification, Refinement, and Implementation components of classical B. A process has a set of named communication channels declared in a **CHANNELS** clause via which values of designated type can be communicated with its environment. It also optionally **INCLUDES** one or more ordinary abstract machines by which it acquires its persistent state. A process will generally also possess a number of named subsidiary actions, defined via its **SUBACTIONS** clause, and specified in our rB action notation. The sole use of these subsidiary actions is in the definition of the process's (unnamed) main action, which appears in its **MAINACTION** clause.

It is important to appreciate that the meaning of a reactive-B process rests entirely in its pattern of communication with its environment via its declared channels, and nothing else. In particular, its persistent state acquired via its included machines is entirely encapsulated within it and serves only to influence its pattern of channel communications as expressed by its main action and associated subsidiary actions. A process's persistent state is never visible to any part of its external environment. We will illustrate our new Process component by means of the example in the following subsection.

2.1 The Fibonacci Generator

This example is adapted from that of the same name in [22]. We specify a process which is required to communicate successive Fibonacci numbers to its environment via a channel called *out*. To do so we first specify our *Fibonacci* abstract machine whose state records the two most recent Fibonacci numbers output, and whose sole operation *outfibstate* generates, records and outputs the next number in the sequence:

```

MACHINE   Fibonacci
VARIABLES  xx , yy
INVARIANT   $xx \in \mathbb{N} \wedge yy \in \mathbb{N}$ 
INITIALISATION   $xx , yy := 1 , 1$ 
OPERATIONS
   $next \leftarrow \text{outfibstate} \hat{=} xx , yy , next := yy , xx + yy , xx + yy$ 
END

```

We now specify our *Fibproc* process to incorporate the *Fibonacci* abstract machine and communicate successive numbers of the Fibonacci sequence as required on channel out:

```

PROCESS Fibproc
INCLUDES Fibonacci
CHANNELS out :  $\mathbb{N}$ 
SUBACTIONS
  InitFib  $\hat{=}$  out .1  $\rightarrow$  out .1  $\rightarrow$  Fibonacci_init ;
  OutFib  $\hat{=}$   $\mu X$  . VAR nn IN nn  $\leftarrow$  outfibstate ; out .nn  $\rightarrow$  X END
MAINACTION
  InitFib ; OutFib
END

```

Trivial as it is, our Fibonacci example nevertheless usefully illustrates a number of features of our rB action notation:

1. Channel names appear in roman font, as in channel out of our *Fibproc* process above, to distinguish them from variable and operation names, which always appear in *italic* font.
2. Every B abstract machine has an intrinsic initialisation operation. The actions of an rB process which includes an abstract machine *M* can initialise the included machine at any time by invoking its intrinsic initialisation operation as *M_init*. We see, for example, that our *Fibonacci* machine's initialisation is invoked by our *Fibproc* process's InitFib subaction by the invocation *Fibonacci_init*.
3. As well as the persistent state embodied by the state variables *xx* and *yy* which our *Fibproc* process acquires from its included *Fibonacci* machine, individual actions can also declare their own local variables, as in the case of *nn* for subaction OutFib of our *Fibproc* process.
4. Actions can invoke operations of an included machine as well as communicate values via channels to the environment, again as seen in subaction OutFib of our *Fibproc* process.
5. Actions can be sequentially composed, as in the main action of our *Fibproc* process, and also in the body of subaction OutFib.
6. Actions can be recursively defined, again as seen in subaction OutFib of our *Fibproc* process. Here the μ signals a recursive definition. OutFib invokes operation *outfibstate* of the included *Fibonacci* machine to obtain the next Fibonacci number which it holds temporarily in local variable *nn* before communicating it to the environment via channel out and then repeating this same pattern of behaviour indefinitely.

It is also convenient to note here that when an rB process **INCLUDES** an abstract machine, then B's usual *semi-hiding* encapsulation principle applies. That is to say, the actions of the process may passively refer directly to the state variables of the included abstract machine, but can modify these only by invoking the operations of the included machine.

3 The rB Action Notation

In the following, A and B denote actions, b denotes a condition on the variables currently in scope, ch denotes a channel and xx and yy denote (lists of) fresh variables:

- **basic actions:** $SKIP$ is the action which terminates immediately without engaging in any channel communication or modifying its process’s encapsulated state. $STOP$ is the action which deadlocks immediately without engaging in any channel communication or modifying its process’s encapsulated state.
- **prefixed action:** $ch.exp \rightarrow A$ communicates the value of expression exp on channel ch and then behaves as action A . If ch is a typeless channel then the expression exp and its associated “.” are omitted, and the communication concerned is known as a simple synchronisation event. If the type of ch is composite, *i.e.* of a form such as $T_1 * T_2$, and exp_1 and exp_2 are expressions respectively of types T_1 and T_2 , then “ $ch.(exp_1, exp_2)$ ” can be alternatively written as “ $ch.exp_1.exp_2$ ”. The prefix combinator “ \rightarrow ” binds strongest of all the combinators of our rB action notation.
- **external choice:** $A \square B$ offers the environment any of the initial communications offered by either A or B . If the environment chooses an initial communication offered uniquely by A the subsequent behaviour of the composite action will be that of A . Similarly, if the environment chooses an initial communication offered uniquely by B the subsequent behaviour of the composite action will be that of B . However, should the environment choose an initial communication which is offered by both A and B the subsequent behaviour of the composite action will be nondeterministically chosen at that point to be either that of A or B .
- **recursion:** $\mu X . action_exp(X)$, where X is a fresh identifier and $action_exp(X)$ is an action expression parameterised by the action X and built from our various rB action combinators, denotes the (refinement-)least fixed point X of $action_exp(X)$. In other words, it denotes the least-deterministic action X for which $X = action_exp(X)$ ¹. The recursion combinator “.” binds weakest of all the combinators of our rB action notation; this means that the body of recursive μ -expression extends as far as possible to the right.
- **operation invocation:** The action comprising the operation invocation $yy \leftarrow op(exp)$ engages in no channel communication but modifies its process’s persistent state via the operation op of its process’s included B machine. Here exp is an expression whose value is passed to the operation as its input parameter value. The output from the operation is received by yy which must be a local variable currently in scope. If the operation’s signature

¹ The existence of such a least fixed point is guaranteed under Tarski’s theorem [18] by the monotonicity of all the rB action combinators with respect to refinement.

has no input parameter the expression exp and its enclosing parentheses are omitted. Similarly, if the operation's signature has no output the receiving variable yy and its associated " \leftarrow " are omitted. An operation invoked outside its precondition will diverge.

- **sequential composition:** The sequential composition $A ; B$ of actions A and B behaves like A until this terminates (if indeed it ever does) and thereafter like B . If A never terminates then $A ; B$ simply behaves like A . The sequential composition operator " $;$ " binds more strongly than the external choice operator " \square ".
- **local variable:** $\text{VAR } xx \text{ IN } A \text{ END}$ introduces the fresh local variable(s) xx whose scope is limited to A and whose type is determined by A .
- **guarded action:** $\text{WHEN } b \text{ THEN } A \text{ END}$ behaves as A when condition b is true and otherwise as STOP . It corresponds to the guarded *Circus* action $b \ \& \ A$.
- **conditional:** $\text{IF } b \text{ THEN } A \text{ ELSE } B \text{ END}$ behaves as A if b is true and otherwise as B . It corresponds to the compound *Circus* action $b \ \& \ A \ \square \ \neg b \ \& \ B$.
- **short conditional:** $\text{IF } b \text{ THEN } A \text{ END}$ behaves as A if b is true and otherwise as SKIP . It corresponds to the compound *Circus* action $b \ \& \ A \ \square \ \neg b \ \& \ \text{SKIP}$.
- **prefix choice:** $\text{ACCEPT } xx \ \text{WHERE } b \ \text{FROM } ch \ \text{THEN } A \ \text{END}$ introduces the fresh local variable(s) xx and is willing to accept from the environment via channel ch the communication of any value for xx which satisfies condition b , after which it behaves as A . Thus the type of xx is determined by the channel type of ch . The construct corresponds to the *Circus* prefix choice action $ch?xx : b \rightarrow A(xx)$. The structure of an ACCEPT clause makes the scope of the local variable(s) xx explicit by restricting this to the condition b and the action A . The " $\text{WHERE } b$ " subclause is optional. An ACCEPT clause without a WHERE subclause is equivalent to one with a " WHERE true " subclause.

4 Two More Reactive-B Specifications

In this section we give two further example rB specifications to illustrate the use of more of the rB action combinators described in the preceding section. The first is an obligatory vending machine example for concurrency traditionalists, while the second specifies the behaviour of a fuel pump.

4.1 The Vending Machine

The *Vending* abstract machine keeps track of the current level of stock held in the actual physical machine, and the unspent credit the customer has currently accumulated by inserting coins:


```

MACHINE  Vending
CONSTANTS  capacity , itemprice
PROPERTIES  capacity  $\in \mathbb{N} \wedge$  itemprice  $\in \mathbb{N}_1$ 
VARIABLES  credit , stock
INVARIANT  credit  $\in \mathbb{N} \wedge$  stock  $\in \mathbb{N}$ 
INITIALISATION  credit , stock := 0 , capacity
OPERATIONS
  items , credleft  $\leftarrow$  dispense  $\hat{=}$ 
    ANY tt WHERE tt  $\in 1 .. ($  credit / itemprice  $)$ 
    THEN items , credleft := tt , credit - (tt  $\times$  itemprice ) END ;
  reset ( cc , ss )  $\hat{=}$  PRE cc  $\in \mathbb{N} \wedge$  ss  $\in \mathbb{N}$  THEN credit , stock := cc , ss END ;
  clearcredit  $\hat{=}$  credit := 0 ;
  addcredit ( nn )  $\hat{=}$  PRE nn  $\in \mathbb{N}$  THEN credit := credit + nn END
END

```

The *Vendingproc* process includes the *Vending* abstract machine, and communicates with its environment –the actual physical machine– to be apprised of the customer’s insertions of coins and pressing of the dispense and returnchange buttons, and to instruct the physical machine how many items to dispense and how much change to give. We note that its **CHANNELS** clause declares four typeless channels as well as two typed ones:

```

PROCESS  Vendingproc
INCLUDES  Vending
CHANNELS
  giveitems :  $\mathbb{N}$  ; givechange :  $\mathbb{N}$  ; insert5p ; insert10p ; dispensebtn ; changebtn
SUBACTIONS
  DispenseItm  $\hat{=}$  dispensebtn  $\rightarrow$ 
    VAR nn , cr IN
      nn , cr  $\leftarrow$  dispense ;
      IF nn : 0..stock THEN giveitems.nn  $\rightarrow$  reset ( cr , stock - nn ) END
    END ;
  RtnChange  $\hat{=}$  changebtn  $\rightarrow$ 
    IF credit > 0 THEN givechange.credit  $\rightarrow$  clearcredit END ;
  InsertMoney  $\hat{=}$  insert5p  $\rightarrow$  addcredit ( 5 )  $\square$  insert10p  $\rightarrow$  addcredit ( 10 )
MAINACTION
  Vending_init ;  $\mu X . ($  InsertMoney  $\square$  DispenseItm  $\square$  RtnChange  $) ; X$ 
END

```

In the *Vendingproc* process above we note how B’s encapsulation semi-hiding principle allows subactions *DispenseItm* and *RtnChange* to refer passively to state variables *stock* and *credit* respectively of the included *Vending* abstract machine.

4.2 The Fuel Pump

This example is adapted from that of the same name in [11]. It models the activities associated with a fuel-dispensing pump in a garage or filling station.

The *Fuelpump* abstract machine records the amount of fuel currently held in the pump's storage tank.

MACHINE *Fuelpump*

VARIABLES *fuel*

INVARIANT $fuel \in \mathbb{N}$

INITIALISATION $fuel := 5000$

OPERATIONS

reload (tt) $\hat{=}$ **PRE** $tt \in \mathbb{N}$ **THEN** $fuel := fuel + tt$ **END** ;

supply (tt) $\hat{=}$ **PRE** $tt \in 1 .. fuel$ **THEN** $fuel := fuel - tt$ **END**

END

The *Pumpproc* process communicates with the physical fuel pump to be apprised of how much fuel is reloaded into the pump's storage tank in each reloading episode, and how much fuel is dispensed by the pump in each dispensing episode. Its **CHANNELS** clause declares five typeless channels as well as two typed ones:

PROCESS *Pumpproc*

INCLUDES *Fuelpump*

CHANNELS *init* ; *liftnozzle* ; *replacenozzle* ; *squeezetrigger* ; *releasetrigger* ;
 $reload : \mathbb{N}$; $enteramount : \mathbb{N}$

SUBACTIONS

PumpIdle $\hat{=}$

liftnozzle \rightarrow **PumpActive** \square

ACCEPT qq **FROM** *reload* **THEN** *reload*(qq) **END** \square

init \rightarrow *Fuelpump_init* ;

PumpActive $\hat{=}$

replacenozzle \rightarrow **SKIP** \square

ACCEPT qq **WHERE** $qq \in 1..fuel$ **FROM** *enteramount*

THEN *squeezetrigger* \rightarrow *supply*(qq) ; *releasetrigger* \rightarrow **SKIP** **END**

MAINACTION

Fuelpump_init ; $\mu X .$ **PumpIdle** ; *X*

END

We note that our *Pumpproc* above is our first example of a process whose actions use our action notation's prefix-choice construct **ACCEPT...END**.

5 Unifying Theories of Programming

We follow *Circus* in adopting Hoare and He's Unifying Theories of Programming (UTP) [7] as an appropriate semantic foundation in terms of which our rB action constructs can be given formal meanings. We note that this whole section,

which is based essentially on chapters 3 and 8 of [7], is merely an exposition for the convenience of the unacquainted reader of the part of UTP which we will subsequently use in section 6 to give a UTP semantics to our rB actions. None of the current section therefore represents any original work of the current authors.

UTP in fact provides a common semantic framework, based on alphabetised binary relations, which encompasses many different programming paradigms. The two basic unifying principles of the UTP approach are (1) that refinement is always modelled by logical implication, and (2) that sequential composition is always modelled by relational composition.

The UTP approach often supplements the regular state variables of a program by introducing further *auxiliary* variables to describe aspects of the observed behaviour of a program which are of particular interest. Unlike the regular variables, such auxiliary variables cannot be directly referenced or manipulated by the program itself. The undashed versions of the regular state variables and these auxiliary variables together record initial observations of the program before execution has occurred, while their dashed counterparts record a corresponding observation of a stable state reached upon execution. In the case of a simple sequential program the only subsequent stable state of interest is the program's final state, but in the case of others, such as reactive programs, there may be many observable intermediate states to be recorded too.

Since in UTP a program is always regarded as being executed in sequence after its immediate predecessor, the current program's initial (undashed) variables always have to match the values of its predecessor's final (dashed) ones. A program's undashed variables can therefore be regarded as recording the state it has inherited from its predecessor.

5.1 Relational Semantics of Sequential Programs

The externally observable behaviour of a simple sequential program on a state space characterised by a list of program variables v is expressed in UTP by the relational predicate $ok \wedge p \Rightarrow ok' \wedge q$ over the alphabet of variables (v, ok, v', ok') , where ok and ok' are boolean variables. Here the undashed variables v represent the regular variables' initial state while their dashed counterparts v' represent the same regular variables' final state, p and q being subsidiary predicates respectively over v and (v, v') . But even a correct program can behave pathologically if wrongly used; the propensity of a sequential program to *diverge* by aborting or failing to terminate is reflected by the auxiliary ok variable which when initially true ($ok = \text{true}$) signifies that this program's predecessor successfully terminated, so allowing it to start, and when finally true ($ok' = \text{true}$) signifies that this program subsequently successfully terminated. An observation involving $ok' = \text{false}$, on the other hand, signifies that the program diverged rather than terminated successfully, for example by becoming trapped in an infinite loop, or by provoking a system error such as a memory address violation and so having to be aborted.

The program's relational predicate $ok \wedge p \Rightarrow ok' \wedge q$ is traditionally abbreviated in UTP to $p \vdash q$, which can be interpreted as saying *if the program starts in an initial state satisfying p , it will terminate in a final state satisfying q .*

5.2 Relational Semantics of Reactive Programs

Unlike that of a simple sequential program, the observable behaviour of a reactive program such as a CSP process will typically include many intermediate interactions with its environment between the start and ultimate end of its execution. Indeed, a reactive program may never reach any ultimate end of its execution. A CSP process has not necessarily diverged merely because it never terminates. For example, the CSP process $\mu X . a \rightarrow X$ engages an endless succession of ' a 's but doesn't diverge. Equally, the primitive CSP process STOP immediately reaches a stable intermediate state where it refuses any interaction at all, and being permanently blocked it neither terminates nor diverges. To model the behaviour of a CSP process we augment ok with the three further auxiliary variables $wait$, tr and ref . The roles of these auxiliary variables are as follows:

- ok is a boolean which in its undashed form indicates whether the current process's predecessor diverged ($ok = \text{false}$) or stabilised ($ok = \text{true}$). Correspondingly, in its dashed form ok' it indicates whether the current process has stabilised ($ok' = \text{true}$), or diverged ($ok' = \text{false}$). The values of the remaining dashed variables other than tr' are significant only when $ok' = \text{true}$.
- $wait$ is a boolean which in its undashed form indicates, when ok holds, whether the current process's predecessor had terminated ($wait = \text{false}$), so allowing the current process to start, or merely reached an intermediate state ($wait = \text{true}$), so inhibiting the current process from exhibiting any behaviour yet. Similarly, in its dashed form $wait'$ it indicates, providing ok' holds, whether the process reached a stable intermediate state awaiting interaction with its environment ($wait' = \text{true}$), or has terminated ($wait' = \text{false}$).
- tr in its undashed form records the trace of events which have already occurred before the start of execution of the current process. On the other hand in its dashed form tr' it records the trace of events which have occurred before and during its execution up to its subsequent observed state. Therefore tr and tr' are of type Σ^* , *i.e.* finite sequences of events of the event alphabet Σ . We note that tr is always a prefix of tr' , which we write as $tr \leq tr'$, and that the trace of events actually engaged in by the current process itself is the trace difference of tr' and tr , which we write as $tr' - tr$. Our trace variables tr and tr' share the unusual distinction that their values are significant even when ok and ok' don't hold. This is because they encode history which cannot be retrospectively altered even by divergence.
- ref in its undashed form records a set of events observed to be refused by the current process's predecessor, and as such is only significant if that

predecessor is in a stable intermediate state, *i.e.* one where *ok* and *wait* are both true. More importantly, in its dashed form *ref'* it records a set of events observed to be refused by the current process in a stable intermediate state, *i.e.* one where *ok'* and *wait'* are both true. As sets of events *ref* and *ref'* are of type $\mathbb{P}\Sigma$, where Σ is the process's alphabet of events. It is important to appreciate that the value of *ref'* in any particular observation can be *any* set of events refused by the process, and does not therefore necessarily comprise *all* the events which the process is disposed to refuse in that particular intermediate state. In particular, the empty set $\{\}$ is always a valid refusal set.

5.3 Reactive Healthiness Conditions

A UTP binary relation characterising any CSP process must comply with a number of healthiness conditions. Among these are the three reactive healthiness conditions R_1 , R_2 and R_3 :

- R_1 ensures that a process's history of interactions with its environment can never be retrospectively altered but only extended: this means that if A is any CSP process, then it has to be the case that $A = A \wedge tr \leq tr'$, where \leq denotes the prefix ordering on traces.
- R_2 ensures that the behaviour of a process is not affected by the history of interactions of its predecessors. So even if we eliminate that history altogether by replacing tr by $\langle \rangle$ and tr' by the trace difference $tr' - tr$ this cannot affect the behaviour of the current process: that is, if A is any CSP process then it must be the case that $A = A[\langle \rangle, tr' - tr / tr, tr']$.
- R_3 ensures that relational composition correctly models actual sequential composition of CSP processes. It does this by ensuring that in any relational composition $P ; Q$ of process relations P and Q the latter makes no contribution to the behaviour of the composition while P is only in a waiting state. So, when restricted to P 's intermediate waiting states, Q must act like the CSP identity and change nothing: this means that if A is any CSP process we must have that $A = II_{\text{CSP}} \triangleleft wait \triangleright A$ ², where II_{CSP} is the CSP reactive identity, which is defined by

$$II_{\text{CSP}} =_{df} (\neg ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v)$$

where v represents the regular state variables.

R_1 , R_2 and R_3 can also be regarded as *healthifiers* or functions which when applied to arbitrary relations over a state characterised by variables *wait*, *tr*, *ref*, v yield corresponding appropriately healthy ones. Thus we have

$$\begin{aligned} R_1(A) &= A \wedge tr \leq tr' \\ R_2(A) &= A[\langle \rangle, tr' - tr / tr, tr'] \\ R_3(A) &= II_{\text{CSP}} \triangleleft wait \triangleright A \end{aligned}$$

² The *conditional* construct $P \triangleleft b \triangleright Q$ is an abbreviation for $(b \wedge P) \vee (\neg b \wedge Q)$.

Each of these three healthifiers is idempotent, since once a relation is R_n -healthy it can't be made any more so. It also turns out that they are commutative, in the sense that irrespective of in what order they are applied to a relation they will yield the same R -healthy relation. We therefore define our composite reactive healthifier R as $R_1 \circ R_2 \circ R_3$.

A *reactive design* is simply an ordinary design over a state characterised by variables $wait, tr, ref, v$ which has been reactively healthified by the application of our composite reactive healthifier R . Thus it is a relation of the form $R(p \vdash q)$ where p and q are predicates over variables $wait, tr, ref, v$ and their dashed counterparts. In the next section we will give each of our rB action constructs a meaning as such a reactive design.

6 Semantics of rB Actions

All our rB actions can be given formal meaning as reactive designs. The way we do so essentially follows that of Oliveira *et al.* in [11] for their corresponding *Circus* actions. To illustrate the technique we give the semantic definitions of some of our rB action constructs in this section. Those of the rest can be inferred from the corresponding *Circus* action definitions in [11].

Basic actions

$$\begin{aligned} \text{SKIP} &=_{df} R(\text{true} \vdash tr' = tr \wedge \neg wait') \\ \text{STOP} &=_{df} R(\text{true} \vdash tr' = tr \wedge wait') \end{aligned}$$

Prefixed action

$$\text{ch.exp} \rightarrow A =_{df} (\text{ch.exp} \rightarrow \text{SKIP}) ; A$$

where

$$\text{ch.exp} \rightarrow \text{SKIP} =_{df} R(\text{true} \vdash \text{do}(\text{ch.exp}) \wedge v' = v)$$

where

$$\text{do}(\text{ch.exp}) =_{df} tr' = tr \wedge \text{ch.exp} \in ref' \triangleleft wait' \triangleright tr' = tr \wedge \langle \text{ch.exp} \rangle$$

Operation invocation: Given the actual operation invocation instance $yy \leftarrow op(exp)$ within an rB action, let the generalised substitution S be the instantiated body of op whose formal input parameter has been replaced by the expression exp and whose formal output parameter has been replaced by the variable yy . Also, let u represent the current state variables, *i.e.* the persistent variables of the containing process together with any action local variables currently in scope. Now let S_u be the frame-enlarged variant of S whose write frame has been extended by u , and let $\text{prd}(S_u)$ be the before-after predicate of S_u and let $\text{trm}(S)$ be the termination predicate of S , all as defined in [5]. Then we interpret $yy \leftarrow op(exp)$ as a reactive design in this way:

$$yy \leftarrow op(exp) =_{df} R(\text{trm}(S) \vdash \text{prd}(S_u) \wedge tr' = tr \wedge \neg wait')$$

Sequential composition: In accordance with UTP principles, the sequential composition $A ; B$ of actions A and B is defined simply as the relational composition of their respective reactive designs.

Local variable:

$$\text{VAR } xx \text{ IN } A \text{ END} \quad =_{df} \quad \exists xx, xx' . A$$

Guarded action:

$$\text{WHEN } b \text{ THEN } A \text{ END} \quad =_{df} \quad R(b \Rightarrow \neg A_t^f \vdash (b \wedge A_t^t) \vee (\neg b \wedge tr' = tr \wedge wait'))$$

where A_t^f is an abbreviation for $A[\text{true}, \text{false}, \text{false} / \text{ok}, \text{wait}, \text{ok}']$ and A_t^t is an abbreviation for $A[\text{true}, \text{false}, \text{true} / \text{ok}, \text{wait}, \text{ok}']$.

7 The Maritime Port

Our final example is adapted from the corresponding CSP||B one in [16], so demonstrating that rB's applicability is not confined to examples from the *Circus* literature alone. It concerns the directing of ships visiting a busy port to discharge their cargoes and/or take on new cargoes. When a ship arrives it is first directed to join the end of the queue of ships waiting to dock at a quay. When the waiting queue is non-empty and there are vacant quays, the ship at the head of the queue may be directed to leave the queue and dock at one of the vacant quays. At any time a ship currently occupying a quay may vacate that quay and depart from the port.

The *PortGlobals* machine introduces sets and constants for the Port system:

```

MACHINE   PortGlobals
SETS     SHIP ; QUAY
CONSTANTS Quays
PROPERTIES Quays ∈ ℱ ( QUAY )
END

```

The *Waiting* machine models the queue of ships waiting to dock:

```

MACHINE   Waiting
SEES     PortGlobals
VARIABLES waitingsq
INVARIANT waitingsq ∈ iseq ( SHIP )
INITIALISATION waitingsq := [ ]
OPERATIONS
  joinqueue ( ss ) ≐ PRE ss ∈ SHIP – WaitingShips
    THEN waitingsq := waitingsq ← ss END ;

```

```

ss ← leavequeue ≐ PRE WaitingShips ≠ {}
    THEN ss , waitingsq := first ( waitingsq ) , tail ( waitingsq ) END

```

DEFINITIONS

```
WaitingShips ≐ ran ( waitingsq )
```

END

The *Docked* machine models the ships currently docked at the quays:

MACHINE *Docked*

SEES *PortGlobals*

VARIABLES *docked*

INVARIANT *docked* ∈ *Quays* ↔ *SHIP*

INITIALISATION *docked* := {}

OPERATIONS

```
qq ← dock ( ss ) ≐
```

```
PRE ss ∈ SHIP - DockedShips ∧ Vacant ≠ {} THEN
```

```
ANY qu WHERE qu ∈ Vacant THEN qq := qu || docked ( qu ) := ss END
```

```
END ;
```

```
leave ( ss ) ≐ PRE ss ∈ DockedShips THEN docked := docked ▷ { ss } END
```

DEFINITIONS

```
DockedShips ≐ ran ( docked ) ; Vacant ≐ Quays - dom ( docked )
```

END

The *Port* machine combines the *Waiting* and *Docked* machines:

MACHINE *Port*

INCLUDES *PortGlobals* , *Waiting* , *Docked*

PROMOTES *leave*

INVARIANT *WaitingShips* ∩ *DockedShips* = {}

OPERATIONS

```
arrive ( ss ) ≐ PRE ss ∈ SHIP - ShipsInPort THEN joinqueue ( ss ) END ;
```

```
ss , qq ← transfer ≐ PRE WaitingShips ≠ {} ∧ Vacant ≠ {}
```

```
THEN ss ← leavequeue || qq ← dock ( first ( waitingsq ) ) END
```

DEFINITIONS

```
ShipsInPort ≐ WaitingShips ∪ DockedShips
```

END

Having specified our various abstract machines for the port system we can now specify our *Portproc* process which directly includes the top-level *Port* machine, and therefore indirectly incorporates the latter's included machines too. The *Portproc* process illustrates some rB action notation features not hitherto employed in our earlier examples, notably the guarded action construct WHEN...THEN...END, the WHERE subclause of the ACCEPT...END construct and the multiple dot notation for communicating tuple values on a channel of a complex type:

PROCESS *Portproc*

INCLUDES *Port* , *PortGlobals*

CHANNELS arrive : *SHIP* ; depart : *SHIP* ; dock : *SHIP* × *QUAY*

SUBACTIONS**Arrive** $\hat{=}$

ACCEPT *sh* **WHERE** $sh \in SHIP - ShipsInPort$ **FROM** arrive
THEN *arrive*(*sh*) **END** ;

Leave $\hat{=}$

ACCEPT *sh* **WHERE** $sh \in DockedShips$ **FROM** depart
THEN *leave*(*sh*) **END** ;

Dock $\hat{=}$

WHEN $WaitingShips \neq \{\}$ \wedge $Vacant \neq \{\}$ **THEN**
VAR *sh* , *qu* **IN** $sh, qu \leftarrow transfer$; *dock.sh.qu* \rightarrow **SKIP** **END**
END

MAINACTION

Port_init ; $\mu X . (Arrive \square Dock \square Leave)$; *X*

END

We see from our *Portproc* process that the system it specifies has to ensure that ships arriving at the port must queue in an orderly fashion awaiting a vacant quay, and depart from the port only after docking at a quay. The system must also decide at which vacant quay the ship at the head of the queue is to be directed to dock. Naturally, the system must be able to handle all these events concurrently since the order in which ships arrive and depart is unpredictable.

8 Conclusion and Future Work

We have presented our extension of classical B called reactive B, and have demonstrated its expressivity by applying it to several examples from the literature of combining state and behavioural formalisms. We have explained how the actions of a reactive-B process can be given a similar UTP relational semantics to that given to *Circus* actions.

Of course, there is a great deal still to be addressed. Foremost, perhaps, there is the important issue of tool support, so vital for the industrial level of robustness and scalability to which we should aspire for any B-related method. Closely associated with this is the degree of mechanisation in type- and consistency-checking to which our reactive B lends itself. Checking our rB processes for freedom from divergence is straightforward since the only source of divergence in our relatively limited action notation is the invoking of an operation outside its precondition, and this is as amenable to static checking in rB as it is in ordinary classical B. Granted, the presence of recursion in the action definitions of rB raises an extra challenge, as compared with classical B. Here, however, we can capitalise on the work of the inventors of CSP||B, who have already encountered the same challenge and solved it by their *control loop invariant* technique described in [19].

Another vital issue is that of mechanised support for refinement. But here we are confident we can borrow from the work of the *Circus* developers who have already amassed an extensive arsenal of refinement rules for their own language, as described in [9], [10] and [11]. Since we have given our rB actions a similar UTP relational semantics to that of *Circus* actions, many of these *Circus* refinement rules should be readily adaptable for rB.

Yet another important topic is the obvious one of how rB processes might themselves be usefully combined to form composite processes. This in its turn raises a whole range of interesting associated issues such interleaving, parallel composition and the hiding of those communications which are only relevant between the processes being combined but not for the wider environment beyond them. But again all these issues have already been comprehensively addressed in *Circus*, which gives us confidence that we can successfully treat them in a similar way for rB.

Acknowledgements

We are grateful for the comments of the anonymous reviewers of the original draft of this paper.

References

1. Abrial, J.-R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge (1996)
2. Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.): ZB 2002. LNCS, vol. 2272. Springer, Heidelberg (2002)
3. Butler, M.J.: csp2B: A practical approach to combining CSP and B. *Formal Asp. Comput.* 12(3), 182–198 (2000)
4. Cavalcanti, A., Sampaio, A., Woodcock, J.: A refinement strategy for *Circus*. *Form. Asp. Comput.* 15(2-3), 146–181 (2003)
5. Dunne, S.E.: A theory of generalised substitutions. In: Bert, et al. (eds.) [2], pp. 270–290
6. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (1985)
7. Hoare, C.A.R., Jifeng, H.: *Unifying Theories of Programming*. Prentice Hall, Englewood Cliffs (1998)
8. Metayer, C., Abrial, J.-R., Voisin, L.: Event-B Language. Technical Report 3.2, Rodin Project (2005), <http://rodin.cs.ncl.ac.uk>
9. Oliveira, M.: *Formal Derivation of State-rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York (2005); YCST-2006/02
10. Oliveira, M., Cavalcanti, A., Woodcock, J.C.P.: A denotational semantics for *Circus*. *Electron. Notes Theor. Comput. Sci.* 187, 107–123 (2007)
11. Oliveira, M., Cavalcanti, A., Woodcock, J.C.P.: A UTP semantics for *Circus*. *Form. Asp. Comput.* 21(1-2), 3–32 (2009)
12. Roscoe, A.W.: *The Theory and Practice of Concurrency*. Prentice Hall, Englewood Cliffs (1998)

13. Schneider, S.A.: Concurrent and Real-time Systems: The CSP Approach. Wiley, Chichester (2000)
14. Schneider, S.A., Treharne, H.E.: Communicating B machines. In: Bert, et al. (eds.) [2], pp. 416–435
15. Schneider, S.A., Treharne, H.E.: CSP theorems for communicating B machines. Formal Asp. Comput. 17(4), 390–422 (2005)
16. Schneider, S.A., Treharne, H.E., Evans, N.: Chunks: Component verification in CSP||B. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 89–108. Springer, Heidelberg (2005)
17. Spivey, J.M.: The Z Notation: a Reference Manual, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
18. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5(2), 285–309 (1955)
19. Treharne, H.E., Schneider, S.A.: How to drive a B machine. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) ZB 2000. LNCS, vol. 1878, pp. 188–208. Springer, Heidelberg (2000)
20. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall, Englewood Cliffs (1996)
21. Woodcock, J.C.P., Cavalcanti, A.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) Proceedings of the 5th Irish Workshop in Formal Methods, IWFM 2001, Workshops in Computing, British Computer Society (2001), <http://ewic.bcs.org/conferences/2001/5thformal/papers>
22. Woodcock, J.C.P., Cavalcanti, A.: The semantics of *Circus*. In: Bert, et al. (eds.) [2], pp. 184–203

Event-B Decomposition for Parallel Programs*

Thai Son Hoang and Jean-Raymond Abrial

Department of Computer Science,
Swiss Federal Institute of Technology Zurich (ETH-Zurich),
CH-8092, Zurich, Switzerland
htson@inf.ethz.ch, jrabrial@neuf.fr

Abstract. We present here a case study developing a parallel program. The approach that we use combines *refinement* and *decomposition* techniques. This involves in the first step to abstractly specify the aim of the program, then subsequently introduce shared information between sub-processes via refinement. Afterwards, decomposition is applied to split the resulting model into sub-models for different processes. These sub-models are later independently developed using refinement. Our approach aids the understanding of parallel programs and reduces the complexity in their proofs of correctness.

Keywords: Event-B, parallel programs, decomposition, refinement.

1 Introduction

We consider here programs that use several co-operating parallel processes in order to compute the intended final result. Proving correctness of such programs is a difficult task because of the interleaved execution of many sub-statements from different processes. These sub-statements may be executed in an unpredictable order. As a result, techniques such as program testing do not give us sufficient confidence about the correctness of these programs, since no execution leading to an error might appear during tests. To achieve correctness, it is therefore necessary to develop these programs and prove them formally.

There are a number of methods for proving the correctness of parallel programs [1]. Our main contribution is an approach applying the technique of refinement and decomposition in Event-B [2], which reduce the complexity of the verification process (more information in Section 5.1). The approach contains four steps as follows.

1. Start with an abstract specification *in-one-shot* giving the purpose of the program.
2. Refine this abstract specification by introducing details about the *shared variables*.
3. Decompose the model in the previous step to split the model into several (abstract) sub-models for processes.
4. Refine each sub-model from the previous step independently.

In the last step, each sub-model can be seen as a new abstract specification and hence application of steps 2, 3 and 4 can be repeated again. The novelty of our approach is in

* Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (<http://www.deploy-project.eu/index.html>). We thank Matthias Schmalz, Christoph Sprenger and David Basin for their comments on drafts of this paper.

step 2 where we specify shared information between processes. This information has two purposes. Firstly, it contains the necessary guarantee condition from each process to establish the final result. Secondly, it also gives the condition on which each process can rely on in further development. This decision to have this step early in our development takes advantage of our decomposition technique and results in simpler models and reduces the complexity of proving programs. This is the main advantage of our method over existing approaches. More information on related work is in Section 5.1.

The rest of the paper is structured as follows. Section 2 gives an overview of the Event-B method and the concept of (shared variable) decomposition. Section 3 introduces the *FindP* program and its formal development using our approach is presented in Section 4. Section 5 compares our approach with some existing methods for developing parallel programs and draws some conclusions.

2 The Event-B Modelling Method

A development in Event-B [6] is a set of formal models. The models are built from expressions in a mathematical language, which are stored in a repository. When presenting our models, we will do so in a pretty-printed form e.g., adding keywords and following layout conventions to aid parsing. Event-B has a semantics based on transition systems and simulation between such systems, described in [3]. We will not describe in detail the Event-B semantics here and instead just illustrate some of the proof obligations that are important for our development.

Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types [6]. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. In the following, we further describe machines and machine refinement.

2.1 Machines

Machines specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by events. Each event is composed of a *guard* $G(t, v)$ (the conjunction of one or more predicates) and an *action* $S(t, v)$, where t are the *parameters* of the event.¹ The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the term “**any t where $G(t, v)$ then $S(t, v)$ end**”. We use the short form “**when $G(v)$ then $S(v)$ end**” when the event does not have any parameters, and we write “**begin $S(v)$ end**” when, in addition, the event’s guard equals *true*. A dedicated event of the last form is used for *initialisation*.

¹ When referring to variables v and parameters t , we usually allow for multiple variables and parameters, i.e., they may be “vectors”.

The action of an event is composed of one or more *assignments* of the form

$$x := E(t, v) \quad (1)$$

$$x \in E(t, v) \quad (2)$$

$$x :| Q(t, v, x') \quad , \quad (3)$$

where x is a variable contained in v , $E(t, v)$ is an expression, and $Q(t, v, x')$ is a predicate. Assignments of the form (1) are *deterministic*, whereas the other two forms are *nondeterministic*. In (2), x (which must be a single variable) is assigned an element of a set. In (3), Q is a “before-after predicate”, which relates the values x (before the action) and x' (afterwards). (3) is the most general form of assignment and nondeterministically selects an after-state x' satisfying Q and assigns it to x . Variables other than x are unchanged by the above assignments. There is also a side condition on the action of an event: the variables on the left-hand side of the assignments contained in the action must be disjoint.

Proof obligations serve to verify certain properties of machines. We only describe the proof obligation for invariant preservation. Formal definitions of all proof obligations are given in [3]. *Invariant preservation* states that invariants hold whenever variables change their values. Obviously, this does not hold a priori for any combination of events and invariants and therefore must be proved. For each event and each invariant, we must prove that the invariant is *re-established* after the event is carried out. More precisely, under the assumption of the invariants and the event’s guard, we must prove that the invariant still holds in any possible state after the event’s execution.

Similar proof obligations are associated with a machine’s initialisation event. The only difference is that there is no assumption that the invariant holds. For brevity, we do not treat initialisation differently from other events. The required modifications of the associated proof obligations are straightforward.

2.2 Machine Refinement

Machine refinement provides a means to introduce details about the dynamic properties of a model [6]. For more details on the theory of refinement, we refer to the Action System formalism [7], which has inspired the development of Event-B. Here we sketch some central proof obligations for machine refinement.

A machine CM can refine another machine AM . We call AM the *abstract* machine and CM the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec . Let the abstract event ea and concrete event ec be:

$$ea \hat{=} \mathbf{any } t \mathbf{ where } G(t, v) \mathbf{ then } S(t, v) \mathbf{ end} \quad (4)$$

$$ec \hat{=} \mathbf{any } u \mathbf{ where } H(u, w) \mathbf{ then } T(u, w) \mathbf{ end} \quad . \quad (5)$$

Somewhat simplified, we say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariant $J(v, w)$ establishes a simulation of ec by ea (*simulation*). Proving guard strengthening just amounts to proving

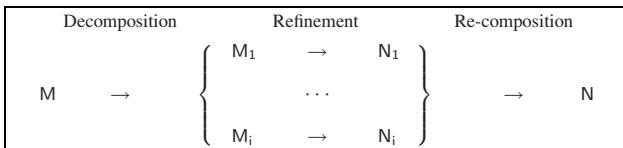
an implication. For simulation, we must prove that *ec* can be “simulated” by *ea*. More precisely, under the assumption of the invariants and of the concrete guard $H(u, w)$ we must show that it is possible to choose a value for the abstract parameter t such that the abstract guard holds and the gluing invariant $J(v, w)$ is re-established. The possible values for the abstract parameter are given as *witness* in *ec* with the keyword **with**.

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing. Moreover, it may be proved that new events do not collectively diverge. In other words, the new events cannot take control forever and hence one of the old events eventually occurs. We will not go into details for *convergent* proof obligation in this paper.

We have used the *Rodin Tool* [4] for our formal development. This is an industrial-strength tool for creating and analysing Event-B models. It includes a proof-obligation generator and support for interactive and semi-automated theorem proving.

2.3 Shared Variable Decomposition

The idea of decomposition is to split a large model into smaller sub-models which can be handled more comfortably than the whole: one should be able to refine these sub-models independently [2]. More precisely, if one starts from an initial (large) model, say M , decomposition allows us to split this model into several sub-models $M_1 \cdots M_i$. These sub-models can then be refined independently yielding $N_1 \cdots N_i$. The correctness of the decomposition technique guarantees that the model N , obtained by re-composing $N_1 \cdots N_i$, is a refinement of the original model M . This process is illustrated in the following diagram:



Generation of sub-models using shared variable decomposition: Given a model M with events $e_1(a)$, $e_2(a, c)$, $e_3(b, c)$, $e_4(b)$ we would like to decompose M into two separate models: M_1 dealing with events e_1 and e_2 ; and M_2 dealing with events e_3 and e_4 .

By giving the above *event partition*, we must also perform a *variable distribution*. This distribution can be derived directly from the information about the partitioning of events and the set of variables that they access. In our example, M_1 must have variables a and c , while M_2 must have variables b and c . As a result, c becomes a *shared variable* between the two models which *cannot be data-refined*. In contrast, the variables a and b are private variables of M_1 and M_2 and can be data-refined by their corresponding sub-refinements.

Moreover, in each sub-model, we need to have a number of *external events* to simulate how shared variables are handled in the non-decomposed model. These events are

² Note that the variables appeared in brackets denote those that are *accessed* by these events, e.g. appearing in guard or action of the corresponding event.

abstract versions of the corresponding internal events and use only the shared variables. In our example, M_1 will have an external event corresponding to e_3 (beside the internal events e_1 and e_2). Symmetrically, M_2 will have an external event corresponding to e_2 . Similar to shared variables, *external events* cannot be further refined.

We also present a practical construction of the external event given its original event. This is illustrated below for an external event $(\text{ext.})e_2$ in sub-model M_2 . Intuitively, this event is the *projection* of the original event, i.e. e_2 , on the state of the sub-model M_2 .

e_2 <pre> any t where $G(t, a, c)$ then $a, c : Q(t, a, c, a', c')$ end </pre>	$(\text{ext.})e_2$ <pre> any t, a where $G(t, a, c)$ then $c : \exists a' \cdot Q(t, a, c, a', c')$ end </pre>
---	---

3 Example: FindP Program

Our running example is a standard problem in the literature for parallel programs. The purpose of the *FindP* program is to find the first index k of an array *ARRAY* that satisfies some property P , if there is one. If this index does not exist, i.e. none of the array elements satisfy P , the program returns $M + 1$, where M is the size of the array.

We are interested in the solution using two parallel processes to independently investigate the array that was given by Rosen [20]. The processes in the original program works on the sets of even and odd indices separately. We present here a slightly generalised version of it where the two processes work on any two different parts of the array, denoted as *PART1* and *PART2*, which cover the entire domain of the array, and are not necessarily disjoint.

The main idea of each process is to independently evaluate the value of the array in ascending order and to publish the first value that it finds. Moreover, from time to time, a process looks at the value that is published by the other process in order to know if it needs to continue the search or if it can terminate early.

The pseudo-code for the main program is given below. Here *index1*, *index2* are the two local indices, and *publish1*, *publish2* are the published results of the processes. In the end, when both processes terminate, the result taken is the minimum of the two published results.

```

 $index1, index2 := \min(PART1), \min(PART2);$ 
 $publish1, publish2 := M + 1, M + 1;$ 
process1 || process2;
 $result := \min(\{publish1, publish2\})$ 

```

The pseudo-code for each process (presented here *process*₁) is as follows. Each process needs to continue only if its local index is smaller than both published results (as indicated by the guard of the loop). If this is the case, the process evaluates the value of the array at the current index and performs appropriate actions: publishing the current index or moving to the next index, if possible.

```

while  $index1 < \min(\{publish1, publish2\})$  do
  if  $ARRAY(index1) = \text{TRUE}$  then  $publish1 := index1$ 
  else  $index1 := \text{the-next-index-in-PART1-or-M+1}$  end
end

```


The key interaction between the two processes appears in the guard of the loop. Here the guard of $process_1$ refers to the published result of $process_2$, which in the meantime could be modified. In other words, $process_1$ needs to read the published value of $process_2$ into some local variable before making the comparison using this local variable. The unfolded version of the $process_1$ is as follows. Our formal development in later sections is guided towards this version of the processes.

```

1 : (read)  read1 := publish2;
2 :        if index1 < min({publish1, read1}) then
           if ARRAY(index1) = TRUE then
(found)    publish1 := index1; goto 3;
           else
(inc)      index1 := the-next-index-in-PART1-or-M+1; goto 1;
           end
           else
(not_found) goto 3
           end
3 : (end)

```

Here we make some assumptions on the atomicity. They are similar to the atomicity assumptions made by Abrial/Cansell [5].

- We have a number of shared variables (e.g. the published values). They are the variables that are written by one process and read by the other process. They are the shared variable with respect to the *read* process.
- We have a number of local variables (e.g. the local indices).
- The events involving only local tests and actions can be performed concurrently.
- There is an elementary atomic action for reading the value of a shared variable into a local variables, e.g. $local_variable := shared_variable$.
- We extend the above atomic action to contain possible local test and local action.

```

when local_test then
  local_variable := shared_variable
  local_action
end

```

Different atomicity assumptions will lead to different *unfolded* versions of our program here. But this will not effect the applicability of our approach.

4 Formal Development

The machine-checked version of the development can be found on the web³. We first present our strategy for developing this program as follows.

Initial model specifies the result of the algorithm directly.

First refinement introduces the local indices of processes.

Decomposition step splits the model into sub-models corresponding to different processes: $main$, $process_1$, $process_2$.

We continue with further refinement steps for $process_1$ ($process_2$ should be developed in symmetrical fashion). Further development of the $main$ process is straightforward and is not of our interest here.

³ URL: <http://deploy-eprints.ecs.soton.ac.uk/154/>

First sub-refinement introduces the local index of the process.

Second sub-refinement introduces the read value of the process.

Third sub-refinement introduces the address counter for scheduling of events.

4.1 Initial Context and Model

The context defines an array of Booleans representing our abstract view.

constants: $ARRAY, M$

axioms:

axm0.1 : $M \in \mathbb{N}_1$

axm0.2 : $ARRAY \in 1..M \rightarrow BOOL$

The initial model contains only one integer variable called *result*. There is only one event *final* (beside the initialisation) to specify the result of the program *in-one-shot*. The aim of the program is encoded in the guard as constraints for parameter *k*.

```

final
  any k where
    k ∈ 1 .. M + 1
    ∀j · j ∈ 1 .. k - 1 ⇒ ARRAY(j) = FALSE
    k ≠ M + 1 ⇒ ARRAY(k) = TRUE
  then
    result := k
  end

```

4.2 First Refinement

The first refinement introduces the idea of using two processes. Here the context needs to be extended to include the notion of two different non-empty parts of the array.

constants: $PART1, PART2$

axioms:

axm1.1 : $PART1 \cup PART2 = 1..M$

axm1.2 : $PART1 \neq \emptyset \wedge PART2 \neq \emptyset$

At this point, the necessary information about the two sub-processes in order to obtain the final result of the program is whether or not they already terminate and the published results of the two processes. They are represented by a pair of variables, namely *finish1* and *publish1* (respectively *finish2* and *publish2*) for *process1* (respectively *process2*). Initially *finish1* (respectively *finish2*) is given the value FALSE, i.e. the process has not yet terminated; and *publish1* (respectively *publish2*) is assigned the value $M + 1$, i.e. the process has not yet found a result.

We first look at the refinement of the final event with the new set of variables. This event is carried out when the two processes have finished and the result taken is just the minimum of the two published values.

```

final
  refines final
  when
    finish1 = TRUE ∧ finish2 = TRUE
  with
    k = min({publish1, publish2})
  then
    result := min({publish1, publish2})
  end

```

In order to prove the refinement of the final event with respect to its abstract version, we need to give a witness for the disappearing parameter k of the abstraction. Here the parameter k is exactly the minimum of the two published values. Given the witness, the *simulation* proof obligation becomes trivial since both the abstract and concrete events assign equivalent expressions to the variable *result*.

We still need to prove *guard strengthening*. This requires us to give some invariants for the newly introduced variables. The invariants are symmetric for $process_1$ and $process_2$, hence we only give the five invariants associated with $process_1$ here.

invariants:

```

inv1.1  $publish1 \neq M + 1 \Rightarrow finish1 = \text{TRUE}$ 
inv1.2  $publish1 \neq M + 1 \Rightarrow publish1 \in PART1$ 
inv1.3  $publish1 \neq M + 1 \Rightarrow \text{ARRAY}(publish1) = \text{TRUE}$ 
inv1.4  $publish1 \neq M + 1 \Rightarrow (\forall i \cdot i \in PART1 \wedge i < publish1 \Rightarrow \text{ARRAY}(i) = \text{FALSE})$ 
inv1.5  $finish1 = \text{TRUE} \wedge publish1 = M + 1 \Rightarrow$ 
 $(\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow \text{ARRAY}(i) = \text{FALSE})$ 

```

inv1.1 states that if $process_1$ has published some result then it must have terminated.

This also means the process can publish at most once.

inv1.2–inv1.4 states that $process_1$ *cannot lie*: if it publishes some result then this must be the smallest index that it can find within $PART1$.

inv1.5 states that in the case where $process_1$ terminates without publishing any values, it has given up because it cannot find any better result than the other process $process_2$. The two possibilities for $process_1$ to terminate are:

- it has searched all the indices in $PART1$ and did not find any result, or
- it looks at the published value of the $process_2$ and knows that it cannot find a better (smaller) result.

In both situations, the invariant holds trivially.

We now abstractly construct the events to model the effect of the two processes on the new variables. These events correspond to the two cases in which a process can terminate. Here, we consider the events corresponding to $process_1$ only.

The first case is when $process_1$ finds a result within $PART1$ and terminates. Here $publish1 = M + 1$ is a theorem, which is the consequence of the first guard $finish1 = \text{FALSE}$ and invariant **inv1.1**. The other case is when $process_1$ terminates without publishing any value.

```

found_1
any  $k$  where
   $finish1 = \text{FALSE}$ 
   $k \in PART1$ 
   $\text{ARRAY}(k) = \text{TRUE}$ 
   $\forall i \cdot i \in PART1 \wedge i < k \Rightarrow \text{ARRAY}(i) = \text{FALSE}$ 
   $publish1 = M + 1$ 
then
   $finish1, publish1 := \text{TRUE}, k$ 
end

```

```

not_found_1
when
   $finish1 = \text{FALSE}$ 
   $\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow$ 
     $\text{ARRAY}(i) = \text{FALSE}$ 
then
   $finish1 := \text{TRUE}$ 
end

```

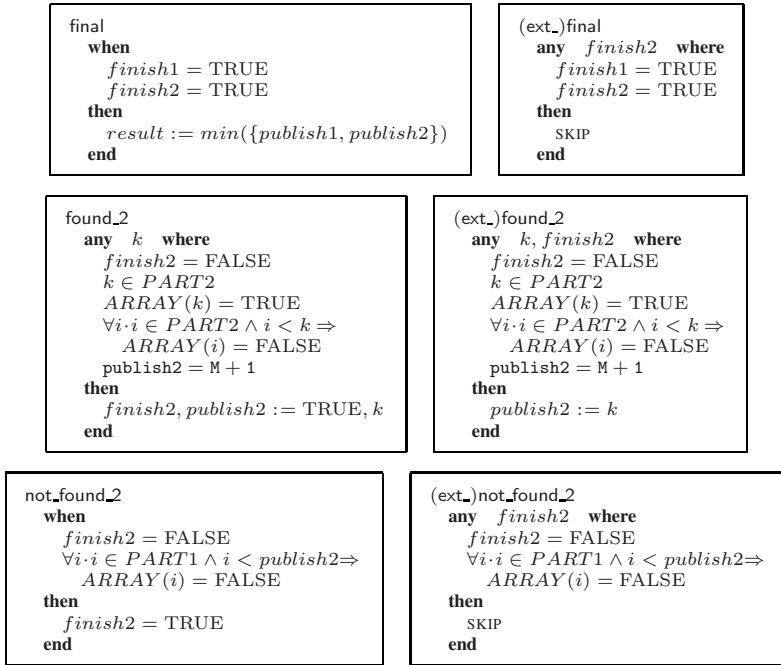
4.3 Decomposition

In the previous refinement step, we introduced the *interface* of the processes, i.e. the shared variables and events describing how these variables can be changed, which guarantees the correctness of the program. At this point, we want to develop in details each

process independently. We apply the technique of decomposition (shared variable) as described earlier in Section 2.3. There will be three different processes: *main* (final), *process₁* (found1, not_found1) and *process₂* (found2, not_found2).

As a result, we have three different sub-models, one for each process. Amongst these sub-models, the development *main* is straightforward and is not of our interest here. We concentrate on the sub-model for *process₁* (*process₂* is symmetric).

The sub-model for *process₁* contains three shared variables: *finish1*, *publish1* and *publish2* and no private variables. This process does not refer to either *result* (the global result) or *finish2* (if the other process has finish or not). According to the event distribution, this model of *process₁* has two internal events, namely found_1 and not_found_1, which are the exact copy of the original events. The other events become external which need to be generated as follows. We present the original events on the left and the corresponding external events for *process₁* on the right.



4.4 Further (sub-)refinements

In this section, we present the sketch of the further development of *process₁*. The refinement steps are all typical super-position refinement where more details about the actual process are introduced at each step as mentioned early in the start of Section 4. We do not present in detail the proofs of the correctness of the refinement steps here.

Introducing the local index: In the first sub-refinement for *process₁*, we introduce the index that the process is currently checking. This is represented by the new variable *index1*. The following invariants state that this process investigates only the part of the array belongs to *PART1* in ascending order and it cannot skip any index.

invariants:
inv2.1 $index1 \neq M + 1 \Rightarrow index1 \in PART1$
inv2.2 $\forall k \cdot k \in PART1 \wedge k < index1 \Rightarrow ARRAY(k) = FALSE$

The internal event `not_found_1` is unchanged. It trivially maintains the new invariants since it only modifies variable `finish1`. The same applies to external events, i.e. `(ext.)final`, `(ext.)found_2`, `(ext.)not_found_2` (which are always unchanged during refinement), since they do not refer to variable `index1`.

We now refine the internal event `found_1` to use `index1`: We also introduce a new event `inc_1` to model the case where the value at the current index is `FALSE` and hence `process1` moves to the next index.

```
found_1
refines found_1
when
  finish1 = FALSE
  index1 ≠ M + 1
  ARRAY(index1) = TRUE
with
  k = index1
then
  finish1, publish1 := TRUE, index1
end
```

```
inc_1
any i where
  ARRAY(index1) = FALSE
  i ≠ M + 1 ⇒ i ∈ PART1
  index1 < i
  ∀j · j ∈ PART1 ∧ index1 < j ⇒ i ≤ j
then
  index1 := i
end
```

For event `found_1`, the information from the witness $k = index1$ and the two invariants declared above guarantees that this is a correct refinement of the abstract event. For event `inc_1`, the parameter i is the smallest index in `PART1` that is greater than `index1`, or $M + 1$ if such an index does not exist. The proof that this event maintains the invariants is intuitive and can be found in our technical report [12].

Introduce the read value: In this refinement, we introduce the read value of process represented by variable `read1`. The constraint for this variable is expressed by invariant **inv3.1**: its value is either $M + 1$ or the published value of the other process, i.e. `publish2`. A new event `read_1` is introduced to model the situation when `process1` reads the published value of `process2`. This event sets the value of `read1` to `publish2` and hence clearly maintains the invariant **inv3.1**.

invariants:
inv3.1 $read1 \neq M + 1 \Rightarrow read1 = publish2$

```
read1
begin
  read1 := publish2
end
```

The only change to event `inc1` is two extra guards: $index1 < read1$ and $index1 < publish1$. Since this event does not change variables `read1` and `publish2`, it preserves the invariant **inv3.1** trivially.

The event `found_1` is refined by replacing the guard $index1 \neq M + 1$ with the following two guards: $index1 < read1$ and $index1 < publish1$. Since both `publish1` is either $M + 1$ or belongs to `PART1`, `publish1` is no greater than $M + 1$. Together with the guard $index1 < publish1$, `index1` is strictly smaller than $M + 1$. Hence the proof obligation for guard strengthening holds trivially.

We refine the remaining internal event `not_found_1` by replacing the guard $\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow ARRAY(i) = FALSE$ with $index1 < read1 \Rightarrow publish1 \neq$

$M + 1$. We do not go into detail of the proof why this is a correct guard strengthening, but refer the readers to our technical report [12].

For the external events, even though they are not refined, we must prove that they maintain the invariant **inv3.1**. In this case, we must consider those events that modify variable *publish2*. In our development, this is event (ext.)*found2*. The important part for our proof in this event is the theorem in the guard, i.e. $\text{publish2} = M + 1$, and the action $\text{publish2} := k$. According to the action, we have to prove that $\text{read1} \neq M + 1 \Rightarrow \text{read1} = k$, under the assumption of the invariants and the guards. From the theorem in guard $\text{publish2} = M + 1$ and invariant **inv3.1**, we have $\text{read1} = M + 1$ (since if it is not, then we have $\text{publish2} = \text{read1} \neq M + 1$). Hence $\text{read1} \neq M + 1 \Rightarrow \text{read1} = k$ holds trivially.

Introduce the address counter: In this last sub-refinement of *process1* we introduce the address counter in order to obtain the unfolded program as described in Section 3. The resulting internal events (with some refinement for guards) are as follows. These events conform with the notion of atomicity mentioned earlier.

<pre> read1 when address1 = 1 then address1, read1 := 2, publish2 end </pre>	<pre> not_found_1 when address1 = 2 ¬(index1 < min({publish1, read1})) then address1, finish1 := 3, TRUE end </pre>
<pre> found_1 when address1 = 2 index1 < min({publish1, read1}) ARRAY(index1) = TRUE then address1 := 3 finish1 := TRUE publish1 := index1 end </pre>	<pre> inc_1 any i where address1 = 2 index1 < min({publish1, read1}) ARRAY(index1) = FALSE i ≠ M + 1 ⇒ i ∈ PART1 index1 < i ∀j · j ∈ PART1 ∧ index1 < j ⇒ i ≤ j then address1, index1 := 1, i end </pre>

4.5 Proof Statistics

The proof statistics for the development is in the table below. We only take into account the number of obligations for sub-refinement models once, since the refinements for both process *process1* and *process2* are symmetric. We can use techniques such as pattern or generic instantiation in order to reuse the sub-development without re-proving again. In the table, 50% of the proof obligations are in the model before decomposing. This indicates that this refinement is the most important and difficult step in our approach.

Model	Number POs	Auto.(%)	Manual (%)
Initial context	0	0 (N/A)	0 (N/A)
Initial model	3	3 (100%)	0 (0%)
First extended context	0	0 (N/A)	0 (N/A)
First refinement	46	44 (96%)	2 (4%)
First sub-refinement	14	10 (71%)	4 (29%)
Second sub-refinement	6	5 (83%)	1 (17%)
Third sub-refinement	22	16 (73%)	6 (27%)
Total	91	78 (86%)	13 (14%)

5 Related Work and Conclusion

5.1 Related Work

The problem of verifying the *FindP* program has been tackled using different methods, notably using Owicki/Gries' *interference-free* [19] and Jones' *rely/guarantee* approach [14,15]. Moreover, the *FindP* program has been used as an illustrated example for the formalisation of these two approaches in Isabelle/HOL [18].

The work of Owicki/Gries [19] extends Hoare's deductive system for sequential programs [13] in order to prove the correctness of parallel programs. Their proofs of correctness for parallel statements centre around the notion of *interference-free* which is defined as follows. Given a proof of Hoare's triple $\{P\} S \{Q\}$ and a statement T with precondition $pre(T)$, T does not interfere with $\{P\} S \{Q\}$ if

InfFree1 $\{Q \wedge pre(T)\} T \{Q\}$, i.e. T maintains the post-condition Q , and
InfFree2 for any sub-statement S' of S , $\{pre(S') \wedge pre(T)\} T \{pre(S')\}$.

Within our approach, the above two conditions are verified during the development of the model at various refinement levels. At the abstract level before decomposition, S and T are some events of the models and the post-condition Q are just some invariants. For example, S are events belonging to $process_1$, T are events belonging to $process_2$, and Q are the invariants that state the outcome of $process_1$, e.g. **inv1.1–inv1.5**. We have to prove that these invariants are maintained by any events T and this corresponds to condition **InfFree1**.

Furthermore, during the sub-refinement of a process, sub-statements S' of S are introduced. At the same time, new invariants are added and these invariants correspond to the preconditions $pre(S')$ in the proof of $\{P\} S \{Q\}$ using Hoare's deductive system. Hence the condition **InfFree2** is verified by proving that events T (now external events) maintain the new invariants.

This is not too surprising, since in our approach, the role of external events is to keep track of the information about the possible changes on shared variables by different processes. During the refinement of a sub-process, we need to take into account the effect of these external events so that they do not "interfere" with the development of this sub-process. The main advantage of our approach over the work from Owicki/Gries is that these external events are at the abstract level rather than concrete statements as defined in the *interference-free* conditions. This reduces the complexity of the verification process.

Compared to the Owicki/Gries approach, our method is closer to the *rely/guarantee* approach of Jones [14]. The approach extends the notion of Hoare's triple $\{P\} S \{Q\}$ to encode the rely condition R and guarantee condition G . By definition, a condition $\{P, R\} S \{G, Q\}$ is satisfied by S if: under the assumptions that S starts in state satisfies the precondition P , and any external transition satisfies the rely condition R ; then S ensures that any internal transition of S satisfies the guarantee condition G , and if S terminates then the final state satisfies postcondition Q .

We focus on an example rule for parallel composition.

$$\begin{array}{l}
 R \vee G_1 \Rightarrow R_2 \quad (\mathbf{RG1}) \\
 R \vee G_2 \Rightarrow R_1 \quad (\mathbf{RG2}) \\
 G_1 \vee G_2 \Rightarrow G \quad (\mathbf{RG3}) \\
 \{P, R_1\} S_1 \{G_1, Q_1\} \quad (\mathbf{RG4}) \\
 \{P, R_2\} S_2 \{G_2, Q_2\} \quad (\mathbf{RG5}) \\
 \hline
 \mathbf{PAR-I} \quad \{P, R\} S_1 \parallel S_2 \{G, Q_1 \wedge Q_2\}
 \end{array}$$

The rule is interpreted as follows. Statement $S_1 \parallel S_2$ satisfies $\{P, R\} S_1 \parallel S_2 \{G, Q_1 \wedge Q_2\}$ if the following conditions are met. Firstly, both “global” rely condition R and the guarantee condition of one statement ensure the rely condition of the other (**RG1** and **RG2**). Secondly, both guarantee conditions of the two statements ensure the global guarantee condition G (**RG3**). Lastly, S_1 and S_2 independently satisfy their corresponding rely/guarantee condition (**RG4** and **RG5**)

Note that both rely and guarantee conditions are relations over two states. They are indeed similar to events in Event-B which correspond to a relations over pre/post-states. Moreover, the implication between rely/guarantee conditions is the same as event refinement. Within our approach, a pair of internal/external events encodes rely/guarantee conditions where the rely condition corresponds to the external event and the guarantee condition corresponds to the internal event. The generation of external events guarantees that they are the abstractions of the corresponding internal events. In fact, our generation of sub-models as described in Section 2.3 guarantees that the resulting sub-models satisfy the parallel composition rule. This is the advantage of our approach over the *rely/guarantee* method. In fact the external events are the strongest possible condition that the other process can rely on. In practise, the rely/guarantee conditions could be more abstract, e.g. requires only that the value of some variables decrease monotonically [16]. Moreover, rely/guarantee is usually used for composition rather than decomposition as in [1].

The decomposition technique also appears in many other approaches, with similar intuition: Breaking a specification into smaller pieces and reasoning about them independently. For example, in the work of Abadi/Lamport [1], this is captured by their *Decomposition Theorem* and a generalised version of it. The most important idea in their approach is to find some properties E (also called *environment*) of the other processes assumed by a process. However, in another study, Lamport claimed that decomposition might not be that useful [17]. One of the argument is the difficulty in inventing the *environment* properties and checking the hypotheses of the decomposition theorem. In our approach, we *derive* these properties from the overall purpose of the program using refinement (step 2 of our approach). This is also the reason why we consider the class of parallel programs that achieve some intended result.

Stepwise refinement has been considered for developing parallel systems in Action System in early work of Back/Sere [8,9]. The shared variable decomposition in Event-B corresponds to their notion of *concurrent action system* (in contrast to *distributed action system* with shared actions). However, the approach presented in [8] based on the notion of refining atomicity introduces the notion of parallelism quite late in the development (almost as the last step of the refinement chain). The reason for this delay is that the decision for implementing the system as concurrent action system or distributed action system can be made as late as possible. In our example, we have this decision of using

shared variables in advance. Hence we can take the advantage of having the decomposition early to reduce the complexity. We consider the use of shared variables as a part of the design process of the program rather than an implementation detail.

5.2 Conclusion

We have presented a method for developing parallel programs using refinement and decomposition techniques. Refinement gives us the possibility to abstractly define the aim of the programs which helps us to understand the purpose of these programs. Decomposition allows us to reduce the complexity of the development by separately developing sub-processes while keeping track of minimum information on what other processes can do. Our approach should be applicable to all programs that use several parallel processes in order to obtain a certain goal.

Our approach introduces the possible *interaction* between processes early in the development in order to take the advantage of decomposition. This is different from the approach where one develops processes according to the implementation of the process with possible *cheating* (e.g. one process directly looks into the value of the other process), and subsequently refines the model until there is no more cheating. This approach has been proposed in [3] and is used in many other examples. Applying this approach without using decomposition, the two processes are developed together, and hence the development also has higher complexity comparing to our approach.

The key aspect of our development using decomposition lies in the model that is being decomposed, where we have to abstractly specify the effect of the two future processes on shared variables. We use the overall intended result of the program to help us to *derive* the requirement on the future processes. Furthermore, as a result of using step-wise refinement, we can develop sub-processes using different implementations as long as they satisfy the abstraction. As an example, we can also “implement” the two processes (inefficiently) by not checking the published values of the other processes or having more fine-grained version of atomicity.

For future work, we would like to apply our method to other standard parallel programs (not necessarily ones with intended final result) known from literature, such as “bounded buffer”, “partition of set” or “bubble-lattice sort”, which have been studied using other approaches [10]. Our approach should not only be used for verification a posteriori but also for finding proofs of correctness for such systems.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. *ACM Trans. Prog. Lang. Syst.* (1995)
2. Abrial, J.-R.: Event model decomposition. Technical Report 626, ETH Zurich (May 2009)
3. Abrial, J.-R.: Modeling in Event-B: System and Software Design. CUP (2009) (to appear)
4. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
5. Abrial, J.-R., Cansell, D.: Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS* (2005)

6. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica* (2006)
7. Back, R.-J.: Refinement calculus, part II: Parallel and reactive programs. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) *REX Workshop*, pp. 67–93 (1989)
8. Back, R.-J., Sere, K.: Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.* (1989)
9. Back, R.-J., Sere, K.: Superposition refinement of parallel algorithms. In: *FORTE* (1991)
10. Barringer, H.: *A Survey of Verification Techniques for Parallel Programs*. LNCS, vol. 191. Springer, Heidelberg (1985)
11. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. CUP (2001)
12. Hoang, T.S.: Event-B development of the FindP program. Technical Report 653, ETH Zurich (November 2009)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* (1969)
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* (1983)
15. Jones, C.B.: The role of proof obligations in software design. In: Ehrig, H., Floyd, C., Nivat, M., Thatcher, J. (eds.) *TAPSOFT 1985 and CSE 1985*. LNCS, vol. 186. Springer, Heidelberg (1985)
16. Jones, C.B.: Splitting atoms safely. *Theor. Comput. Sci.* (2007)
17. Lampert, L.: Composition: A way to make proofs harder. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) *COMPOS 1997*. LNCS, vol. 1536, p. 402. Springer, Heidelberg (1998)
18. Prensa Nieto, L.: *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München (2001)
19. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. *Acta Inf.* (1976)
20. Rosen, B.K.: Correctness of parallel programs: The Church-Rosser approach. *Theor. Comput. Sci.* (1976)

Communication Systems in ClawZ

Michael Vernon¹, Frank Zeyda², and Ana Cavalcanti²

¹ QinetiQ, Cody Technology Park, Farnborough, Hampshire, GU14 0LX, U.K.
mrvernon1@qinetiq.com

² University of York, York, YO10 5DD, U.K.
{zeyda, ana}@cs.york.ac.uk

Abstract. We investigate the use of ClawZ, a suite of tools for the verification of implementations of control laws, to construct formal models for control systems in the area of communications and signal-processing intensive applications. Whereas ClawZ has been successfully applied to verify control components in avionic systems, special requirements need to be identified and addressed to extend its use to the aforementioned application domain. This gives rise to several extensions, which we explain and subsequently validate by constructing the Z model of a software-defined radio communication device. The experience reported provides insight into general issues surrounding the use and extension of ClawZ.

Keywords: control laws, signal processing, formal models, Z, Simulink.

1 Introduction

Control law diagrams are a graphical notation widely used by engineers to specify the behaviour of control systems. In industry, the commercial tool Simulink by MathWorks [11] is a de-facto standard for the design of control diagrams. Roughly speaking, control diagrams consist of blocks that carry out elementary functions, and wires that transmit data values between those blocks. Diagrams communicate with the environment through designated input and output port blocks. They may also exhibit structure in which the functionality of basic blocks may itself be described by virtue of lower-level diagrams. Additionally, Simulink provides a comprehensive library of blocks and supplementary toolboxes to support the specification of control systems for particular application domains.

Here, a formal approach to verification of implementations of diagrams is advocated. If we cannot rely on automatic code generators to ensure correctness, for instance, because code has to be optimised, the ClawZ suite of tools [4,2] can be used to construct a proof of the correctness of an Ada implementation. ClawZ is a highly automated set of utilities for use in industrial-scale projects.

Software-defined radios have recently gained popularity [10]; they perform most of their signal-processing operations in software, for instance, on a personal computer or DSP. This allows them to support simultaneously many communication standards, each requiring specific demodulation and decoding techniques, in a single integrated piece of hardware [8]. Their potential use in the military sector and other safety-critical areas highlights the need for formal verification [6].

Although ClawZ provides flexible mechanisms to configure and extend it for use with a wider class of diagrams, it lacks support for control laws that are typically found in the design of signal-processing and radio communication devices. These diagrams differ from others in that they require support for complex number arithmetics. Simulink is oblivious to this distinction due to the effective polymorphism of block behaviours, but the formal model needs to reflect the difference. Similarly, ClawZ does not support matrices as signal types.

A second problem is that many blocks commonly found in signal-processing models, such as filters, modulators, Fourier transformers, and so on, are not part of the library of translatable blocks in ClawZ. Extending this library requires Z models to be developed for the blocks according to their function.

Here we report on work that realises the above enhancements, validates them in the context of a software-defined radio case study, and thereby provides evidence that it is possible to use ClawZ for generating formal models for this class of diagrams. This widens the applicability of ClawZ, and sheds light into a few issues of the ClawZ approach to building Z models, which imposes limitations on automation, namely due to insensitivity to signal types.

In Section 2 we provide further details on ClawZ and identify requirements for using it for signal-processing control systems. Section 3 discusses our extensions of ClawZ; Section 4 validates them using our case study. Finally in Section 5 we draw our conclusions and suggest future work.

2 Preliminaries

After providing an overview of ClawZ in Section 2.1, we discuss the main features of communication control systems in Section 2.2.

2.1 ClawZ

The ClawZ verification process involves the construction of a Z model for a Simulink diagram acting as a specification of behaviour to which an implementing Ada code has to adhere. ProofPower-Z [13], a theorem prover for Z based on higher-order logic (HOL), is used for mechanical proof. Correctness is established by an embedding of the refinement calculus into ProofPower-Z. It has been successfully used in industry, with a measured cost reduction of 20% in the certification of avionics systems. The fact that it can be adapted to other areas, and that the same high level of automation can be achieved by programming of proof tactics, makes this approach very attractive.

The ClawZ tools are bespoke and automated, tailored for engineers without in-depth knowledge of formal specification and proof. The tool that carries out the translation of diagrams into Z specifications is the Z Producer. To illustrate its approach, we consider the diagram consisting of a Sum and Product block in Fig. 1. The main schema in the specification generated for this diagram is also in Fig. 1. It introduces components for the input and output ports of the diagram, namely $In1?$, $In2?$, $In3?$ and $Out1!$; port variables are obtained by suffixing

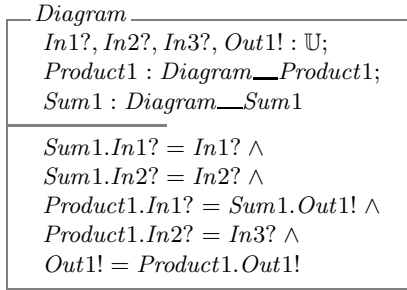
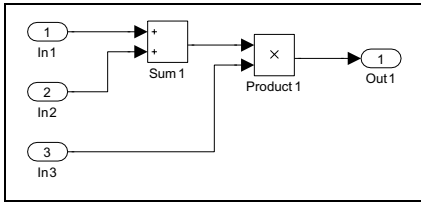


Fig. 1. Simple Simulink diagram consisting of two blocks

In and *Out* with the port number. The schema also includes components that characterise the behaviour of the blocks.

Whereas the *Product1* and *Sum1* components represent particular instances of the *Product* and *Sum* blocks, *Diagram_Product1* and *Diagram_Sum1* are schema types that encapsulate the behaviours of these blocks. Upon translation these types are introduced by associating them with suitable block definitions in the ClawZ block library. The library is contained in a separate *ProofPower-Z* theory acting as a carrier to hold those definitions. *ProofPower* theories are in essence collections of type definitions, constants, defining axioms and theorems.

The type *Diagram_Product1*, for example, is defined as *Product_M2* where *Product_M2* is the block schema specified in the library. Its definition is given below and follows the same conventions on port names.



Above, the types of the ports are explicitly given, whereas in *Diagram* they are unspecified; \mathbb{U} acts as a generic type to be inferred by the typechecker. The equations in the predicate of *Diagram* such as $Product1.In1? = Sum1.Out1!$ encode the wiring of blocks: each connecting wire results in one equality.

Block specifications may be functions yielding schemas too. This allows models to be parameterised by arguments set for the block inside Simulink. For example, the initial output of a unit delay block, which delays a signal by one cycle of the control system, is a parameter of its model in the library: a function from \mathbb{U} to the schema type representing the block.

The Simulink diagrams to be translated by the Z Producer can be arbitrarily structured. For example, the diagram in Fig. 1 could itself appear as a block in a higher-level model. In this case *Diagram* would act in turn as a schema type for the aggregated component representing that block. ClawZ also includes additional schemas in the model that specify the behaviour of blocks for reset and hold cycles, that is when the signal value is either reset to some initial value, or simply retained. Finally, an additional feature allows certain simple block models to be constructed on-the-fly rather than imported from the library.

The above only considers a simple example but in essence illustrates how the generation of Z models is carried out. A crucial aspect is that it is realised automatically, mostly as a syntactic transformation from the Simulink MDL file that gives the textual description of a diagram to the respective ProofPower-Z theory source file. This surfaces, for example, in that the types of port components in the *Diagram* schema are given as \mathbb{U} due to no type information for signals being available that could otherwise be exploited to specify exact types.

2.2 Signal-Processing Features

The following techniques are particularly relevant for signal-processing models. Their support in ClawZ is the primary objective of our work. Communication devices heavily rely on them, but they are also relevant for the many applications that require digital image and sound processing or data compression.

Filtering: Filtering in essence allows to shape signals by amplifying frequencies within desired ranges while suppressing others considered as noise or irrelevant. It is a fundamental operation within many signal-processing algorithms to extract information from signals and prepare them for further processing. The theory of digital filters is well-developed; basic classifications are Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. Filters are usually characterised by their dimension and filter coefficients. Such can either be specified statically, or, in adaptive filters, adjusted dynamically according to the minimisation of the error between a desired and actual signal.

Modulation: Modulation is used, for instance, to transfer binary data over analog passband channels, and is geared towards the capabilities and characteristics of the channel to maximise the amount of information transmitted. Modulation is usually carried out before information is transmitted through a channel, and demodulation, its inverse, to retrieve the information upon reception.

Various approaches to modulation exist such as Quadrature Amplitude Modulation (QAM), Phase-Shift Keying (PSK), Frequency-Shift Keying (FSK), and others. Modulation gives rise to signals being interpreted in the complex number plane; an essential aspect for their support is hence complex arithmetics.

Encoders / Decoders: The encoding of signals has the dual purpose of adding redundancy for error detection and correction, and encrypting signals that carry sensitive information, for example in military communication devices. It is usually carried out prior to modulation. An example we considered is the Trellis encoding of digital signals which is based on convolution codes. Those codes are frequently used in digital radio applications because of their favourable properties approaching theoretical limits for the amount of information that they transport through a lossy channel.

Fourier Transformation: Fourier-transformation is a final operation of interest, playing an important part in analysing and compressing signals, for example in image or speech processing applications. Again, the Fourier-transform of

a real-valued signal is usually a complex function where phase and amplitude encode amplification and phase-shift of respective frequencies. Again this makes the support for complex numbers and their operators imperative.

3 Extension of ClawZ

In this section we explain in more detail how we extend ClawZ to support the signal-processing features that were outlined in the previous section. For this we first discuss the support for additional data types like complex numbers and matrices, and then report on additions made to the ClawZ block library.

3.1 Addition of Data Types

The subset of the Simulink notation that can be handled (modelled in Z) using ClawZ only admits scalars (of type real), and vectors (of type real). In communication-related control laws we often require to operate on scalars, vectors and matrices of complex numbers. They are used, for example, to encode the amplitude and phase of signals in the frequency domain as they occur in Fourier transforms or demodulation of signals. This suggested two fundamental extensions to the ClawZ tools: one is to deal with complex numbers as such, and another is to support complex vectors and matrices as data values being passed between the blocks of a Simulink diagram.

For integrating these extensions it makes sense to distinguish between two independent concerns: firstly to formalise them in the logic of the underlying theorem prover, ProofPower-Z, and secondly to extend the Z Producer to handle the new types in the translation of Simulink diagrams into formal Z models. There exists no comprehensive embedding of complex numbers in ProofPower-Z to our knowledge, however, a case study is available on the ProofPower web-pages that illustrates such an extension in principle; we pursue a similar approach being described in what follows. The complete set of definitions can be found in Vernon's MMath thesis [14], and the ProofPower-Z theory source is made available for download at <http://www.cs.york.ac.uk/circus/tp/tools.html>.

To formalise complex numbers we introduce a new axiomatic constant \mathbb{C} as the set of tuples of real numbers: $\mathbb{C} \hat{=} \mathbb{R} \times \mathbb{R}$. This set acts as the *type* used for complex numbers. The first component represents the real part of the number, and the second, the imaginary part. ProofPower-Z supports real-number arithmetics by means of a collection of relations and functions that operate on elements of \mathbb{R} such as $+_R$, $*_R$, \leq_R , and so on. The subscript highlights what type of value these functions expect, and a repository of axioms and derived theorems permits reasoning about formulae involving the operators. Similarly, we introduce a set of function definitions that operate on elements of \mathbb{C} .

Most of the axioms for operators directly correspond to familiar textbook definitions of those operators. For example, the following Z axiomatic definition introduces multiplication of complex numbers.

$$\left| \begin{array}{l} \underline{- *_{\mathbb{C}} - : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}} \\ \forall z, w : \mathbb{C} \bullet z *_{\mathbb{C}} w = (z.1 *_R w.1 -_R z.2 *_R w.2, z.1 *_R w.2 +_R z.2 *_R w.1) \end{array} \right.$$

The dot operator selects the components of a tuple. For lack of space we will not further discuss the remaining operators but point to [14] for their definitions.

Vectors of complex numbers are characterised by sequences over \mathbb{C} being similar to how ClawZ encodes vectors of real numbers in the Z model of diagrams. Again, a collection of useful operators is defined, for instance, to calculate the scalar product and the sum of vectors. Complex matrices, on the other hand, we characterise by sequences of sequences over elements from \mathbb{C} ; each inner sequence represents one row of the matrix. It shall be noted that other characterisations are conceivable too, but at present we have no conclusive evidence to favour one over another. The following set introduces the matrix type formally in Z.

$$\left| M_C \hat{=} \{m : \text{seq}_1 (\text{seq}_1 \mathbb{C}) \mid \forall i, j : 1..(\# m) \bullet \# m(i) = \# m(j)\} \right.$$

A requirement here is that the inner sequences must all have the same length. We also exclude the limit case of zero-dimensional matrices by confining ourselves to non-empty sequences; in practice there is no need for such matrices.

As with vectors, there exists a number of specific operations applicable to matrices. The following definition, for instance, introduces multiplication of complex matrices. The number of columns of the first matrix $\# A(1)$ must be equal to the number of rows of the second matrix $\# B$ to apply the operator.

$$\left| \begin{array}{l} \underline{- *_{MC} - : M_C \times M_C \mapsto M_C} \\ \text{dom}(- *_{MC} -) = \{A, B : M_C \mid \# A(1) = \# B\} \wedge \\ (\forall A, B : M_C \mid \# A(1) = \# B \bullet \\ A *_{MC} B = \{m : 1..(\# A) \bullet m \mapsto \{n : 1..(\# B(1)) \bullet \\ n \mapsto \text{SumSeq}_C (\{k : 1..(\# A(1)) \bullet k \mapsto A(m)(k) *_C B(k)(n)\})\}) \end{array} \right.$$

Multiplication is carried out according to the rule $C_{mn} = \sum_{k=1}^K A_{mk} *_C B_{kn}$ where m ranges over the rows of A , and n over the columns of B . The function SumSeq_C is introduced to calculate the sum of the elements of a complex sequence.

Other operators that have been formalised include matrix addition, transposition, complex-conjugate, and functions to extract the real and imaginary part of complex matrices. We also provided operators for scalar multiplication as well as multiplication of matrices with (complex) vectors of the correct size.

ClawZ Integration: Whereas the formalisation of complex arithmetics and matrix algebra is not a difficult problem *per se*, more challenging proves to incorporate the new types and operators into the generation of Z models from diagrams. As previously mentioned the translation carried out by ClawZ is mostly a syntactic transformation process in that it does not utilise information about signal types; type information for signals is neither inferred nor exploited. (To be accurate, there are situations where signal types are taken into account, that is, the synthesis of certain kinds of blocks, but this is not the general case.)


```

BlockSpecification {
  Zname Sum_P2
  SelectionParameters {
    BlockType Sum
    Ports [2, 1]
    Inputs "2"
    InputTypes "RR"
  }
}

BlockSpecification {
  Zname Sum_P2C
  SelectionParameters {
    BlockType Sum
    Ports [2, 1]
    Inputs "2"
    InputTypes "CC"
  }
}

```

Fig. 2. Entries in ClawZ’s library meta-file for the Sum block

A problem in the translation arises due to the fact that Simulink blocks exhibit polymorphic behaviours, meaning the same type of block may process input signals of different types. For example, the Sum block can be used to add two scalar inputs, one scalar and one vector, two vectors, or even matrices if their dimensions agree. In the formal model such blocks must have different specifications, depending on their types of inputs. The problem is exacerbated in our work as we moreover have to consider operations on complex types.

To illustrate the above, consider several possible encodings of the Sum1 block in Fig. 1 as a Z schema. One may assume the inputs $In1?$ and $In2?$ to be real scalars, giving rise to $[In1?, In2?, Out1! : \mathbb{R} \mid Out1! = In1? +_R In2?]$ as its block schema. We may alternatively consider the inputs to be vectors, matrices, or complex scalars, giving rise to different characterisations. The problem does in fact surface even when translating ‘conventional’ diagrams, and the solution adopted by ClawZ is to support means of selectively determining what underlying formal model to use for particular blocks. Consequently, the user is sometimes required to analyse the diagram by hand when generating its Z model.

To address this problem in the context of our extensions, we propose an initial analysis of the Simulink diagrams prior to translation by the Z Producer in which type information is injected into the model. This can either be done manually, or automatically by means of a typechecker. A separate file is created that contains the type information for blocks, and a tool populates them into the respective Simulink MDL file. Type information is made explicit through additional attributes (name/value pairs) in the records that describe information related to the blocks as they are encoded in the MDL file of the model.

The above process allows us to make the syntactic matching of the Z Producer, that infers which schemas are used to characterise block functionality, sensitive to (semantic) type information. To explain, the association of entries in the Simulink file and Z schemas is determined by the ‘library meta-file’ of ClawZ. Fig. 2 shows how we extend the default entries in this file to take into account the injected type attributes. The entries for **Zname** determine the schema used for the block, and the **InputTypes** attributes are additional matching conditions. Instantiation of a block upon translation only takes place if all attributes in the **SelectionParameters** clause are present in the model file. In particular, the **InputTypes** attributes have to exist. Although currently we insert them manually, their generation can potentially be automated by a tool.

In practice, in order to support the new types no low-level modifications to the Z Producer and its code are necessary. The axiomatisation of complex numbers, vectors, matrices, common operators, and Simulink block functionality can be cleanly encapsulated in a collection of designated ProofPower-Z theories. These theories are then merged with the default theory of ClawZ, containing the tool itself as well as the standard library of block definitions. The resulting database of theories is then configured as the default parent when generating the ProofPower-Z database for the specification of particular Simulink models.

3.2 Support for Communication Blocks

The applications we like to consider contain various types of blocks which are initially not supported as part of the translatable subsets of ClawZ. To incorporate support for these blocks we have extended the ClawZ block library. It consists of providing specifications for the blocks as Z schemas and library-meta data to recognise the blocks in Simulink diagrams and thereby generally enable their translation. For reasons of space we cannot comment on all extensions here but will discuss the most significant ones. The complete ProofPower-Z theories are available from <http://www.cs.york.ac.uk/circus/tp/tools.html>.

Digital Filtering: Digital filters are frequently used in signal-processing and communications applications. Their effect can be generally described by the following difference equation.

$$y(n) = \frac{1}{a_0} \left(\sum_{i=0}^M b_i * x(n - i) - \sum_{j=1}^N a_j * y(n - j) \right)$$

Here $x(n)$ refers to the input signal and $y(n)$ to the output signal at time step n ; the a_j and b_i are the feedback and forward filter coefficients, respectively. Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters are both characterised by the above equation with the difference that for FIR filters N is zero, hence the output only depends on the history of inputs $x(n - k)$ for $k \in 0..M$, and not recursively on previous outputs. As the name suggests, the impulse response function for FIR filters is non-zero only for a finite range whereas for IIR filters it usually extends to infinity.

There are various ways of realising filters by means of primitive elements such as Gain, UnitDelay and Sum blocks realising multiplication with a factor, delay of a signal, and summation. The Direct Form I utilises two cascades of delay blocks as buffers, one for the inputs and one for the output; the Direct Form II, on the other hand, is more compact using only one such cascade. If the filter is modelled by a lower-level subsystem, we can indeed use ClawZ *as is* to generate the Z model — as long as inputs are assumed to be real values.

The Communications Toolbox for Simulink, on the other hand, provides new atomic blocks that support signal filtering. For these blocks we have to provide schema definitions that specify their functionality in order to allow for their translation into Z models. Based on the actual implementation of the filter it is

$$\begin{array}{|l}
IIR1 : [a, b : \text{seq}_1 \mathbb{C}] \rightarrow [x, y : \mathbb{C}; x_buff, x_buff', y_buff, y_buff' : \text{seq } \mathbb{C}] \\
\forall pars : [a, b : \text{seq}_1 \mathbb{C}] \bullet \\
\quad IIR\ pars = [x, y : \mathbb{C}; x_buff, x_buff', y_buff, y_buff' : \text{seq } \mathbb{C} \mid \\
\quad \# x_buff = \# pars.b - 1 \wedge \# x_buff' = \# x_buff \wedge \\
\quad \# y_buff = \# pars.a - 1 \wedge \# y_buff' = \# y_buff \wedge \\
\quad (\exists x_sum, y_sum : \mathbb{C} \mid \\
\quad \quad x_sum = \text{SumSeq}_C (\text{MultSeq} (pars.b, \langle x \rangle \wedge x_buff)) \wedge \\
\quad \quad y_sum = \text{SumSeq}_C (\\
\quad \quad \quad \{i : 1..(\# pars.a - 1) \bullet i \mapsto pars.a(i + 1) * y_buff(i)\} \bullet \\
\quad \quad \quad y = (\text{recip}_C\ pars.a(1)) * (x_sum - y_sum) \\
\quad \quad \quad x_buff' = \langle x \rangle \wedge \{i : 1..(\# x_buff - 1) \bullet i \mapsto x_buff(i)\} \wedge \\
\quad \quad \quad y_buff' = \langle y \rangle \wedge \{i : 1..(\# y_buff - 1) \bullet i \mapsto y_buff(i)\})]
\end{array}$$

Fig. 3. Schema specifying the digital filter block in the Direct Form I

beneficial to use a formal model that is closest to it, we therefore provide a set of (equivalent) specifications reflecting various filter implementations. However, for space considerations we only discuss one of them.

Fig. 3 includes the schema that directly translates the difference equation, resembling the realisation of the filter in the Direct Form I. The schema is obtained by applying *IIR1* to a binding of type $[a, b : \text{seq}_1 \mathbb{C}]$, which provides the filter coefficients. Here, a and b are parameters that are extracted from the attributes of the block as it is encoded in the MDL file; they are set inside the Simulink tool. The schema has the components x and y , corresponding to the current input and output, and additional state components x_buff and y_buff , including their primed counterparts, to maintain a history of previous inputs and outputs; the latter are modelled by sequences. The length of the buffer sequences is one less the length of the respective coefficient sequences $pars.a$ and $pars.b$ which implicitly determine the dimension of the filter. The two summation terms in the difference equation are assigned to the local constants x_sum and y_sum . Both are used to calculate the output y by multiplication with the reciprocal value of a_0 , given by $pars.a(1)$. Observe that MultSeq_C realises element-wise multiplication of complex sequences. Finally it is necessary to shift the contents of the buffers, adding the current input and output as new head elements.

The above schema cannot be directly used by ClawZ since it does not conform to the naming conventions on input and output ports and parameters. We lift it into a block schema *IIR1_Block* by renaming a and b to *NumCoeffs* and *DenCoeffs*, as well as x and y to *In1?* and *Out1!* to achieve this conformance.

The schema is configured for translation using the library-meta file entry included in Fig. 4. Here, the matching attributes specify **Reference** as the block type, hinting that the block is instantiated from a supplementary Simulink toolbox. It is further classified as a filter by the **SourceType** attribute, and the remaining attributes specify the kind of filter used. Since the block has to transmit several parameters, they also need to be identified in the library meta-file entry; it is done by virtue of the **TransmittedParameters** clause.

```

BlockSpecification {
  Zname IIR1_Block
  SelectionParameters {
    BlockType Reference
    SourceType "Digital Filter"
    TypePopup "IIR (poles & zeros)"
    IIRFiltStruct "Direct form I"
    CoeffSource "Specify via dialog"
  }
  TransmittedParameters {
    NumCoeffs Vector
    DenCoeffs Vector
  }
}

```

Fig. 4. Library meta-file entry for the IIR1 filter block

<p><i>DFT_Block</i></p> <p><i>In1?</i>, <i>Out1!</i> : seq₁ ℂ</p> <hr/> <p># <i>Out1!</i> = # <i>In1?</i> ∧ (∀ <i>k</i> : 1..(# <i>In1?</i>) • <i>Out1!</i>(<i>k</i>) = <i>SumSeq_C</i> ({<i>n</i> : 1..(# <i>In1?</i>) • <i>n</i> ↦ <i>In1?</i>(<i>n</i>) * <i>exp</i>(~_C (0.0, 2.0 *<i>R</i> π) *<i>C</i> <i>z2c</i> ((<i>k</i> - 1) * (<i>n</i> - 1)) /_C (<i>z2c</i> # <i>In1?</i>))}))</p>

Fig. 5. Block Schema for the Discrete Fourier Transformation

In addition to the standard filters two instances of adaptive filters have been included, respectively supporting the Least Mean Square (LMS) and Root Mean Square (RMS) algorithms for adjusting the filter coefficients. They are not further discussed here but explained in more detail in [7]. Notably, the specification of the RMS filter required complex matrices to record correlations.

Fourier Transformation: Fourier transformations are performed to convert a signal in time into a corresponding signal in the frequency domain. The discrete Fourier transform (DFT) of a signal $x(n)$ for $n \in 0..(N - 1)$ and its inverse is defined by the following pair of equations.

$$y(k) = \sum_{n=0}^{N-1} x(n) \exp\left(-\frac{2\pi ikn}{N}\right) \quad \text{and} \quad x(k) = \frac{1}{N} \sum_{n=0}^{N-1} y(n) \exp\left(\frac{2\pi ikn}{N}\right)$$

The Fourier transform $y(n)$ will usually be a vector of complex values that determine the amplitude and phase of equidistant frequencies. The DFT is always an invertible transformation, assuming the signal is repeated periodically.

To support a corresponding block of the Communications Toolbox that performs this operation, we first provide additional definitions for complex exponentiation and the constant π . Exponentiation was defined using the Taylor expansion $e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!}$, but other approaches may be possible too, for example using Euler’s equation in the particular case above. This allowed us to define the corresponding block schema for the operation as given in Fig. 5. The

function $z2c$ here is a utility operator that converts an integer into a corresponding complex number, and \sim_C is negation of complex numbers.

Modulation: The block we will look at in more detail here is the Quadrature Amplitude Modulation (QAM). The essence of QAM modulation is that an integer value in the range $1..n$ to be encoded is mapped onto a point in the complex plane given by the signal constellation diagram. For 16-QAM ($n = 16$) the latter consists of a regular grid of 16 equidistant points where each point represents one of the symbols in the permissible range 1 to n . The encoding simply involves associating each value with a point in the grid, and the decoding determines the symbol of the point that is closest to a given point represented by the complex (baseband) signal obtained from the demodulated carrier signal.

In order to characterise this functionality as a block schema we first introduce a constant $RQAM_16_SCD : \text{seq } \mathbb{C}$ being a sequence that records for each of the values in the range 1 to 16 the corresponding point associated in the complex plane. Encoding is now simply applying the sequence as a function.

For the decoding we have to determine which symbol is closest to the actually received signal. It is realised by the following block schema.

$RQAM_16_Demod_Block$
$In1? : \mathbb{C}; Out1! : \mathbb{Z}$
$Out1! \in 1..16 \wedge (\forall n : 1..16 \bullet$ $Abs_C(In1? -_C RQAM_16_SCD(Out1!)) \leq$ $Abs_C(In1? -_C RQAM_16_SCD(n)))$

Above Abs_C is the absolute value of a complex number being its distance from the origin. We simply require that the symbol output by $Out1!$ is the one which has the minimal distance to $In1?$ amongst all points on the grid.

We have encoded several blocks beyond the ones mentioned in this section, for example the Trellis encoding of a signal for a specific convolution code, and its decoding using the Viterbi algorithm. This is reported in more detail in [14].

4 Case Study: Software-Defined Radio

Software-defined radios (SDRs) are radio-communication devices in which components typically found in those devices like, for example, mixers, filters, amplifiers, modulators and demodulators, and so on, are implemented in software rather than being statically realised in hardware. They are designed to sometimes carry out the work of multiple radio devices in a single piece of hardware as they have the ability to support different bandwidths, modulation techniques and communication standards all at once. There are various examples of SDRs in the home and consumer market, for instance in mobile phone devices, but notably military applications profit from their versatility with the addition of encryption and security-related features; this renders the SDR as a potential safety-critical system whose development profits from the use of ClawZ.

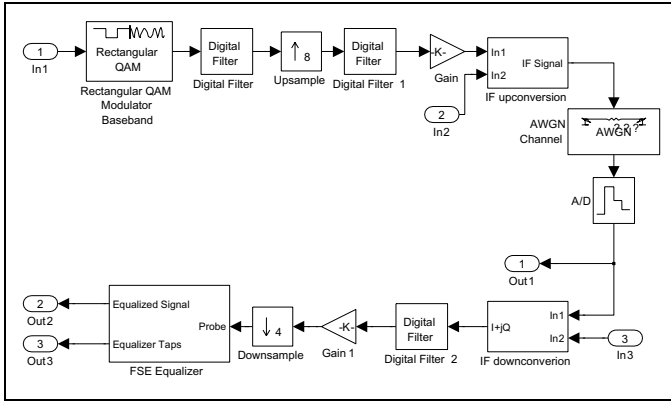


Fig. 6. Control law diagram for a Software-defined Radio

The ClawZ extensions we discussed in the previous section have been used to construct the formal model for the Simulink diagram of a specimen SDR which can be found as one of the examples published on-line by MathWorks [5]. The corresponding diagram is included in Fig. 6. It models the modulation and encoding of the source signal, transmission through the ether, and demodulation and decoding of the received signal. The diagram requires the Communications and Signal Processing Toolbox for Simulink. It is hierarchically organised as IF upconversion, IF downconversion and FSE Equalizer are subsystem blocks. (Their respective diagrams are omitted here.) The model is overall not complex in structure, but almost all its basic blocks were taken from the external toolboxes.

The model contains various elements which ClawZ initially did not recognise, and whose support we previously discussed. The input signal $In1$ is first submitted to a QAM modulator and further passed to a digital filter. These are elementary blocks of the Simulink Communications Toolbox utilised by the model. The following Upsample block changes the rate of the signal, and the output is further submitted to a digital IIR filter. Since the QAM modulation produces a complex output, both filters operate on complex signals. The conversion of the complex (baseband) signal into a real (passband) signal is achieved by the IF upconversion subsystem. It modulates the signal onto a carrier frequency obtained via $In2$ in order to prepare it for transmission. The transmission of the signal is simulated by a noisy channel, that is the AWGN Channel.

Upon reception similar operations are performed to the ones already mentioned; we will not explain them in detail here. It shall be noted, however, that the FSE Equalizer subsystem requires the LMS adaptive filtering block.

Prior to the extension of ClawZ very little of the model could actually have been translated. Following the extensions, it was possible, with a few model-specific customisations, to translate the entire diagram, and successfully parse and typecheck the Z specification within ProofPower. A few blocks had to be further added like the AWGN block to simulate additive white Gaussian noise,

or the `Upsample` and `Downsample` blocks whose function it is to change the rate of a signal. This did not pose a problem in practical terms.

To handle the problem of polymorphic blocks we implemented a Java utility `MdlMergeApp` which merges the attributes of two Simulink MDL files. Whereas one file serves as the actually model, the other only contains the residual MDL attributes for types. Keeping the two separate has the advantage that the model can be modified without already specified type information being lost.

We finally shall point out that no verification of code of the SDR diagram has been attempted so far, but the construction of the formal model paves the way for future work on this, including other kinds of formal analysis.

5 Conclusion and Future Work

In this paper we have reported on several extensions to `ClawZ` that pave the way for its use for generating formal models of control laws typically found in communications and signal-processing devices. It is an application domain that so far could not take advantage of the `ClawZ` tool support, and our experience suggests it being possible to apply `ClawZ` for such systems too without incurring fundamental changes to its underlying implementation and architecture.

The extensions entailed the introduction of new data types and additions to the library of supported blocks. To evaluate the additions, we animated respective block schemas using the `Z` animator `Jaza` [13,12]. This has been done, where possible, for the formalisation of complex arithmetics, complex vector and matrix operations, and importantly high-level specifications of blocks. In some cases definitions had to be rewritten to be processed by the `Jaza` tool; for instance, `Jaza` generally does not allow axiomatic definitions, hence functions such as `IIR1` had to be rewritten into schemas while setting their parameters to constant values. Further, inductive definitions had to be suitably unfolded.

Further validation took place in the automatic construction of a formal model for the non-trivial control law of a software-defined radio. This case study provided an initial motivation and benchmark for what support may be required for communications applications, but also highlighted imminent problems in using `ClawZ` in this context. A particular advantage of `ClawZ` is that it reduces the user interaction in the verification process, and the generation of `Z` models for diagrams is an aspect which can be entirely automated. On the other hand, the polymorphism of blocks puts limitations on automation in the current translation approach. A solution to this problem is either to alter the translation strategy by formally embedding polymorphic block behaviours, or, as we did, introduce additional steps that inject type information.

Future work consists of first developing a more comprehensive coverage of blocks from the Communications Toolbox of Simulink, and testing our extensions with a larger collection of diagrams of real-world applications. Although our current theories already provide support for many of the blocks, there are, for instance, gaps in supporting the various modulations techniques.

A second major area of follow-up work is to examine the verification of code within the new settings. ClawZ provides a powerful universal proof tactic (Supertac) to discharge verification conditions arising in the verification of code, and a process called ‘witnessing’ conducts the proof in an incremental manner as to increase the success of the automatic proof steps [219]. It is likely that the proof tactic will have to be adjusted to take full advantage of automation.

Finally, a third desirable extension is to implement a tool that automatically injects type information into Simulink models which then, as previously explained, can be exploited by ClawZ in constructing Z models. For this the control law at first must be typechecked, and secondly type information has to be written back into the MDL file. We have developed a Java component library that parses and processes MDL files, making the latter trivial. The implementation of a typechecker for control laws is pending work, but is not challenging.

Acknowledgements. We are grateful to QinetiQ for making their ClawZ tool suite available. Especially, we like to thank Collin O’Halloran and Nick Tudor for their consultation and involvement. We also would like to thank EPSRC for funding this work as part of the research grant EP/E025366/1.

References

1. Lemma 1. ProofPower and ProofPowerZ (1984–2009), <http://www.lemma-one.com/ProofPower/index/index.html>
2. Adams, M., Clayton, P.: ClawZ: Cost-Effective Formal Verification of Control Systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 465–479. Springer, Heidelberg (2005)
3. Arthan, R.: On Formal Specification of a Proof Tool. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 551, pp. 356–370. Springer, Heidelberg (1991), Technical report, <http://www.lemma-one.com/ProofPower/papers>
4. Arthan, R., Caseley, P., O’Halloran, C., Smith, A.: ClawZ: Control laws in Z. In: Third International Conference on Formal Engineering Methods (ICFEM), September 2000, pp. 169–176. IEEE Computer Society Digital Library (2000)
5. Bletsis, K.: Software Defined Radio (July 2002), Simulink Model, <http://www.mathworks.com/matlabcentral/fileexchange/1987>
6. Public Safety Special Interest Group. Software Defined Radio Technology for Public Safety. Technical report (April 2006), <http://www.ece.vt.edu/swe/chamrad/psi/SDRF-06-A-0001-V0.00.pdf>
7. Haykin, S.: Adaptive Filter Theory. Prentice Hall Information and System Sciences Series. Prentice Hall, Englewood Cliffs (2001)
8. Jondral, F.: Software-Defined Radio — Basics and Evolution to Cognitive Radio. Journal of Wireless Communications and Networking 2005(4), 275–283 (2005)
9. QinetiQ Ltd., 85 Buckingham Gate, London SW1E 6BP, UK. ClawZ Toolset User Guide (2007); Draft document for version 2.2.alpha6 of ClawZ
10. Reeds, J.: Software Radio: A Modern Approach to Radio Engineering. Communications Engineering and Emerging Technologies Series. Prentice Hall, Englewood Cliffs (2002)
11. Inc. The MathWorks. Simulink ® (1994–2008)

12. Utting, M.: Data Structures for Z Testing Tools. In: The 4th Workshop on Tools for Systems Design and Verification (July 2000), <http://www.cs.waikato.ac.nz/~marku/jaza>
13. Utting, M.: Jaza User Manual and Tutorial (June 2005), <http://www.cs.waikato.ac.nz/~marku/jaza/userman.pdf>
14. Vernon, M.: The Modelling and Verification of Software-Defined Radio Techniques in Communication Applications. Master's thesis, University of York, Heslington, York, YO10 5DD, UK (May 2008)


Formalising and Validating RBAC-to-XACML Translation Using Lightweight Formal Methods

Mark Slaymaker, David Power, and Andrew Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
mark.slaymaker@comlab.ox.ac.uk, david.power@comlab.ox.ac.uk,
andrew.simpson@comlab.ox.ac.uk

Abstract. The topic of *access control* has received a new lease of life in recent years as the need for assurance that the *correct* access control policy is in place is seen by many as crucial to providing assurance to individuals that their data is being treated appropriately. This trend is likely to continue with the increase in popularity of social networking sites and shifts to ‘cloud’-like commercial services: in both contexts, a clear statement of “who can do what” to one’s data is key in engendering trust. While approaches such as role-based access control (RBAC) provide a degree of abstraction, therefore increasing manageability and accessibility, policy languages such as the XML-based XACML provide greater degrees of expressibility—and, as a result, increased complexity. In this paper we explore how the mutual benefits of both RBAC and XACML, and Alloy and Z, may be used to best effect. RBAC is used as an accessible conceptual model; XACML is used as a language of implementation. Our concern is to facilitate the construction and reuse of role-based policies, which may subsequently be deployed in terms of XACML. We wish to provide assurance that these representations and transformations are, in some sense, correct. To this end, we consider formal models of both RBAC and XACML in terms of Z. We also describe how we have taken initial steps in utilising the Alloy Analyzer tool to provide a level of assurance that the two representations are consistent.

1 Introduction

Increasing amounts of data are being collected on all of us with respect to our different roles: teams are determining how best to utilise this data to turn us into more profitable consumers, to predict health problems before they manifest themselves, and to identify individuals likely to compromise national security. As data collection increases, and more interesting ways to utilise this data emerge, so do concerns about privacy: the question of who can see what, and under what conditions, is an important one in such contexts. In recent years, the XML-based XACML (eXtensible Access Control Markup Language)  has emerged as the *de facto* standard for policy capture and enforcement in service-oriented systems.

¹ See http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

XACML offers a high degree of expressiveness—but a consequence of XACML’s flexibility is complexity: even simple combinations of policies can run to tens of pages as a result of XML’s verbosity. In parallel, role-based access control (RBAC) [4] has emerged as a model of access control that makes authorisation policies more accessible and manageable: permissions are associated with roles, and, in turn, roles are associated with users. While RBAC benefits from conceptual simplicity, it suffers from a lack of expressiveness (extensions represented by, for example, [15] and [9] notwithstanding)—meaning that data owners can be restricted with respect to the types of policies that they may construct.

Our focus is not the capture of role-based policies in XACML—there is, after all, an RBAC profile for XACML—nor is it the development of formal models of access control, which has a rich history (see, for example, [14]). Rather, it is combining the benefits of RBAC and XACML to best effect: by utilising a simple, conceptual model for accessibility and assurance, prior to mapping to XACML for deployment. Formal metamodels of RBAC and XACML, given in terms of Z [12], are used to give confidence as to the transformation process; formal analysis is undertaken via the Alloy Analyzer [8].

We adopt ‘Spivey-style’ Z [12], which gives rise to the possibility of utilising the type checker *fuzz* [13]. We have consciously taken a ‘hybrid’ approach in our use of Z and Alloy as we are keen to leverage their complementary benefits: a long-term goal of our work is to allow the construction and analysis of tools by end-users, and Z, due to its accessibility and the fact that it provides a natural means of representing relational structures, is an appropriate candidate for an initial prototype language for policy capture; and as the Alloy Analyzer has emerged as the pre-eminent means of simulating and checking state-based descriptions, it is an obvious candidate to provide support for assurance. (While some may argue that Alloy meets all of these *desiderata*, our experience is that, in general, Z is more accessible to ‘the layman’; hence its use in our work.)

Our work is driven by practical concerns: we wish to offer policy writers a means of capturing expressive access control policies, with some degree of assurance that these policies are, in some sense, ‘correct’. Our middleware framework *sif* (for service-oriented interoperability framework) [11] facilitates the sharing and aggregating of data from disparate data sources. Via XACML, fine-grained access control is supported. While, currently, these policies are created via primitive policy editors, the ‘vision’ is that data owners will construct abstract policies via tool support, with validated formal representations being transformed into executable policies. This paper represents continued work along this path.

2 Formalising RBAC

Role-based access control (RBAC) is concerned with the relationships between users and roles, and roles and permissions. A generalised RBAC model was proposed in [3], from which the following important definitions may be taken.

1. For each subject, the active role is the one that the subject is currently using.
2. Each subject may be authorised to perform one or more roles.
3. Each role may be authorised to perform one or more transactions.

There are four components in the ANSI standard for RBAC: core RBAC, role hierarchies, and dynamic and static separation of duty. We consider only the first (mandatory) component: core RBAC, which relates permissions, roles and users: permissions are associated with roles, and roles are assigned to users. (It is worth noting, however, that in [10], it is shown how a more complex, hierarchical RBAC representation might be ‘normalised’ to an equivalent representation in core RBAC.) While other characterisations, extensions and notations for role-based access control (such as, for example, OrBAC [1]) exist, we choose to build upon an existing formal model of the ANSI standard: that of [10]. It should also be noted that issues such as administration of policies are not of concern here.

First, we introduce the basic types *User*, *Role*, *Action* and *Resource*, and characterise a permission as an action-resource pair:

$$\begin{aligned} & [User, Role, Action, Resource] \\ & PRMSBase == Action \times Resource \end{aligned}$$

The core RBAC system, captured by the schema *Core*, consists of the relations *UA* and *PA*, which represent the relationships that hold between users and roles, and roles and permissions respectively. These relations are restricted to the members of *USERS*, *ROLES* and *PRMS*, which represent the sets of current users, roles and permissions, and form part of the *Core* schema.

$$\begin{aligned} Core \hat{=} & [UA : User \leftrightarrow Role; PA : Role \leftrightarrow PRMSBase; \\ & USERS : \mathbb{P} User; ROLES : \mathbb{P} Role; PRMS : \mathbb{P} PRMSBase | \\ & UA \in USERS \leftrightarrow ROLES \wedge PA \in ROLES \leftrightarrow PRMS] \end{aligned}$$

3 Formalising XACML

XACML is an OASIS standard that defines two XML-based languages: the policy language is used to describe general access control requirements; the request/response language allows one to construct a query to determine whether a particular action should be permitted. The policy language, which is of concern to us, has standard extension points that allow one to define aspects such as new functions, data types, and logics to combine such entities.

An XACML policy can have any number of *rules*. At the heart of most rules is a *condition*, which is a Boolean function. If the condition evaluates to true, then the rule’s *effect* (either ‘permit’ or ‘deny’) is returned. A rule specifies a *target*, which defines: the set of the *subjects* who can access the resource; and the *resource* that the subject can access; the *action* that the subject can undertake on the resource; the *environmental* attributes that are relevant to an authorisation decision and are independent of a particular subject, resource or action.

The values of the request attributes are compared with those of the policy document so that a decision can be made with respect to access permission. When many different policies exist, a *policy set document* is defined as a combining set of many policies.

A policy or policy set may contain multiple rules or policies, each of which may evaluate to different authorisation decisions. In order for a final decision to be made, combining algorithms are used, with *policy combining algorithms* being used by policy sets and *rule combining algorithms* being used by policies. Each algorithm represents a different means of combining multiple decisions into a single authorisation decision. The authorisation decision in relation to a subject requesting permission for an action on a resource in an environment can take one of four values: ‘permit’, ‘deny’, ‘indeterminate’, and ‘not applicable’.

When the policy decision point (PDP) compares the attribute values contained in the request document with those contained in the policy or policy set document, a response document is generated. The response document includes an answer containing the authorisation decision. This result, together with an optional set of *obligations*, is returned to the policy enforcement point (PEP) by the PDP. Obligations are sets of operations that must be performed by the PEP in conjunction with an authorisation decision; an obligation may be associated with a positive or negative authorisation decision.

We start by introducing our basic types, which represent sets of unique identifiers for policy sets, policies, rules, targets, and requests. Additionally, we introduce *Environment*, which contains information pertaining to the operating environment, and *Obligation*, which contains those activities that the PEP must undertake before access is permitted. As these entities have little relevance to our discourse, it is appropriate to abstract away from their details in our model.

[*PolicySetID*, *PolicyID*, *RuleID*, *TargetID*, *RequestID*, *Environment*, *Obligation*]

There is a special *TargetID* associated with the empty target:

| *empty_target* : *TargetID*

We use *Subject* as an abbreviation for *User* (subjects are XACML’s equivalent of users).

Subject == *User*

Next, the free type *PolicyRef* is introduced, which deals with the fact that a policy set can reference instances of both policies and policy sets. To this end, the constructors take elements of *PolicyID* and *PolicySetID* as arguments.

PolicyRef ::= *Pol*⟨⟨*PolicyID*⟩⟩ | *PolSet*⟨⟨*PolicySetID*⟩⟩

It is necessary to introduce two free types to handle the evaluation of elements. First, *Effect* contains the possible effects that can be returned from the evaluation of rules, policies and policy sets. Second, *EvalRes* reflects the evaluation of a condition within the context of a specific request, as well as the possible outcomes of a match operation between a request and a target. In addition, rules and policies are associated with particular subsets of *Effect*.

$$\begin{aligned}
\textit{Effect} & ::= \textit{Permit} \mid \textit{Deny} \mid \textit{NotApplicable} \mid \textit{Indeterminate} \mid \\
& \quad \textit{IndeterminateDeny} \mid \textit{IndeterminatePermit} \\
\textit{EvalRes} & ::= \textit{TRUE} \mid \textit{FALSE} \mid \textit{INDETERMINATE} \\
\textit{EffectRule} & == \textit{Effect} \setminus \{ \textit{Indeterminate} \} \\
\textit{EffectPolicy} & == \textit{Effect} \setminus \{ \textit{IndeterminateDeny}, \textit{IndeterminatePermit} \}
\end{aligned}$$

Our final free types introduce the identifiers for the different rule and policy combining algorithms. Only the basic combining algorithms of the XACML specification are considered here. It would, however, be perfectly possible to define any number of other combining strategies to suit a given situation.

$$\begin{aligned}
\textit{RulComAlgID} & ::= \textit{rulPermitOverRide} \mid \textit{rulDenyOverRide} \mid \textit{rulFirstApplicable} \\
\textit{PolComAlgID} & ::= \textit{polPermitOverRide} \mid \textit{polDenyOverRide} \mid \textit{polFirstApplicable}
\end{aligned}$$

A request is modelled as consisting of a request identifier, together with sets made up of elements from the types *Subject*, *Action*, *Resource*, and *Environment*. It is worth noting that *Action* has been modelled as a set even though it is constrained to be an atomic value: this is done to allow a generic function to be defined to evaluate all the parts of a *request* against their corresponding parts in a *Target*. The request, then, effectively represents a subject asking to perform an action on a resource and also captures any relevant environment information.

$$\begin{aligned}
\textit{Request} \hat{=} & [\textit{request} : \textit{RequestID}; \textit{sub} : \mathbb{P}_1 \textit{Subject}; \\
& \quad \textit{act} : \mathbb{P}_1 \textit{Action}; \textit{res} : \mathbb{P}_1 \textit{Resource}; \textit{env} : \mathbb{P} \textit{Environment} \mid \\
& \quad \# \textit{act} = 1 \wedge \# \textit{env} \leq 1]
\end{aligned}$$

The *Target* schema is an abstraction of the Target element of the XACML specification, which is described as: “a Target is a conjunctive sequence of Subjects, Resources, Actions and Environments.” As the other elements behave in much the same way, it suffices to discuss only subjects in the following.

A *Subjects* element is described as a disjunctive sequence of *Subject* elements, with a *Subject* element being a conjunctive sequence of *SubjectMatch* elements. In turn, the *SubjectMatch* element defines a matching function and the element in the request context that it should be applied to.

We have chosen to model the *Subject*, *Action*, *Resource* and *Environment* elements within the *Target* schema as functions mapping members of the appropriate type to elements of *EvalRes*. Each of the sections of an XACML target—Subjects, Actions, Resources and Environments—is represented in the model by a sequence of these functions. For example, the functions in the sequence *sub* each represent an XACML Subject element. The function is effectively abstracting away the conjunctive sequence of *SubjectMatch* elements. It is worth noting that we have chosen to use sequences to model the various components of *Target* as this simplifies the construction of functions that are used to evaluate the matching of requests to targets in the full model.

$$\begin{aligned}
\textit{Target} \hat{=} & [\textit{tid} : \textit{TargetID}; \\
& \quad \textit{sub} : \text{seq}(\textit{Subject} \rightarrow \textit{EvalRes}); \textit{act} : \text{seq}(\textit{Action} \rightarrow \textit{EvalRes}); \\
& \quad \textit{res} : \text{seq}(\textit{Resource} \rightarrow \textit{EvalRes}); \textit{env} : \text{seq}(\textit{Environment} \rightarrow \textit{EvalRes})]
\end{aligned}$$

A rule is composed of: an identifier, which we denote as *rid*; a target for which the rule is applicable; a condition that is evaluated in relation to a given request; and an effect that is returned if the condition associated with the rule evaluates to true. The effect can only have the values *permit* or *deny*: if the condition evaluates to false or the target is not matched, then the rule will return an effect of *NotApplicable*; if the evaluation were to fail for any reason, then a value of *Indeterminate* will be returned.

$$\begin{aligned} \text{Rule} \hat{=} & [\text{rid} : \text{RuleID}; \text{target} : \text{TargetID}; \\ & \text{condition} : \text{seq}(\text{Request} \rightarrow \text{EvalRes}); \text{effect} : \text{EffectRule} \mid \\ & \text{effect} \in \{\text{Permit}, \text{Deny}\} \wedge \# \text{condition} \leq 1] \end{aligned}$$

A *policy* consists of several aspects, namely: an identifier, *pid*, to uniquely identify it; a target for which the policy is applicable; a sequence of *RuleIDs*; and a rule combining algorithm, defining how the effects of the sequence of rules should be combined. Finally, the *Policy* schema also contains a (possibly empty) set of obligations. Note that it is possible (albeit undesirable) for *inPol* to be empty as the XACML standard does not preclude this. Further, it is also possible for the same rule identifier to appear more than once within *inPol*.

$$\begin{aligned} \text{Policy} \hat{=} & [\text{pid} : \text{PolicyID}; \text{target} : \text{TargetID}; \\ & \text{inPol} : \text{seq} \text{RuleID}; \text{rca} : \text{RulComAlgID}; \text{obli} : \mathbb{P} \text{Obligation}] \end{aligned}$$

The *PolicySet* schema consists of five components. First, *psid* acts as a unique identifier for a policy set. Second, *target* is the target for which the policy set is applicable. Next, *pca* identifies the relevant policy combining algorithm, i.e. it determines how the effects of the sequence of policies and (further) policy sets associated with a policy set are combined. The sequence of elements of type *PolicyRef*—constructed from elements of both *PolicyID* and *PolicySetID*—allows a particular policy set to effectively contain a sequence of both policies and policy sets. Finally, a policy set may contain an optional set of obligations.

$$\begin{aligned} \text{PolicySet} \hat{=} & [\text{psid} : \text{PolicySetID}; \text{target} : \text{TargetID}; \\ & \text{pca} : \text{PolComAlgID}; \text{inPolSet} : \text{seq} \text{PolicyRef}; \text{obli} : \mathbb{P} \text{Obligation}] \end{aligned}$$

Again, it is possible, albeit undesirable, for *inPolSet* to be empty. Note that *inPolSet* (and, indeed, *inPol* in *Policy*) are defined as sequences, rather than sets, to deal with those cases (such as the application of the first applicable combining algorithm) in which the order of rules (or policy references) is important.

Although the repetition of elements of both *PolicyID* and *PolicySetID* within a policy set is permissible, repetition of elements of the latter is problematic. This is due to the fact that such a situation might result in a recursive referencing of policy sets. If such a policy were to be implemented, this cycle of references might lead to an infinitely recursing process. Of course, one of the benefits afforded by a formal model is that we have the potential to check for the absence of such cycles, as well as other healthiness conditions that we might wish to enforce.

While a number of optional elements for policies and policy sets exist in the specification, we have omitted these elements from this description as they add little of value to the narrative.

We now define *XACML* as a collection of functions that map identifiers to instances of policy sets, policies, and so on. The constraints on the functions serve to ensure that each function maps an identifier to a binding containing the same identifier. In addition, we define *rootPol*, which represents the top level policy set or policy to be evaluated to check for applicability to any particular request.

$\begin{array}{l} \textit{XACML} \\ \hline \textit{getPolicySet} : \textit{PolicySetID} \mapsto \textit{PolicySet}; \textit{getPolicy} : \textit{PolicyID} \mapsto \textit{Policy} \\ \textit{getRule} : \textit{RuleID} \mapsto \textit{Rule}; \textit{getTarget} : \textit{TargetID} \mapsto \textit{Target} \\ \textit{rootPol} : \mathbb{P} \textit{PolicyRef} \\ \hline (\forall \textit{psi} : \textit{PolicySetID} \mid \textit{psi} \in \text{dom } \textit{getPolicySet} \bullet (\textit{getPolicySet } \textit{psi}).\textit{psid} = \textit{psi}) \\ (\forall \textit{pi} : \textit{PolicyID} \mid \textit{pi} \in \text{dom } \textit{getPolicy} \bullet (\textit{getPolicy } \textit{pi}).\textit{pid} = \textit{pi}) \\ (\forall \textit{ri} : \textit{RuleID} \mid \textit{ri} \in \text{dom } \textit{getRule} \bullet (\textit{getRule } \textit{ri}).\textit{rid} = \textit{ri}) \\ (\forall \textit{ti} : \textit{TargetID} \mid \textit{ti} \in \text{dom } \textit{getTarget} \bullet (\textit{getTarget } \textit{ti}).\textit{tid} = \textit{ti}) \end{array}$
--

4 From RBAC to XACML

In this section we define the functions necessary to convert an access control policy defined in core RBAC into an equivalent policy in XACML.

4.1 Helper Functions

The first helper function we will consider is the *compactXACML* function, which takes an arbitrary number of *XACML* instances and combines them into a single *XACML* instance. The constraints on the functions *getPolicySet*, *getPolicy*, *getRule* and *getTarget* ensure that the result of each pair-wise union are contained within the appropriate function definitions. Additionally, the final five clauses ensure the overall functions contain the result of the union of all of the contributing functions.

$\begin{array}{l} \textit{compactXACML} : \mathbb{P} \textit{XACML} \mapsto \textit{XACML} \\ \hline \forall s : \mathbb{P} \textit{XACML} \mid \\ (\forall x1, x2 : s \bullet \\ \quad x1.\textit{getPolicySet} \cup x2.\textit{getPolicySet} \in \textit{PolicySetID} \mapsto \textit{PolicySet} \wedge \\ \quad x1.\textit{getPolicy} \cup x2.\textit{getPolicy} \in \textit{PolicyID} \mapsto \textit{Policy} \wedge \\ \quad x1.\textit{getRule} \cup x2.\textit{getRule} \in \textit{RuleID} \mapsto \textit{Rule} \wedge \\ \quad x1.\textit{getTarget} \cup x2.\textit{getTarget} \in \textit{TargetID} \mapsto \textit{Target}) \bullet \\ \quad (\textit{compactXACML } s).\textit{getPolicySet} = \bigcup \{x : s \bullet x.\textit{getPolicySet}\} \wedge \\ \quad (\textit{compactXACML } s).\textit{getPolicy} = \bigcup \{x : s \bullet x.\textit{getPolicy}\} \wedge \\ \quad (\textit{compactXACML } s).\textit{getRule} = \bigcup \{x : s \bullet x.\textit{getRule}\} \wedge \\ \quad (\textit{compactXACML } s).\textit{getTarget} = \bigcup \{x : s \bullet x.\textit{getTarget}\} \wedge \\ \quad (\textit{compactXACML } s).\textit{rootPol} = \bigcup \{x : s \bullet x.\textit{rootPol}\} \end{array}$

We assume the existence of functions capable of generating the various identifiers required: *genTid*, *genTid1*, *genTid2*, *genRid*, *genPid*, and *genPSid*. The

three functions that generate a *TargetID* (*genTid*, *genTid1* and *genTid2*) are constrained to produce disjoint sets of identifiers. (Due to space limitations, the constraints are omitted.)

$$\begin{array}{l} \text{genTid} : \mathbb{P} \text{User} \times \text{Role} \times \mathbb{P} \text{PRMSBase} \mapsto \text{TargetID} \\ \text{genTid1} : \text{PRMSBase} \mapsto \text{TargetID} \\ \text{genTid2} : \mathbb{P}(\mathbb{P} \text{User} \times \text{Role} \times \mathbb{P} \text{PRMSBase}) \mapsto \text{TargetID} \\ \text{genRid} : \text{PRMSBase} \mapsto \text{RuleID} \\ \text{genPid} : \mathbb{P} \text{User} \times \text{Role} \times \mathbb{P} \text{PRMSBase} \mapsto \text{PolicyID} \\ \text{genPSid} : \mathbb{P}(\mathbb{P} \text{User} \times \text{Role} \times \mathbb{P} \text{PRMSBase}) \mapsto \text{PolicySetID} \end{array}$$

The next helper function is a generic function that converts a set to a sequence containing the same elements as the original set. This function is required for creating the sequences necessary for certain aspects of the XACML model: in particular, for creating the sequences that make up *inPol* and *inPolSet*.

$$\begin{array}{l} \text{sequence} : \mathbb{P} X \rightarrow \text{seq } X \\ \forall xs : \mathbb{P} X \bullet \# \text{sequence}(xs) = \#xs \wedge \text{ran } \text{sequence}(xs) = xs \end{array}$$

We now define the generic function *targetElementEquals*, which is used to define the functions that will be used in *Target*. The function returns a function that has the property that it returns *FALSE* for all input values—other than for the input *x*, for which it returns *TRUE*.

$$\begin{array}{l} \text{targetElementEqual} : X \rightarrow X \rightarrow \text{EvalRes} \\ \forall x, y : X \bullet \\ \quad x = y \Rightarrow \text{targetElementEqual } x \ y = \text{TRUE} \\ \quad \wedge \\ \quad x \neq y \Rightarrow \text{targetElementEqual } x \ y = \text{FALSE} \end{array}$$

4.2 Translation

The following four functions define the translation process that converts an element of *Core*, which is the RBAC representation of an access control policy, into an element of *XACML*. We initially define the creation of rules, and describe how these are combined into policies. These policies are then appropriately combined into an overall XACML representation of the original RBAC policy.

The *makeRule* function defines how a *PRMSBase* from an instance of *Core* is used to generate a fragment of XACML consisting of only a *Rule*, together with an appropriate rule identifier. The action and resource parts of the *PRMSBase* are used as the parameters for *targetElementEqual* to define the match functions for the action and resource parts of the target *t.act* and *t.res* respectively. The other elements of the target, *t.sub* and *t.env*, are associated with empty sequences. The identifiers *t.tid* and *r.rid* are generated by *genTid1* and *genRid*

each taking the supplied *PRMSBase* as the input parameter. The target for the rule is set to the target defined, the condition is set to an empty sequence as this evaluates to true, and the effect is set to *Permit*. Finally, the constraints ensure the relevant mappings are contained in the lookup functions of the *XACML* instance being generated. As this is only a small fragment of the overall *XACML* policy being generated, there is no need to define a root policy set in *x.rootpol*.

$$\begin{array}{l}
 \hline
 \text{makeRule} : \text{PRMSBase} \rightarrow (\text{XACML} \times \text{RuleID}) \\
 \hline
 \forall \text{prm} : \text{PRMSBase} \bullet \exists x : \text{XACML}; t : \text{Target}; r : \text{Rule} \bullet \\
 \text{makeRule prm} = (x, r.\text{rid}) \wedge \\
 t.\text{tid} = \text{genTid1 prm} \wedge t.\text{sub} = \langle \rangle \wedge t.\text{env} = \langle \rangle \wedge \\
 t.\text{act} = \langle \text{targetElementEqual}(\text{first prm}) \rangle \wedge \\
 t.\text{res} = \langle \text{targetElementEqual}(\text{second prm}) \rangle \wedge \\
 r.\text{rid} = \text{genRid prm} \wedge r.\text{target} = t.\text{tid} \wedge \\
 r.\text{condition} = \langle \rangle \wedge r.\text{effect} = \text{Permit} \wedge \\
 x.\text{getPolicySet} = \emptyset \wedge x.\text{getPolicy} = \emptyset \wedge \\
 x.\text{getRule} = \{r.\text{rid} \mapsto r\} \wedge x.\text{getTarget} = \{t.\text{tid} \mapsto t\} \wedge x.\text{rootPol} = \emptyset
 \end{array}$$

The next function, *makePolicy*, utilises *makeRule* when building the overall policy that is constructed from a particular RBAC role. It takes a tuple containing the set of users associated with a role, the role itself and the set of permissions associated with the role, and generates pairs consisting of an instance of *XACML* along with the element of *PolicyID* relating to the newly defined policy. The only parts of the target that are of interest are the target identifier, *t.tid*, and the target subject matching functions, *t.sub*, which is a sequence of functions that will match any of the users associated with the role. A set of these functions, one for each user, is created and converted into a sequence to match the *XACML* definition. Similarly, *r.inPol* is generated by creating a set of rules relating to each of the *PRMSBase* values and using the sequence function to convert the set into a sequence. The element of *XACML* generated is the result of *compactXACML* being applied to the set of rules generated and then being applied to the resulting element of *XACML* and the policy *XACML* fragment.

$$\begin{array}{l}
 \hline
 \text{makePolicy} : (\mathbb{P} \text{User} \times \text{Role} \times \mathbb{P} \text{PRMSBase}) \rightarrow (\text{XACML} \times \text{PolicyID}) \\
 \hline
 \forall us : \mathbb{P} \text{User}; r : \text{Role}; \text{prm} : \mathbb{P} \text{PRMSBase} \bullet \\
 \exists x1, x2 : \text{XACML}; t : \text{Target}; p : \text{Policy} \bullet \\
 \text{makePolicy}(us, r, \text{prm}) = (\text{compactXACML}\{x1, x2\}, p.\text{pid}) \wedge \\
 t.\text{tid} = \text{genTid}(us, r, \text{prm}) \wedge \\
 t.\text{act} = \langle \rangle \wedge t.\text{res} = \langle \rangle \wedge t.\text{env} = \langle \rangle \wedge \\
 t.\text{sub} = \text{sequence} \{u : us \bullet \text{targetElementEqual } u\} \wedge \\
 p.\text{pid} = \text{genPid}(us, r, \text{prm}) \wedge p.\text{target} = t.\text{tid} \wedge \\
 p.\text{inPol} = \text{sequence}(\text{ran}\{\text{perm} : \text{prm} \bullet \text{makeRule perm}\}) \wedge \\
 p.\text{rca} = \text{rulPermitOverRide} \wedge p.\text{obli} = \emptyset \wedge \\
 x1 = \text{compactXACML}(\text{dom}\{\text{perm} : \text{prm} \bullet \text{makeRule perm}\}) \wedge \\
 x2.\text{getPolicySet} = \emptyset \wedge x2.\text{getPolicy} = \{p.\text{pid} \mapsto p\} \wedge \\
 x2.\text{getRule} = \emptyset \wedge x2.\text{getTarget} = \{t.\text{tid} \mapsto t\} \wedge x2.\text{rootPol} = \emptyset
 \end{array}$$

It is now possible to define the function *makeXACML*, which takes a set of roles and generates an equivalent element of *XACML*. The interesting aspects

are the generation of a policy set from the policies generated; in addition, the policy set is the compaction of the policies and the policy set defined.

$$\begin{array}{l}
\hline
\text{makeXACML} : \mathbb{P}(\mathbb{P} \text{ User} \times \text{Role} \times \mathbb{P} \text{ PRMSBase}) \rightarrow \text{XACML} \\
\hline
\forall \text{ params} : \mathbb{P}(\mathbb{P} \text{ User} \times \text{Role} \times \mathbb{P} \text{ PRMSBase}) \bullet \\
\exists x1, x2 : \text{XACML}; t : \text{Target}; ps : \text{PolicySet} \bullet \\
\text{makeXACML}(\text{params}) = \text{compactXACML}\{x1, x2\} \wedge \\
t.\text{tid} = \text{genTid2 params} \wedge \\
t.\text{act} = \langle \rangle \wedge t.\text{res} = \langle \rangle \wedge t.\text{sub} = \langle \rangle \wedge t.\text{env} = \langle \rangle \wedge \\
ps.\text{psid} = \text{genPSid params} \wedge ps.\text{target} = t.\text{tid} \wedge \\
ps.\text{pca} = \text{polPermitOverRide} \wedge ps.\text{obli} = \emptyset \wedge \\
ps.\text{inPolSet} = \text{sequence}(\{\text{prm} : \text{params} \bullet \text{Pol}(\text{second}(\text{makePolicy prm}))\}) \wedge \\
x1 = \text{compactXACML}(\text{dom}\{\text{prm} : \text{params} \bullet \text{makePolicy prm}\}) \wedge \\
x2.\text{getPolicySet} = \{ps.\text{psid} \mapsto ps\} \wedge \\
x2.\text{getPolicy} = \emptyset \wedge x2.\text{getRule} = \emptyset \wedge \\
x2.\text{getTarget} = \{t.\text{tid} \mapsto t\} \wedge x2.\text{rootPol} = \{\text{PolSet } ps.\text{psid}\}
\end{array}$$

5 Towards an Alloy Model for Formal Analysis

In this section we take the Z models of RBAC and XACML, along with the translation and helper functions, and produce a corresponding Alloy model. We have produced a more abstract representation of the Z schemas and functions for use with the Alloy Analyzer. We have been able to make the Alloy model of XACML more abstract than the Z model because we are only considering XACML policies that are the result of translating from an RBAC policy. This has allowed us to simplify the XACML structure from rules, policies and policy sets by collapsing it into a simple set of policies. This is possible because the policy set defined by the translation has a target that matches any request, additionally we are only considering a single policy set and are utilising the permit override combining algorithm—so for the purposes of the simplified Alloy model we only need to consider a single set of policies.

Each policy is modelled as a set of users who have a particular role and the set of permissions that the same role has. This is equivalent to combining the rules within a policy into a single rule, which is then mapped into a policy by adding the subject (user) information of the containing policy. With this compression, each Alloy policy in the translation represents a single role, containing the details of the users that are associated with that role and the permissions that role gives to act on particular resources.

The Alloy model consists of a number of modules. The `accesscontrol/types` module defines the signatures and facts that are the main primitives which are used for defining the core RBAC and XACML models. First we define `EvalRes` and then define possible values by extension. The `one sig` (for ‘signature’) decoration on the definitions of `Permit` and `Deny` ensures that any reference to `Permit` (respectively `Deny`) will be the same instance. Next, `User`, `Role`, `Action` and `Resource` are defined—with `Action` and `Resource` being used to define the contents of a `PRMSBase`. A request is then defined as a coupling of an element

of `User` and an element of `PRMSBase`, which effectively models a user requesting permission to perform an action on a resource. Finally, we add a fact that ensures each `PRMSBase` is unique and no two have the same action/resource pair.

```
module accesscontrol/types
abstract sig EvalRes {}
one sig Permit, Deny extends EvalRes {}
sig User, Role, Action, Resource {}
sig PRMSBase { action : Action, resource : Resource }
sig Request { u : User, p : PRMSBase }
fact uniquePRMSBase { all disj pb1, pb2 : PRMSBase |
    pb1.action != pb2.action || pb1.resource != pb2.resource }
```

The module `accesscontrol/rbac` is the RBAC model in Alloy, utilising the definitions of `accesscontrol/types`. The definition of `Core` is based on that of Section 2, where `UA` is a mapping between `Users` and `Roles`, `PA` is a mapping between `Roles` and `PRMSBase`, and `USERS`, `ROLES` and `PRMS` are subsets of `User`, `Role` and `PRMSBase` respectively. The relationships are constrained to ensure that the elements of `UA` and `PA` are within `USERS`, `ROLES` and `PRMS`, the set of roles in `UA` matches the set of roles in `PA` (`User.UA = PA.PRMSBase`), and `ROLES` is the set of `Roles` from `PA.PRMSBase`. The `evalRBAC` function takes elements of `Core` and `Request` as input, and returns `Permit` if the corresponding `User` and `PRMSBase` in the request exist in the product of `UA` and `PA` in the `Core` of the RBAC being used; otherwise it returns `Deny`.

```
module accesscontrol/rbac
open accesscontrol/types
sig Core { UA : User -> Role, PA : Role -> PRMSBase,
    USERS : set User, ROLES : set Role, PRMS : set PRMSBase }
{ UA in USERS -> ROLES && PA in ROLES -> PRMS
    User.UA = PA.PRMSBase && ROLES = PA.PRMSBase }
fun evalRBAC(rbac: Core, req : Request) : EvalRes {
    ((req.u -> req.p) in ((rbac.UA).(rbac.PA))) => Permit else Deny }
```

Next, we define a simplified representation of XACML in Alloy. As discussed at the start of this section, we are only concerned with policies (as opposed to policy sets) in this model as we can effectively combine the rules into a single policy. A `Policy` is simply defined as a set of `Users` who have a particular `Role` and the associated set of permissions. This is abstracted from a set of subjects that a policy is associated with which contains a number of rules which are matched by the action resource pair from the `PRMSBase`. The `Role` attribute acts as an identifier for a particular policy to ensure the correct grouping of `Users` and `PRMSBase` are maintained. XACML is then defined as a set of these policies. The `evalXACML` function returns `Permit` if the user and permission of the request match a user and permission in a policy; otherwise it returns `Deny`.

```
module accesscontrol/xacml
open accesscontrol/types
sig Policy { u : set User, r : Role, p : set PRMSBase }
sig XACML { policies : set Policy }
fun evalXACML(x: XACML, req : Request) : EvalRes {
    (some pol : x.policies |
        ((req.u -> req.p) in ((pol.u) -> (pol.p)))) => Permit else Deny }
```

We now define a number of facts and functions in Alloy that facilitate the translation of a policy represented as an RBAC `Core` into an equivalent policy

in XACML. The `makePolicy` function takes a `Core` and a `Role` as input, and generates a `Policy`, such that: `u` is the set of users that have the inputted `Role`; `p` is the set of `PRMSbase` elements associated with the `Role`; and `role` is set to be the supplied role as an identifier. We can then use this function in `makeXACML` to create a set of policies relating to each role in the supplied element of `Core`. The created set is then assigned to the `policies` field of an element of XACML. We then define a predicate that checks if the result of evaluating a request against an RBAC policy is the same as the result of applying the same request to the resulting XACML policy created by converting the RBAC policy. We have also defined facts that ensure that non-null policies and XACML exist.

```

module accesscontrol/rbactoxacml
open accesscontrol/rbac
open accesscontrol/xacml
fact PoliciesExist {
  all rbac : Core , rol : Role | one pol : Policy |
    pol.r = rol && pol.u = rbac.UA.rol && pol.p = rol.(rbac.PA) }
fun makePolicy(rbac : Core , rol : Role) : Policy {
  { pol : Policy | rol in rbac.ROLES && pol.r = rol &&
    pol.u = rbac.UA.rol && pol.p = rol.(rbac.PA) } }
fun makeXACML(rbac : Core ) : XACML {
  { x : XACML | x.policies = { pol : Policy |
    some rol : rbac.ROLES | pol = makePolicy[rbac, rol]} } }
fact XACMLexist {
  all rbac : Core | one x : XACML |
    x.policies = {pol : Policy | some rol : rbac.ROLES | pol = makePolicy[rbac, rol]} }
pred convert ( rbac : Core , req : Request ) {
  evalRBAC[rbac, req] = evalXACML[makeXACML[rbac], req] }

```

It is worth considering how Alloy treats functions. Consider the function `makePolicy` and the fact `PoliciesExist`. If this function were defined in `Z`, the declaration `pol : Policy` would indicate that `pol` ranges over every possible value of `Policy`—which is possibly infinite. In Alloy, `pol` would only range over the values of `Policy` contained within the instance being explored—which may not have a policy meeting the criteria. The fact `PoliciesExist` is a generator axiom which ensures that there is a policy within the instance that meets the criteria.

It is now possible to define an assertion that, for any RBAC policy defined as a `Core`, the result from any request will be the same as the result obtained from applying the request to the resulting translation to an XACML policy. The assertion uses the `convert` predicate to check the result from the evaluations match. The result of executing the assertion is shown below—indicating no counter-example was found.

```

module accesscontrol/rbactoxacmltest
open accesscontrol/rbactoxacml
assert eq { all rbac : Core , req : Request | convert [rbac, req] }
check eq for 5

```

Executing "Check eq for 5"

```

Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=20
12800 vars. 605 primary vars. 32832 clauses. 185ms.
No counterexample found. Assertion may be valid. 26629ms.

```

6 Discussion

The work described in this paper is driven by a long-term vision: we wish to balance the flexibility afforded by policy languages such as XACML with the manageability—provided via abstraction—afforded by the role-based paradigm. Further, we wish to provide policy writers with a means of establishing that the policies they write capture their intentions: as legislation increases in this area, this will become increasingly important. Our goal is assurance, rather than proof; hence the utilisation of a model checker such as the Alloy Analyzer tool (as opposed to, say, a theorem prover). In addition, there are many scenarios in which it is necessary to translate access control policies from one representation to another in a way that is guaranteed to preserve meaning—thereby giving rise to a need for formal semantics and a formal transformation process. Examples include migrating an existing access control policies into a new representation in a new system, and being able to compare the combined effect of policies written in different languages in a distributed, heterogeneous environment.

In this paper we have described the work undertaken in translating RBAC policies into (a formal representation of) XACML. To facilitate this, we have modelled RBAC and XACML using Z, and developed Z definitions of functions to perform the translation. In addition to defining the translation functions, we have also provided an abstract model in Alloy, which we have used to gain a level of assurance about our translations. Although we have only considered core RBAC, in [10] consideration is given as to how a hierarchal RBAC policy might be converted into a core RBAC policy; as such, we are confident that the techniques described in this paper can be extended in a straightforward fashion to deal with more complex cases.

Others within the community have considered formal descriptions of XACML (notably via VDM++ [2]) and used XACML as a target representation for analysis performed in RW [16]. In [7], the use of Alloy to verify a subset of the XACML language is described. In [5] the authors describe a tool—called Margrave—which translates access control policies written in XACML into Binary Decision Diagrams (BDDs). Finally, in [6]—a position paper—the authors propose an approach for conformance checking of XACML policies based on existing toolsets and techniques. Our focus in this paper has not been the formal representations of access control policies *per se*, but, rather, the consideration of how such models might form the basis for transformation.

There is much left to do in terms of marrying formality, expressibility and usability. One immediate area of future work involves extending the work of this paper to produce an XACML model capable of dealing with the various combining algorithms, as well as functions for translating hierarchal RBAC policies without having to flatten them into a core representation. On a more practical level, we have started the development of tool support for the construction of abstract representations of policies—which can then be formally analysed prior to mapping to XACML for deployment. Finally, our mapping from Z to Alloy is currently undertaken in a manual fashion—which has clear drawbacks; this process will need to be automated as we move forward with our work.

References

1. Abou El Kalam, A., El Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G.: Organization Based Access Control. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2003) (June 2003)
2. Bryans, J., Fitzgerald, J.S.: Formal Engineering of XACML Access Control Policies in VDM++. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 37–56. Springer, Heidelberg (2007), <http://dblp.uni-trier.de/db/conf/icfem/icfem2007.html#BryansF07>
3. Ferraiolo, D.F., Kuhn, D.R.: Role-based access control. In: Proceedings of the 15th National Computer Security Conference (1992)
4. Ferraiolo, D.F., Kuhn, D.R., Chandramouli, R.: Role-based access control. Artech House Publishers, Boston (2003)
5. Fislser, K., Krishnamurthi, S., Meyerovich, L., Tshantz, M.C.: Verification and change-impact analysis of access-control policies. In: Proceedings of ICSE 2005 (2005)
6. Hu, V.C., Martin, E., Hwang, J., Xie, T.: Conformance checking of access control policies specified in XACML. In: Proceedings of the 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007), Beijing, China, July 2007, pp. 275–280 (2007)
7. Hughes, G., Bultan, T.: Automated verification of XACML policies using a SAT solver. In: Proceedings of the Workshop on Web Quality, Verification and Validation, WQVV 2007 (2007)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
9. Kondo, S., Iwaihara, M., Yoshikawa, M., Torato, M.: Extending RBAC for large enterprises and its quantitative risk evaluation. In: The 8th IFIP conference on e-Business, e-Services, and e-Society, pp. 99–112 (2008)
10. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On formalising and normalising role-based access control systems. *The Computer Journal* 52(3), 303–325 (2009)
11. Simpson, A.C., Power, D.J., Russell, D., Slaymaker, M.A., Kouadri-Mostefaoui, G., Ma, X., Wilson, G.: A healthcare-driven framework for facilitating the secure sharing of data across organisational boundaries. *Studies in Health Technology and Informatics* 138, 3–12 (2008)
12. Spivey, J.M.: *The Z Notation: A Reference Manual*, 2nd edn. Prentice-Hall International, Englewood Cliffs (1992)
13. Spivey, J.M.: *The Fuzz Manual*, 2nd edn. (2000)
14. Stepney, S., Lord, S.P.: Formal specification of an access control system. *Software—Practice and Experience* 17(9), 575–593 (1987)
15. Swift, M.M., Brundrett, P., Van Dyke, C., Garg, P., Hopkins, A., Chan, S., Goertzel, M., Jensenworth, G.: Improving the granularity of access control in windows NT. In: Proceedings of the Sixth ACM symposium on Access control models and technologies (SACMAT 2001), pp. 87–96 (2001)
16. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems in XACML. In: FMSE 2004: Proceedings of the 2004 ACM workshop on Formal methods in security engineering, pp. 56–65. ACM Press, New York (2004)

Towards Formally Templated Relational Database Representations in Z

Nicolas Wu and Andrew Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{nicolas.wu, andrew.simpson}@comlab.ox.ac.uk

Abstract. Many authors have drawn parallels between the relational model of data and the formal description technique Z, yet none of these contributions have managed to be both close to the relational model in terms of providing a practical means of database design and fully formal in terms of providing an appropriate metamodel. We compare these various formalisms, and suggest how the use of the formal template approach of Amálio *et al* might help to overcome some of the issues faced. We demonstrate the application of this work via a short case study, and suggest further enhancements to the template language.

1 Introduction

The success of the relational model of data [1] may be attributed to the fact that it is easy to understand, and that the interaction with its databases through SQL is intuitive. The practical design of relational databases has traditionally focused on the use of approaches such as entity-relationship diagrams and, recently, UML. These methods are semi-formal: despite having a formal syntax, their semantics are loose, allowing for rapid prototyping and pragmatic development— aspects which have contributed to the success of UML in the wider software engineering context. However, since their semantics are not formalised, the models they produce can be ambiguous, possibly inconsistent, and cannot be analysed mechanically. Furthermore, the use of graphical notations means that the focus is often exclusively on the conceptual relationships between entities; issues pertaining to constraints, which strengthen those relationships, are often ignored. Constraints are important because databases represent some structure that has a certain constraining context (the ‘business rules’), and the database needs to faithfully represent that structure—yet this aspect is often overlooked in practice. Thus, the potential value of formal description techniques in the design of relational databases is clear: they provide a convenient means of expressing and reasoning about constraints early on in the development process.

We are concerned with the accurate representation of integrity constraints as pre- and post-conditions in a notation that supports reasoning about those constraints. Capturing integrity constraints is in itself useful, but doing so in a formal language that allows the mechanical analysis and manipulation of those

conditions has the potential to provide assurance of certain properties of the database at the design stage, rather than through consistency checks before transactions commit. The Z notation [2] is a respectable candidate as a means of working with constraints: Z and the relational model both have their roots in predicate calculus and set theory, with these similarities having been noted before [3]. Indeed, several authors have made use of this connection in an educational context [4,5], by leveraging not only the logical and philosophical common ground of the two notations, but also the structural similarities that they share.

Other formal languages have been investigated as tools to aid in the design and development of databases. Notably, Schewe *et al* [6], Hayes [7], Mammari and Laleau [8,9], Laleau and Polack [10,11] and Davies *et al* [12] have all used formal methods other than Z, typically with a particular focus on deriving database applications from their specifications. Our interest is in the design methodology that surrounds the creation of a database specification, and we focus on Z since it has had a long history of success as a specification language, and there are several examples of its use for database design [13]. Furthermore, it is arguably the most widely known of all formal methods, and is relatively accessible for a formal notation.

Although various methods for modelling relational databases in Z have been suggested, to the best of the authors' knowledge, there has been no independent attempt to compare the various formalisations, or to document the development of these ideas. This paper considers the three key approaches from the literature, and suggests how a fully formal model of relational databases, that also reflects the close ties between Z and the relational model might be achieved. In general, the success of previous attempts to formalise databases in Z have been limited in one of two respects: they are either too far from being used as a tool for database design, or they are not fully formal. In this paper we propose a novel solution to this problem, based on the Formal Template Language (the FTL) introduced by Amálio *et al* [14]. The FTL was developed as a means of describing object oriented design patterns in Z, and we note that the formalism of object oriented approaches is similar to the schemas used to describe the relational model. The application of the FTL in our context is beneficial not only because it allows us to specify a formal database framework, but also because it raises issues that motivate extensions to the FTL.

We have two key motivations: first, we endeavour to show that, to a certain extent, Z can be used as a practical development tool for database design; second, we wish to motivate further research into ways in which Z can be templated.

2 Analysis of Approaches in Z

Relational Model Schemas

Early work from Sufrin *et al* [15] (later expanded by Hayes [16]) contained an example of a schematic description of the relational model. We start, then, with a description of the relational model that follows their approach before considering the currently accepted approaches to relational database representations in Z.

When using this approach, it is assumed that there are given sets *Attributes* and *Entities*, which contain all the attributes and entities that might exist. From these given sets, tuples are defined as such:

$$\mathit{Tuple}[\mathit{Attributes}, \mathit{Entities}] == \mathit{Attributes} \multimap \mathit{Entities}$$

This representation of a tuple is simply a function from *Attributes* without specific domains to *Entities*; we leave the specification of what might be considered a valid tuple until later.

Relations are nothing more than sets of tuples with common domains:

$$\mathit{Relation}[\mathit{Attributes}, \mathit{Entities}] == \{ \mathit{relation} : \mathbb{P} \mathit{Tuple}[\mathit{Attributes}, \mathit{Entities}] \mid (\forall \mathit{tuple}_1, \mathit{tuple}_2 : \mathit{relation} \bullet \mathit{dom} \mathit{tuple}_1 = \mathit{dom} \mathit{tuple}_2) \}$$

Using these definitions, operations such as relational union, intersection and difference are already defined, since we can use the usual set operators on our sets of tuples. Operations such as *join* can be defined explicitly. To do so, we first define an equivalence relation (often termed the *compatibility* between finite mappings) that determines if two tuples agree on their common fields:

$$\begin{array}{|l} \hline \hline \mathit{[Attributes, Entities]} \\ \hline \mathit{--} \cong \mathit{--} : \mathit{Tuple}[\mathit{Attributes}, \mathit{Entities}] \leftrightarrow \mathit{Tuple}[\mathit{Attributes}, \mathit{Entities}] \\ \hline \forall \mathit{tuple}_1, \mathit{tuple}_2 : \mathit{Tuple}[\mathit{Attributes}, \mathit{Entities}] \bullet \\ \mathit{tuple}_1 \cong \mathit{tuple}_2 \Leftrightarrow (\mathit{dom} \mathit{tuple}_2) \triangleleft \mathit{tuple}_1 = (\mathit{dom} \mathit{tuple}_1) \triangleleft \mathit{tuple}_2 \\ \hline \hline \end{array}$$

Using this relation, the definition of *join* becomes relatively straightforward:

$$\begin{array}{|l} \hline \hline \mathit{[Attributes, Entities]} \\ \hline \mathit{--} \bowtie \mathit{--} : \mathit{Relation}[\mathit{Attributes}, \mathit{Entities}] \times \mathit{Relation}[\mathit{Attributes}, \mathit{Entities}] \\ \rightarrow \mathit{Relation}[\mathit{Attributes}, \mathit{Entities}] \\ \hline \forall \mathit{relation}_1, \mathit{relation}_2 : \mathit{Relation}[\mathit{Attributes}, \mathit{Entities}] \bullet \\ \mathit{relation}_1 \bowtie \mathit{relation}_2 = \{ \mathit{tuple}_1 : \mathit{relation}_1; \mathit{tuple}_2 : \mathit{relation}_2 \mid \\ \mathit{tuple}_1 \cong \mathit{tuple}_2 \bullet \mathit{tuple}_1 \cup \mathit{tuple}_2 \} \\ \hline \hline \end{array}$$

At this point we might note that there is no type checking in our definition, and that mal-formed relations could well be defined. To resolve this, we need to define a *TypedRelation*:

$$\begin{array}{|l} \hline \mathit{TypedRelation}[\mathit{Attributes}, \mathit{Entities}] \\ \hline \mathit{relation} : \mathit{Relation}[\mathit{Attributes}, \mathit{Entities}]; \mathit{types} : \mathit{Attributes} \leftrightarrow \mathit{Entities} \\ \hline \forall \mathit{tuple} : \mathit{relation} \bullet \mathit{dom} \mathit{tuple} = \mathit{dom} \mathit{types} \wedge \mathit{tuple} \subseteq \mathit{types} \\ \hline \hline \end{array}$$

In this example, the implementation details of not only the relational model, but also the implied databases that will use that model, are made explicit and unambiguous. This is an important feature of this approach: it is fully formal, and can therefore be checked by tools for consistency and correctness.

Our characterisation of databases in this way could well continue, resulting in both the specification of a full database management system, and whatever database instance we might require. The definitions we have introduced so far have been formal, and completely unambiguous. Using the schemas above, one could well start designing databases and impose constraints on the relations and tuples using the Z notation. However, we must admit that the focus of these schemas is more concerned with the details of the database management system itself, and could be regarded as rather implementation-focused. In terms of its use as a means of assisting in database design, it is far from that domain; furthermore, the details of implementation are too closely tied with the design of the database itself. We are concerned with using Z as a development platform for particular database instances, and would favour a style that is closer to the problem domain. We might also note that in terms of type checking, this definition is not making full use of Z 's in-built system, and has had to resort to an explicit check based on sets. Although the intention is clear, the details may obscure the bigger picture, since we wish to work at a level where we can assume that the database management system exists—and where we are interested in designing a particular database with appropriate constraints.

Mapping Schemas and Relations

Edmond [417] focuses extensively on the mapping between Z as a specification language and SQL as an implementation language. This gives us a further motivation for using Z as a means of describing the constraints that are found in a database: problems that can be expressed in Z can be easily ported to database applications that support SQL. We take the well-known *BirthdayBook* example [2], where our task is to define a means of storing a set of birthdays. The definition is quite natural, making use of the function *birthday* for the mapping between names and dates, and the variable *known*.

$$\begin{aligned} \textit{BirthdayBook} \hat{=} & [\textit{known} : \mathbb{P} \textit{NAME}; \textit{birthday} : \textit{NAME} \rightarrow \textit{DATE} \mid \\ & \textit{known} = \text{dom } \textit{birthday}] \end{aligned}$$

We also need to provide a schema that represents the relation that will store the information. We use a relation called *Person* to store the details of the people whose birthdays we are interested in. Here we use schemas as a means of describing the relation type, since this provides a simple mechanism for mapping attributes to their values. Using schemas as a means of representing tuple types allows us to use Z to directly represent attribute domains, such as *NAME* and *DATE*, as given sets. The database holds a set *people* that represents the tuples in our relation. In our small example, only one relation is required.

$$\textit{Person} \hat{=} [\textit{name} : \textit{NAME}; \textit{birth} : \textit{DATE}]$$

$$\begin{aligned} \textit{BirthdayDatabase} \hat{=} & [\textit{people} : \mathbb{P} \textit{Person} \mid \\ & \# \{ \textit{person} : \textit{people} \bullet \textit{person.name} \} = \# \textit{people}] \end{aligned}$$

The database schema is also used to enforce constraints on the relations contained within, such as the primary key constraint described here that establishes that *name* is a primary key.

To link the *BirthdayBook* description with the *BirthdayDatabase*, we define a mapping between the two. The schema *Mapping* can be used as a coupling invariant that links the abstract *BirthdayBook* schema to the more concrete *BirthdayDatabase* schema. Operations on *BirthdayBook* can then be translated into operations on *BirthdayDatabase* in the usual way.

$$\begin{aligned} \textit{Mapping} &\hat{=} [\textit{BirthdayBook}; \textit{BirthdayDatabase} \mid \\ &\textit{birthday} = \{ \textit{person} : \textit{people} \bullet \textit{person.name} \mapsto \textit{person.birth} \}] \end{aligned}$$

Edmond's contribution considers at great length the issue of Z as a means of information modelling, and the transition between Z and SQL. The focus of the work is not on providing a framework for designing databases in Z; rather, the developer is to acquire general skills for developing databases, and must reimplement key features such as primary key constraints in every instance. The focus of Edmond's work is on designing representations in Z and porting them to SQL, rather than on using Z as a means of specifying SQL constraints more directly. One might argue that this is not a difficult task; a counter-argument, though, is that any provision of tools should include basic guidelines for problems that are often encountered, thus saving time, and enabling thoughts to be directed towards the real task at hand. Although this work indicates a method for translating between Z and SQL, it does not provide a design framework, or specific instruction with regards to database design methodology in terms of artifacts that are to be included in schema descriptions.

Database Relation Schemas

Possibly the most comprehensive attempt at using Z as a means of developing databases is found in the work of de Barros (see, for example, [18] and [19]), which is focused on the provision of a framework, or pattern of development, to enable software engineers to develop database specifications using Z. Guidelines are suggested to solve often visited problems, and a clear step by step methodology for specifying complete database systems with constraints is presented. This work uses a slightly different approach to modelling the tuples of the database, and is focused on bringing the Z description of databases even closer to its relational roots, by allowing constraints to be expressed at various different levels.

Particular emphasis is made on the difference between the relation *intention*, and *extension*, and this is a feature that sets this work apart from that of Edmond, since it allows constraints to be expressed on a more local, per relation, level, as well as on the global database level. In this context, the relation intention indicates the attributes and their domains that will constitute the relation's type. Since the relation intention is considered to be the type of a relation, it is written in capitals, as such:

$$\textit{RELATION} \hat{=} [\textit{Attribute}_1 : \textit{DOMAIN}_1; \textit{Attribute}_2 : \textit{DOMAIN}_2; \dots]$$

Each relation intention describes the range of possible tuples that a tuple might validly take, whether it exists in the database or not. The relation extension is its counterpart, and records the tuples that are present in the database:

$$Relation \hat{=} [relation : \mathbb{P} RELATION \mid KEY_OF relation Attribute_i; \dots]$$

The relation extension also holds various constraints local to the relation; we have included the predicate *KEY_OF* that may, or may not, appear in a definition.

$KEY_OF : \mathbb{P} A \rightarrow (A \rightarrow B) \rightarrow BOOL$ $\forall relation : \mathbb{P} A; attribute : (A \rightarrow B) \bullet KEY_OF relation attribute \Leftrightarrow$ $(\forall t_1, t_2 : relation \bullet attribute t_1 = attribute t_2 \Leftrightarrow t_1 = t_2)$

Other predicates such as *REQUIRED* are defined similarly. (Whereas the previous works had focused on completely determined database values, de Barros introduced null values into his model within each attribute domain.)

Using Z schemas as relational schemas affords us the convenience of referring to attributes by name, but this is at the cost of a reduced set of standard operations on the schemas themselves: the relations based on set functions of Sufrin *et al* gave us most of the relational operators for free, and we needed only to define join. Using Z schema operators in this context has already been explored in [20], which concludes that higher order extensions to Z would be required to fully support a “schemas as schemas” representation where general operators can be defined to support relational schema manipulation using Z schemas as relational ones. Nevertheless, de Barros demonstrates that any specific required operator between two relations can be crafted to work appropriately.

The relations that make up the database are combined in a single schema:

$$Database \hat{=} [Relation_1; Relation_2; \dots \mid FOR_KEY Relation_i ForeignKey Relation_j PrimaryKey; \dots]$$

The *Database* schema is also used as a means of storing global constraints, such as the attributes that are to be considered as foreign keys. In these definitions, de Barros makes it clear that special functions such as *KEY_OF* and *FOR_KEY* are syntactic sugar for generic definitions that allow the reuse of commonly required predicates. Intuitively, these definitions are straightforward, but a close inspection of their definitions reveals that they are not as formal as we would desire—certainly they are insufficient for a fully formal metamodel.

de Barros’ contribution is considerable; he offers a well documented guide to specifying database systems in Z and his method ought to help database designers to find ambiguities and deficiencies in the requirements specification. In particular, the work makes the formal specification of such systems accessible. Further, the method that he suggests covers many aspects of design, and suggests a good level of abstraction to deal with problems that require a database.

There are several things to note. First, there is the implicit assumption that the use of an informal “...” in a schema description is unambiguous and meaningful. This is not an unreasonable assumption when the method proposed is

followed by a developer, but means that the work cannot be automated without further formalisation. Second, much of the work that is detailed is repetitive, such as the required definitions of certain operators (insertions, for example), for each relation that is constructed. Some repetition is avoided through generic macros like *KEY_OF*, but these only represent a fraction of the requirements that are needed in a specification. This repetition is tedious, yet necessary since the schemas produced are crucial in order for the system to make sense. Clearly, there is scope for automation here, and one might look to higher order functions for this purpose, but these are not within the framework that Z offers. Finally, there is a sleight of hand when discussing attributes as candidate keys, and the other suggested macro style predicates. Although the intention of this work is clear, the definitions of certain key functions, such as *KEY_OF*, are not part of the syntax of Z, since the type *BOOL* does not exist as a basic construct. This approach is understandable as it is desirable to capture the repetitive nature of the predicates they express. Leaving aside this criticism, there are further technical difficulties. The type of *KEY_OF* takes two generic parameters, yet at the point of use in the *Relation* schema, we see that it takes an attribute as an argument. This attribute would have type *DOMAIN_i*, yet the function requires the attribute to have generic type $A \rightarrow B$, so the substitution is not valid. It is natural for such errors to creep into a complicated specification, and these are usually caught by a type checker. However, due to the informal nature of some of the schemas described (in part, necessitated by a lack of second-order theory, as noted in [20]), no such type checking is possible. These problems all arise due to informal notation that is used when describing the specification method, and the lesson to be learnt is clear: the automated checking of our specifications is necessary, since we cannot rely on our specifications to be correctly defined.

Despite its intention to provide a formal framework of development with specific instructions, or a prescription for a means of development, there are clear limitations due to its lack of formal precision when considering some of the functions that are intended to be reused in different contexts.

In this section we have discussed previous work that has considered Z as a means of designing databases. For the remainder of this paper, we do not focus on the design of databases *per se*, but, rather, we concentrate on how previous expositions can be further formalised. In each of the above, there is a tension between the description of the formulation of a solution to the representation of databases, and an instantiation of that solution. This tension exists because the role of Z is mixed—it serves as both a meta-language used to describe the form that database instances should take, and as the language used to describe a particular instance. This, inevitably, leads to confusion.

3 Formal Templates for Relational Schemas

In this section, we provide an overview of the pertinent features of the FTL; we also provide an example of a database description using this notation using the guidelines set out by de Barros.

Although developed for a different context (a means of providing a framework to express object oriented patterns in Z), the FTL is a promising approach to generics that enables the formal description of templated schemas, which can be instantiated to create design specifications for databases.

Formal templates aim to elucidate informal template descriptions such as

$$Name \hat{=} [declarations \mid predicates]$$

where there is no syntactic separation between what can be considered part of the specification itself, and what is to be specialised for a specific purpose. The tokens *Name*, *declarations*, and *predicates* are intended to be placeholders for other variable names and expressions, yet they appear to be part of the specification itself. Such a definition might be described more formally by the following template:

$$\langle Name \rangle \hat{=} [\langle declaration \rangle] \mid [\langle predicate \rangle]$$

Templates such as this are to be regarded as nothing more than sequences of characters, where the special symbols \langle and \rangle are used to denote *placeholders* for other sequences of characters, and \llbracket and \rrbracket indicate that elements inside these brackets are to be replaced by multiple occurrences. Elements found in placeholders are replaced by strings determined by a structure that represents an environment within which the template should be evaluated.

The structure used to hold the desired mappings is a tree of environments, where each environment indicates the mappings for placeholders not in lists, and the branches of a tree hold environments intended to hold mappings for the placeholders in lists. (For brevity we have simplified the exposition and removed certain features, such as template choice, that we do not make use of in this work. For full details see [21].) As such, we find that these representations are difficult to work with, but for the purposes of this paper, we maintain the original structure proposed by Amálio.

This kind of replacement is between arbitrary strings, and need not apply only to Z schemas: the intention is that it could be used for any language. This means that even if a template is designed with Z replacement in mind, the result of applying a mapping to this template need not make sense in Z. In order to apply semantics to the result of these substitutions, we assume the existence of a predicate that indicates whether or not an expression is a well-formed and type-correct Z specification. This is used to restrict templates and mappings to those that, when used in conjunction, produce meaningful specifications.

We have already discussed the success Z has had in the context of teaching database design and we now focus on improving its formality to a point where database schemas can automatically be generated from templates. The development method suggested by de Barros is the most thorough, and we proceed by applying the FTL to this description in order to further formalise its exposition, and to overcome the shortcomings described in the previous section. The application of the FTL to this description of a database system is relatively straightforward, since we need only replace the various elements of the database description with templated counterparts.

The relation intention is nothing more than a schema that holds a list of the attributes that its corresponding relation should have.

$$\boxed{\begin{array}{l} \langle \textit{RELATION} \rangle \text{---} \\ \llbracket \langle \textit{attribute} \rangle : \langle \textit{ATTRIBUTE} \rangle \rrbracket \end{array}}$$

This template can be instantiated with the following placeholder mapping to generate the schema for our birthday database example:

$$\begin{array}{l} \textit{tree}(\{\textit{RELATION} \mapsto \textit{PERSON}\}, \\ \quad \langle \textit{tree}(\{\textit{attribute} \mapsto \textit{name}, \textit{ATTRIBUTE} \mapsto \textit{NAME}\}, \langle \rangle), \\ \quad \textit{tree}(\{\textit{attribute} \mapsto \textit{birthday}, \textit{ATTRIBUTE} \mapsto \textit{DATE}\}, \langle \rangle) \rangle) \end{array}$$

$$\textit{PERSON} \hat{=} [\textit{name} : \textit{NAME}; \textit{birthday} : \textit{DATE}]$$

The relation extension can be adapted similarly, and here we have embedded the primary key constraint directly into the schema:

$$\boxed{\begin{array}{l} \langle \textit{Relation} \rangle \text{---} \\ \langle \textit{relations} \rangle : \mathbb{P} \langle \textit{RELATION} \rangle \\ \hline (\forall t_1, t_2 : \langle \textit{relations} \rangle \bullet \llbracket t_1.\langle \textit{key} \rangle = t_2.\langle \textit{key} \rangle \rrbracket \wedge t_1 = t_2) \end{array}}$$

The primary key constraint that we use here is similar to that of de Barros, but with a modification in the way composite keys are handled: we have added the parameter \wedge to the template list brackets as a *list separator*. The list separator is an optional argument to template list brackets, and specifies a sequence of characters that is to separate fragments generated by a particular list. (We shall see an example of its use in an instantiation of a composite key later.) For now, we instantiate this template with the following mapping, where a single attribute serves as a key, generating the relation extension *Person*:

$$\begin{array}{l} \textit{tree}(\{\textit{Relation} \mapsto \textit{Person}, \textit{relations} \mapsto \textit{people}, \textit{RELATION} \mapsto \textit{PERSON}\}, \\ \quad \langle \textit{tree}(\{\textit{key} \mapsto \textit{name}\}, \langle \rangle) \rangle) \end{array}$$

$$\begin{array}{l} \textit{Person} \hat{=} \\ \quad [\textit{people} : \mathbb{P} \textit{PERSON} \mid \forall t_1, t_2 : \textit{people} \bullet t_1.\textit{name} = t_2.\textit{name} \Leftrightarrow t_1 = t_2] \end{array}$$

The inclusion of a foreign key constraint is also easily demonstrated. To do so, we instantiate another relation through two more schemas, and place the foreign key constraint in a separate database schema that represents the overall database state. First, we define a relation *Author* that will use the *name* attribute of the *Person* relation as a foreign key.

As with the *Person* relation, we first instantiate the relation intention:

$$\begin{array}{l} \textit{tree}(\{\textit{RELATION} \mapsto \textit{AUTHOR}\}, \\ \quad \langle \textit{tree}(\{\textit{attribute} \mapsto \textit{title}, \textit{ATTRIBUTE} \mapsto \textit{TITLE}\}, \langle \rangle), \\ \quad \textit{tree}(\{\textit{attribute} \mapsto \textit{author}, \textit{ATTRIBUTE} \mapsto \textit{NAME}\}, \langle \rangle) \rangle) \end{array}$$

This mapping produces the following schema:

$$AUTHOR \hat{=} [title : TITLE; author : NAME]$$

Now we provide a mapping to be used with the relation extension template, which results in the *Author* schema:

$$tree(\{Relation \mapsto Author, relations \mapsto authors, RELATION \mapsto AUTHOR\}, \\ \langle tree(\{key \mapsto title\}, \langle \rangle), tree(\{key \mapsto author\}, \langle \rangle) \rangle)$$

$$Author \hat{=} [authors : \mathbb{P} AUTHOR \mid \forall t_1, t_2 : authors \bullet \\ t_1.title = t_2.title \wedge t_1.author = t_2.author \Leftrightarrow t_1 = t_2]$$

Here we have made use of the list separator, mentioned previously, to help with the definition of a composite key parameter, made up of both *title* and *author*.

Finally, we define the database template that contains all the relations, as well as any foreign key constraints that might be present.

$$\begin{array}{|l} \hline Database \\ \hline \llbracket \langle Relation \rangle \rrbracket \\ \llbracket \forall t_n : \langle relation_{native} \rangle \bullet \\ (\exists_1 t_f : \langle relation_{foreign} \rangle \bullet \llbracket t_n.\langle key_{native} \rangle = t_f.\langle key_{foreign} \rangle \rrbracket \wedge) \rrbracket \\ \hline \end{array}$$

This definition makes use of two separate template lists. The FTL instantiates lists using replacements in the order that they are found in the environment until no appropriate substitutions are available for the list. We use the following environment to instantiate the database schema and produce *Database*:

$$tree(\{\}, \\ \langle tree(\{Relation \mapsto Person, \\ relation_{native} \mapsto authors, relation_{foreign} \mapsto people\}, \\ \langle tree(\{key_{native} = author, key_{foreign} = name\}, \langle \rangle)\rangle), \\ tree(\{Relation \mapsto Author\}, \langle \rangle) \rangle)$$

$$Database \hat{=} [Person; Author \mid \forall t_n : authors \bullet \\ (\exists_1 t_f : people \bullet t_n.author = t_f.name)]$$

We have used the FTL to help provide a clear distinction between the components of the specification that are part of the database framework, and those that are associated with a particular instantiation. To this end, our application of the FTL has been successful, and in formalising this distinction we have gained the ability to mechanically produce a range of schemas from templates.

To further our example, we might consider an auxiliary template that supports the insertion of data into its associated schema.

$$\begin{array}{|l} \hline \langle Relation \rangle Insert \\ \hline \Delta Database; \exists Database \setminus (\langle relations \rangle) \\ \langle relations \rangle? : \mathbb{P} \langle RELATION \rangle \\ \langle relations \rangle' = \langle relations \rangle \cup \langle relations \rangle? \\ \hline \end{array}$$

This can be instantiated with the environments used for *Person* and *Author*. Once instantiated with the *Person* environment, we have the following schema:

$\begin{array}{l} \textit{PersonInsert} \\ \Delta\textit{Database}; \exists\textit{Database} \setminus (\textit{people}); \textit{people}? : \mathbb{P} \textit{PERSON} \\ \textit{people}' = \textit{people} \cup \textit{people}? \end{array}$
--

Encapsulating the general form of the auxiliary schemas with a template in this way saves us from having to manually define a whole family of schemas that are related, and relatively routine to derive. Typically, we would expect there to be many different relations, and a number of auxiliary schemas for each of those relations, such as schemas to initialise the relation and others to delete records. Without the FTL we would have to design each of these schemas manually, and be careful that each specification is correct. By using templates, this mundane work is avoided since schemas can be mechanically produced from the appropriate environments.

In addition, Amálio has shown that the use of FTL also brings the possibility of performing meta-proofs on templates, allowing certain proof obligations on sets of related schemas to be discharged once for all. This is made possible by proving a meta-theorem concerning the template that is general enough to cover any instantiations of that template. This would certainly be of use for discharging obligations of many of the supporting schemas that surround database specifications, such as state initialisations.

4 Extending the Formal Template Language

The previous section has shown how the application of templates to database descriptions in Z is beneficial, and we have demonstrated how this formalises the specification of a framework of schemas. The key observation of this paper is that although the use of the FTL is useful in this context and holds promising results, there are a number of shortcomings that need to be addressed in order to make its application truly practical for our purposes. Some of these issues have been addressed by a prototype tool developed by the authors that allows schemas to be named, and to be automatically instantiated from templates and environments. We outline our solution in this section.

Our examples avoid the explicit naming of certain useful predicates, and have resorted to writing the predicates for primary and foreign keys in place, rather than referring to them as template fragments. This can certainly be considered a regression, since de Barros's *KEY_OF* function enabled a succinct, parameterised predicate to be used where appropriate. Indeed, we have made these omissions since the original specification of the FTL does not have adequate machinery to support, amongst other things, naming and referencing templates. (The FTL allows certain parts of the template to be optional, but this is not the same as using named template substitutions when they are desired.)

It is tempting to use the name of the schema as the formal template name. For example, we might have called the templates we defined earlier $\langle \textit{RELATION} \rangle$,

$\langle Relation \rangle$, and *Database*, depending on the schema name that was involved. However, this approach is not satisfactory since these string fragments are incidental parts of the template definition.

Templates are nothing more than sequences of characters, so we can validly assign a template fragment as a sequence of characters bound to a Z variable:

$$KEY_OF == (\forall t_1, t_2 : \langle relation \rangle \bullet t_1.\langle key \rangle = t_2.\langle key \rangle \Leftrightarrow t_1 = t_2)$$

By modifying the original template, we can make use of this reference:

$\langle Relation \rangle$	$\langle relation \rangle : \mathbb{P}\langle RELATION \rangle$
$\langle KEY_OF \rangle$	

Here, the placeholder $\langle KEY_OF \rangle$ does not hold an ordinary variable, but rather, it can be read as a placeholder for the appropriate template. However, when we allow this kind of substitution, there are still problems that need to be addressed. Importing one template into another like this implicitly adds more placeholders that need to be initialised by the environment mapping. The difficulty lies in the possibility of recursive template inclusion—since some of these mappings could point to further templates—and how this might be appropriately handled. If left unchecked then some template definitions are impossible to evaluate, whilst checking for these recursions will require the notion of template dependency tracking, and this is an issue that we have not yet resolved.

An inconvenience of the FTL is the rather unwieldy nature of its instantiation environments, where nested tree structures are required to instantiate nested lists of placeholders. These environments are required to produce a finished database specification, and so they become an integral part of the design process where any specific constraints are expressed as the target of a placeholder. The environments of the FTL somewhat obscure this important role, and it is desirable to express environments in a form that more closely captures the salient features of schemas. Our approach to facilitating the definition of environments is to use templates as a means of generating instantiation environments from valid strings. To this end, our tool can make use of templates to generate specific parsers that create environments from parsed strings; each basic template construct produces a particular parser, and parser combinator functions are used to produce an overall parser that represents an entire template. This allows the user to define templates that accept “minimal schemas” as input, and derive all the related schemas from the environments produced, in combination with the relevant auxiliary templates. Of course, not all templates can be used “in reverse” through this process, but this solution has proved to be effective for practical use cases.

5 Conclusions

There are two key observations that we have made in this paper. First, we have noted that although using Z as a framework that enforces constraints in the

design of relational databases is considered a viable method, it is not formal in every aspect. When using Z to make design recommendations, rather than to design specifications directly, we find that there is no separation between these two different uses for Z, and that this leads to difficulties. It is to this end that using formal templates is useful, and we anticipate that to some extent using the FTL can alleviate this problem. Our second observation is that using the FTL in our context has raised some interesting avenues for future research. Primarily, we saw that the original FTL lacks any means of referring to templates by name, and that template substitution is impossible. Such a feature is important in making templates more readable and flexible, especially for our purposes, where design recommendations are the purpose of using Z.

There are other features that are desirable for a notation that can describe formal frameworks of development in Z. For example, we see that in many of our definitions, the use of different cases has semantics attached: typically, upper case, lower case, and camel case names respectively represent types, variables and schemas. This convention can be maintained in the names of the template substitution parameters, but at present there is no way to enforce the preservation of this naming convention on the schema level. At present, our tool automatically augments environments that are created by using its parsers generated from templates with mappings from all keywords in their various cases, and maps these to values with their cases modified in similar ways. Despite these shortcomings, using the FTL to formalise database metamodels allows the generation of database schema representations in Z from instantiation environments. Its use clarifies ambiguities that appear in previous applications of Z in this area, and allows us to further focus on the design issues that pertain to a specific database.

Finally, we note that there are similarities between our descriptions of relations and objects in Z for which the FTL was designed. The resemblance of the work of de Barros, compared to that of Hall [22], in representing objects, is significant. In Hall's work we find almost identical definitions for class intentions and extensions as we have seen for relations: the main difference is that objects have a notion of *self*, and this can easily be understood as a special attribute that is a primary key candidate. This resemblance in terms of Z specifications is particularly revealing, since it points to similarities between the relational model and object class representations. Indeed, this connection has already been addressed in the well known Object-Relation Mapping, but the fact that the similarity is so easily demonstrated in Z is testimony to its abstract descriptive power.

References

1. Codd, E.F.: A relational model of data for large shared data banks. *Communications of the ACM* 13(6), 377–387 (1970)
2. Spivey, J.M.: *The Z notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs (1992)
3. van Diepen, M.J., van Hee, K.M.: A Formal Semantics for Z and the Link between Z and the Relational Algebra. In: Langmaack, H., Hoare, C.A.R., Bjorner, D. (eds.) *VDM 1990*. LNCS, vol. 428, pp. 526–551. Springer, Heidelberg (1990)

4. Edmond, D.: *Information Modeling: Specification and Implementation*. Prentice-Hall, Englewood Cliffs (1992)
5. Davies, J.W.M., Simpson, A.C., Martin, A.P.: Teaching Formal Methods in Context. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 185–202. Springer, Heidelberg (2004)
6. Schewe, K.D., Schmidt, J.W., Wetzel, I.: Specification and refinement in an integrated database application environment. In: Prehn, S., Toetenel, H. (eds.) VDM 1991. LNCS, vol. 551, pp. 496–510. Springer, Heidelberg (1991)
7. Hayes, I.: VDM and Z: A comparative case study. *Formal Aspects of Computing* 4(1), 76–99 (1992)
8. Mammarr, A., Laleau, R.: Design of an automatic prover dedicated to the refinement of database applications. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 834–854. Springer, Heidelberg (2003)
9. Mammarr, A., Laleau, R.: A formal approach based on UML and B for the specification and development of database applications. *Automated Software Engineering* 13(4), 497–528 (2006)
10. Laleau, R., Polack, F.: Specification of integrity-preserving operations in information systems by using a formal UML-based language. *Information & Software Technology* 43(12), 693–704 (2001)
11. Laleau, R., Polack, F.: Using formal metamodels to check consistency of functional views in information systems specification. *Information & Software Technology* 50(7-8), 797–814 (2008)
12. Davies, J.W.M., Welch, J., Cavarra, A.L., Crichton, E.: On the Generation of Object Databases using Booster. In: *Proceedings of the 11th IEEE Conference on the Engineering of Complex Computer Systems* (2006)
13. Gray, D.: The Formal Specification of a Small Bookshop Information System. *IEEE Transactions on Software Engineering* 14(2), 263–272 (1988)
14. Amálio, N., Stepney, S., Polack, F.: Formal Proof from UML Models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 418–433. Springer, Heidelberg (2004)
15. Sufirin, B.A., Morgan, C.C., Sørensen, I.H., Hayes, I.J.: Notes for a Z handbook. Programming Research Group, Oxford University Computing Laboratory (1984)
16. Hayes, I.J., Jones, C.B., Nicholls, J.E.: Understanding the differences between VDM and Z. *ACM SIGSOFT Software Engineering Notes* 19(3), 75–81 (1994)
17. Edmond, D.: Refining Database Systems. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1995. LNCS, vol. 967, pp. 25–44. Springer, Heidelberg (1995)
18. de Barros, R.S.M.: Deriving Relational Database Programs from Formal Specifications. In: Naftalin, M., Bertrán, M., Denvir, T. (eds.) FME 1994. LNCS, vol. 873, p. 703. Springer, Heidelberg (1994)
19. de Barros, R.S.M.: On the Formal Specification and Derivation of Relational Database Applications. PhD thesis, Dept. of Computing Science, University of Glasgow (1994)
20. Martin, A.P., Simpson, A.C.: Generalising the Z schema calculus: database schemas and beyond. In: *Proceedings of APSEC 2003*, pp. 28–37 (2003)
21. Amálio, N., Stepney, S., Polack, F.: A Formal Template Language Enabling Metaproof. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 252–267. Springer, Heidelberg (2006)
22. Hall, A.: Specifying and Interpreting Class Hierarchies in Z. In: *Proceedings of the 1994 Z User Workshop*, pp. 120–138 (1994)

Translating Z to Alloy

Petra Malik, Lindsay Groves, and Clare Lenihan

Victoria University, Wellington, New Zealand

Petra.Malik@ecs.vuw.ac.nz,

Lindsay.Groves@ecs.vuw.ac.nz,

Clare.Lenihan@ecs.vuw.ac.nz

Abstract. Few tools are available to help with the difficult task of validating that a Z specification captures its intended meaning. One tool that has been proven to be useful for validating specifications is the Alloy Analyzer, an interactive tool for checking and visualising Alloy models. However, Z specifications need to be translated to Alloy notation to make use of the Alloy Analyzer. These translations have been performed manually so far, which is a cumbersome and error-prone activity. The aim of this paper is to explore to what extent this process can be automated.

The paper identifies a subset of Z that can be straightforwardly translated to Alloy, and the translation for this subset is formalised. More complex constructs, like schemas, are harder to translate. The paper gives a brief overview of the problems, and discusses alternative translation approaches.

1 Introduction

The *Z notation* [18,8] has been widely used for the design and specification of computing systems, however, one of the barriers to more widespread use of Z is the lack of tool support. Existing Z tools provide only limited help with the difficult task of validating that a Z specification captures its intended meaning. A parser and typechecker [12,19] can be used to ensure that specifications are correct with regard to syntax and type constraints, but semantic errors cannot be detected using these tools. Theorem provers [11,13] can be used to investigate properties of interest, but using a theorem prover is usually tedious and requires expert knowledge. Animators [16,22,7] allow the evaluation and execution of certain predicates and expressions in the context of a specification, but the few Z animators available have not yet reached the level of maturity provided by similar tools for other notations, like B and Alloy.

This paper considers translating Z to Alloy [9] to give Z users access to the Alloy Analyzer. The Alloy Analyzer can be used to generate instances of Alloy specifications as well as to check user-specified properties. This provides immediate visual feedback to the specification designer and is an invaluable help in checking and validating Alloy specifications. Others have already shown that the Alloy Analyzer can be successfully used to check certain aspects of Z specifications. Bolton [3], for example, gives an encoding for some Z data types in

<pre> section <i>birthdaybook</i> parents <i>standard_toolkit</i> [<i>NAME</i>, <i>DATE</i>] ----- <i>BBook</i> <i>known</i> : \mathbb{P} <i>NAME</i> <i>birthday</i> : <i>NAME</i> \leftrightarrow <i>DATE</i> ----- <i>known</i> = dom <i>birthday</i> ----- <i>Init</i> <i>BBook</i> ----- <i>known</i> = \emptyset ----- <i>Add</i> Δ<i>BBook</i> <i>name?</i> : <i>NAME</i> <i>date?</i> : <i>DATE</i> ----- <i>name?</i> \notin <i>known</i> <i>birthday'</i> = <i>birthday</i> \cup {<i>name?</i> \mapsto <i>date?</i>} </pre>	<pre> module <i>birthdaybook</i> open <i>util/relation</i> sig <i>NAME</i>, <i>DATE</i> {} sig <i>BBook</i> { <i>known</i>: set <i>NAME</i>, <i>birthday</i>: <i>NAME</i> \rightarrow lone <i>DATE</i> }{ <i>known</i> = dom[<i>birthday</i>] } pred <i>Init</i> [<i>s</i>: <i>BBook</i>] { no <i>s.known</i> } pred <i>Add</i> [<i>s</i>,<i>s'</i>: <i>BBook</i>, <i>name_in</i>: <i>NAME</i>, <i>date_in</i>: <i>DATE</i>] { <i>name_in</i> !in <i>s.known</i> <i>s'.birthday</i> = <i>s.birthday</i> + (<i>name_in</i> \rightarrow <i>date_in</i>) } </pre>
---	--

Fig. 1. Z and Alloy specifications for a simplified version of Spivey’s birthday book [18]

Alloy and then uses the Alloy Analyzer to identify a retrieve relation to verify refinement between two Z data types. Estler and Wehrheim [6] use the Alloy Analyzer to check refactorings of Z specifications. Ramananandro [15] uses the Alloy Analyzer to check the Z specification and refinement proofs for Mondex, an electronic purse. Kang and Jackson [10] model and analyse a flash file system, which was originally specified in Z.

All these translations were done by hand, and consider selected examples only. Figure 1 shows a Z specification, and a typical hand-translation into Alloy. The similarities between Z and Alloy are seductive, and the translation seems straightforward. But are these two specifications semantically equivalent? Does a property that holds for the Z specification also hold true for the corresponding Alloy, and vice versa? Can the translation be automated, and are there alternative translation approaches? While some of these questions have been answered by the aforementioned authors in the context of their example, no systematic investigation has been carried out so far. The aim of this paper is to generalise from these examples and consider, in a systematic way, what can be translated automatically and to what.

$$\begin{aligned}
&\{NAME \mapsto \emptyset, \\
&\quad DATE \mapsto \emptyset, \\
&\quad BBook \mapsto \{\{known \mapsto \emptyset, birthday \mapsto \emptyset\}\}, \\
&\quad\quad Init \mapsto \{\{known \mapsto \emptyset, birthday \mapsto \emptyset\}\}, \\
&\quad\quad Add \mapsto \emptyset\}
\end{aligned}$$

Fig. 2. A model for the Z specification of the birthday book

The structure of this paper is as follows. Section 2 shows the relationship between Z models and Alloy instances, and highlights their subtle but important differences. Section 3 formally defines a subset of Z and its translation to Alloy. Section 4 shows how the language can be extended to more complex structures, and also points out difficulties with the translation, and alternative translation approaches. Finally, Section 5 discusses the contributions of this paper and outlines future work.

2 Z Models and Alloy Instances

The ISO Z Standard [8] defines the meaning of a specification as a set of models. Models associate global names of the specification with semantic values that satisfy the constraints of the specification. One possible model for the Z specification of the birthday book is given in Figure 2. It associates the given types *NAME* and *DATE* with the empty set. *BBook* and *Init* are associated with the set of tuples that give *known* and *birthday* values; the empty set in this case. The state space *BBook* contains just one element, the state where *known* and *birthday* are both empty. The operation *Add* is empty since there is no element in the set *NAME* that could be added.

A more interesting model is given in Figure 3. In this model, there are exactly two names (*N0* and *N1*) and one date (*D*). The state space represented by *BBook* has four elements: the empty birthday book, the birthday book that maps name *N0* to date *D*, the birthday book that maps name *N1* to date *D*, and the birthday book that maps both names to date *D*. Exactly one state satisfies the initialisation condition. The operation *Add* has four elements. If the pre-state is empty, either name *N0* or name *N1* can be added. If both names are already mapped to date *D* in the pre-state, operation *Add* is not defined. If only one of the names is mapped to date *D* in the pre-state, the other name can be added.

To the authors' knowledge, there is no tool support that allows visualisation or even computation of models for Z specifications. We will see next how the Alloy Analyzer can be used to compute and visually explore instances. Instances are bindings from names to values that satisfy the constraints of a given Alloy specification. To make instance finding feasible, a scope limits the number of elements in the considered universe and the analysis performed by the Alloy Analyzer effectively explores every instance within the scope.

$\{NAME \mapsto \{N0, N1\},$
 $DATE \mapsto \{D\},$
 $BBook \mapsto \{\{known \mapsto \emptyset, birthday \mapsto \emptyset\},$
 $\quad \{known \mapsto \{N0\}, birthday \mapsto \{N0 \mapsto D\}\},$
 $\quad \{known \mapsto \{N1\}, birthday \mapsto \{N1 \mapsto D\}\},$
 $\quad \{known \mapsto \{N0, N1\}, birthday \mapsto \{N0 \mapsto D, N1 \mapsto D\}\}\},$
 $Init \mapsto \{\{known \mapsto \emptyset, birthday \mapsto \emptyset\}\},$
 $Add \mapsto \{\{known \mapsto \emptyset, birthday \mapsto \emptyset, name? \mapsto N0, date? \mapsto D,$
 $\quad known' \mapsto \{N0\}, birthday' \mapsto \{N0 \mapsto D\}\},$
 $\{known \mapsto \emptyset, birthday \mapsto \emptyset, name? \mapsto N1, date? \mapsto D,$
 $\quad known' \mapsto \{N1\}, birthday' \mapsto \{N1 \mapsto D\}\},$
 $\{known \mapsto \{N0\}, birthday \mapsto \{N0 \mapsto D\}\}, name? \mapsto N1, date? \mapsto D,$
 $\quad known' \mapsto \{N0, N1\}, birthday' \mapsto \{N0 \mapsto D, N1 \mapsto D\}\},$
 $\{known \mapsto \{N1\}, birthday \mapsto \{N1 \mapsto D\}\}, name? \mapsto N0, date? \mapsto D,$
 $\quad known' \mapsto \{N0, N1\}, birthday' \mapsto \{N0 \mapsto D, N1 \mapsto D\}\}\}$

Fig. 3. Another model for the Z specification of the birthday book

For example, Figure 4 shows visualisations of two of the (many) instances of the Alloy specification of the birthday book given in Figure 1 obtained by executing the following command:

`run {} for 4 but 2 NAME, 1 DATE`

The scope of this command restricts the size of each signature to a maximum of four elements (the default is three) apart from NAME, which is allowed two, and DATE, which is only allowed to contain one element. In Figure 4, elements of NAME are represented by boxes, elements of DATE are surrounded by a hexagon, and elements of BBook are surrounded by ellipses. The relations known and birthday are represented by arrows.

Like the Z model in Figure 3, both instances show two names and one date as well as possible values for known and birthday for the birthday book state

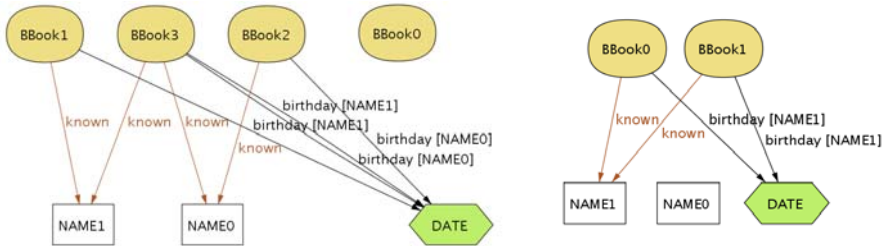


Fig. 4. Two instances of the Alloy version of the birthday book given in Figure 1

BBook. That is, both instances are representations of the same *Z* model, or parts thereof. The left instance, for example, does not include values for **Init** and **Add**. The right instance shows only two elements of the set **BBook**. Furthermore, these two states **BBook0** and **BBook1** represent the same birthday book (with **NAME1** being the only known name, mapped to **DATE**).

That Alloy instances represent parts of a *Z* model is both a blessing and a curse. On the one hand, we are usually not interested in seeing all the possible states of the system (in our example, elements of *BBook*) in one instance. By showing parts of a *Z* model, Alloy instances provide a tractable way of exploring these usually huge sets. On the other hand, allowing instances that represent only part of a *Z* model does not preserve semantics of the original *Z* specification. For example, in *Z* we can prove:

$$\#NAME = 2 \wedge \#DATE = 1 \Rightarrow \#BBook = 4$$

That is, for every *Z* model of the birthday book specification with exactly two names and one date, the state schema *BBook* has exactly four elements, each with a different set associated to *birthday* as seen in Figure 3. This is not true for the Alloy instances as seen in Figure 4, where there are fewer **BBook** elements and two different elements represent the same mapping from name to birthday. The problem is that we chose to represent a *Z* schema (the set of all bindings that satisfy the constraints of the schema) as an Alloy signature, which denotes just some set of atoms that is not constrained to represent all possible values of the corresponding schema.

It is possible to add a constraint to the Alloy specification to enforce that no two distinct elements from **BBook** represent the same birthday book [15]:

```
pred Canonicalisation {
  all disj b1,b2: BBook | b1.birthday != b2.birthday
}
```

It is also possible to add a constraint to ensure that all possible birthday books are represented [9, Section 5.3.1]:

```
pred GeneratorAxiom {
  some b: BBook | no b.birthday
  all b: BBook, n: NAME-b.known, d: DATE |
    some b': BBook |
      b'.birthday = b.birthday + n->d
}
run { GeneratorAxiom and Canonicalisation } for 4
```

In fact, *Canonicalisation* and *GeneratorAxiom* together ensure a one-to-one correspondence between *Z* models and Alloy instances. The left instance in Figure 4 is one of the solutions generated by the above run command while the right instance is not. However, adding a generator axiom is impractical in most cases due to the state explosion problem. Jackson [9, Section 5.3.2] argues that generator axioms are rarely needed in practice and more research is needed to

investigate how this shortcoming affects the usability of the Alloy Analyzer for checking Z specifications.

The next section gives a semantic preserving translation for a small subset of the Z notation into Alloy so that the models of the Z specification correspond directly to the instances of the corresponding Alloy specification.

3 A Semantics Preserving Translation for a Subset of Z

The basic building blocks of Alloy specifications are *atoms* and *relations*. Atoms are primitive entities: indivisible, immutable, and uninterpreted, like the elements of given sets in Z. Each variable in Alloy denotes a relation, a structure that relates atoms. A Z specification that uses only given types, elements and subsets of given types, and tuples and relations over given types can be directly translated to a semantically equivalent Alloy specification. Such a translation can be seen as a shallow embedding of a subset of Z into Alloy. In a shallow embedding, the meaning of the Z specification is retained by the translation into a semantically equivalent representation in Alloy. The translation ensures that the models of the Z specification are directly represented by the instances for the corresponding Alloy specification. No generator or canonicalisation axioms are required.

Figure 5 defines Z paragraphs for which a semantics preserving translation into Alloy is provided below. We assume that the Z specification to be translated has been successfully parsed and typechecked. The language is a subset of that defined by the ISO Z standard's annotated syntax [8, Clause 10]. It includes paragraphs that declare given types, elements and subsets of given types, and tuples and relations over given types. Expressions are annotated with similarly restricted types. All the predicates of the ISO Z standard's annotated syntax are included.

Note that the annotated syntax provided by the ISO Z standard is smaller than the full Z language. For example, it does not include existential quantification. Syntactic transformation rules [8, Clause 12] relate Z phrases of the full Z language to equivalent phrases within the language defined by the annotated syntax. Existential quantification, for example, is transformed to a negation of a universal quantification. This means that, by providing a translation to Alloy for a subset of the annotated syntax, we get the translation of some constructs that are not in the annotated syntax like existential quantification for free. Although this is convenient for the formal treatment of the translation to Alloy, a tool does not have to follow this approach and could translate existential quantification directly to the Alloy quantifier *some*.

Figure 6 gives the definition of a translation function $\llbracket \cdot \rrbracket$ that accepts a phrase of our restricted Z annotated syntax, and returns an Alloy phrase. We follow ISO Z standard notation to represent Z phrases. The main complexity is related to declarations. Declarations are translated in different ways, depending on where they are used. Declarations of global variables are defined by the translation function $\llbracket \cdot \rrbracket^g$, declarations of local variables are defined by the translation

```

Paragraph = ZED , [-tok , NAME , ]-tok , END (* given types *)
           | AX , SchText , END (* axiomatic description *)
           ;
SchText = [-tok , Decl , ]-tok , Predicate , ]-tok
         ;
Decl = [-tok , NAME , : , Expression , ]-tok
      | [-tok , NAME , : , P , Expression , ]-tok
      | Decl , ^ , Decl
      ;
Predicate = Expression , ∈ , Expression (* membership *)
           | Expression , ∈ , {-tok , Expression , }-tok (* equality *)
           | true (* truth *)
           | ¬ , Predicate (* negation *)
           | Predicate , ^ , Predicate (* conjunction *)
           | ∀ , Decl , • , Predicate (* universal quantification *)
           | ∃1 , Decl , • , Predicate (* unique existential quantification *)
           ;
Expression = Expr , ∘ , Type
           ;
Expr = NAME (* reference *)
      | {-tok , SchText , • , Expression , }-tok (* set comprehension *)
      | (-tok , Expression , , -tok
         Expression , { , -tok , Expression } , )-tok (* tuple extension *)
      ;
Type = Type2
      | P , Type2
      ;
Type2 = GIVEN , NAME , { × , GIVEN , NAME }
      ;
    
```

Fig. 5. Z paragraphs that can be straightforwardly translated to Alloy. This grammar has been adapted from the Z standard [8, Clause 10]; mathematical definitions from ISO/IEC 13568:2002 (Z standard) are copyright ISO.

function $\llbracket \cdot \rrbracket^c$. Some of this complexity might be avoided by translating to Kodkod [20] rather than Alloy.

The output of the translation function should be thought of as a tree structure, and parentheses might be needed when printed¹. However, before such an Alloy phrase can be printed, some more processing steps might be needed. Firstly, due to the fact that global Alloy relations are defined within the signature of its first component, the translation function might provide multiple definitions for the same signature. These need to be merged before valid Alloy can be printed. Secondly, names need to be translated to valid Alloy names, which unlike Z names are restricted to a subset of the printable ASCII characters.

¹ Precedences in Alloy are slightly different from Z. In Z, for example, disjunction binds tighter than implication and implication binds tighter than equivalence, while in Alloy implication and equivalence bind tighter than disjunction.

$$\begin{array}{l}
 \llbracket \text{ZED } [i] \text{ END} \rrbracket = \text{sig } i \{ \} \\
 \llbracket \text{AX } t \text{ END} \rrbracket = \llbracket t \rrbracket^{\mathcal{G}} \\
 \llbracket [i : e] \rrbracket^{\mathcal{L}} = i : \llbracket e \rrbracket \\
 \llbracket [i : \mathbb{P} e] \rrbracket^{\mathcal{L}} = i : \text{set } \llbracket e \rrbracket \\
 \llbracket d_1 \wedge d_2 \rrbracket^{\mathcal{L}} = \llbracket d_1 \rrbracket^{\mathcal{L}} , \llbracket d_2 \rrbracket^{\mathcal{L}} \\
 \llbracket [i] \rrbracket = i \\
 \llbracket \{ [d | p] \bullet e \circ \tau \} \rrbracket = \{ j_1 : i_1, \dots, j_n : i_n \mid \text{some } \llbracket d \rrbracket^{\mathcal{L}} \mid \llbracket p \rrbracket \text{ and} \\
 \llbracket (j_1, \dots, j_n) \rrbracket = \llbracket e \rrbracket \} \\
 \llbracket (e_1, \dots, e_n) \rrbracket = \llbracket e_1 \rrbracket \rightarrow \dots \rightarrow \llbracket e_n \rrbracket \\
 \\
 \llbracket [i_0 : (e \circ \mathbb{P} \text{ GIVEN } i_1)] \rrbracket^{\mathcal{G}} = \text{one sig } i_0 \text{ in } i_1 \{ \} \text{ fact } \{ i_0 \text{ in } \llbracket e \rrbracket \} \\
 \llbracket [i_0 : \mathbb{P} (e \circ \mathbb{P} \text{ GIVEN } i_1)] \rrbracket^{\mathcal{G}} = \text{sig } i_0 \text{ in } i_1 \{ \} \text{ fact } \{ i_0 \text{ in } \llbracket e \rrbracket \} \\
 \llbracket [i_0 : (e \circ \mathbb{P} \tau)] \rrbracket^{\mathcal{G}} = \text{sig } i_1 \{ i_0 : \text{set } i_2 \rightarrow \dots \rightarrow i_n \} \\
 \text{fact } \{ \text{one } i_0 \text{ and } i_0 \text{ in } \llbracket e \rrbracket \} \\
 \llbracket [i_0 : \mathbb{P} (e \circ \mathbb{P} \tau)] \rrbracket^{\mathcal{G}} = \text{sig } i_1 \{ i_0 : \text{set } i_2 \rightarrow \dots \rightarrow i_n \} \\
 \text{fact } \{ i_0 \text{ in } \llbracket e \rrbracket \} \\
 \llbracket d_1 \wedge d_2 \rrbracket^{\mathcal{G}} = \llbracket d_1 \rrbracket^{\mathcal{G}} \llbracket d_2 \rrbracket^{\mathcal{G}} \\
 \llbracket [d | p] \rrbracket^{\mathcal{G}} = \llbracket d \rrbracket^{\mathcal{G}} \text{ fact } \{ \llbracket p \rrbracket \}
 \end{array}$$

where $\tau = \text{GIVEN } i_1 \times \dots \times \text{GIVEN } i_n$

Fig. 6. Transformation rules from Z to Alloy

Next, we demonstrate the utility of our translation by means of an example. Consider an alternative specification of the state space for a birthday book given in Figure 7. The Z given types are translated to Alloy signatures:

sig NAME { } sig DATE { } sig BBOOK { }

In both Alloy and Z, the set of names, the set of dates, and the set of bbooks are guaranteed to be disjoint.

The Z expression $BBOOK \times NAME$ gets transformed by the ISO Z standard's syntactic transformation rules to $\{ [b : BBOOK] \wedge [n : NAME] \bullet (b, n) \}$, where b and n are fresh variables. Its type is $\mathbb{P}(\text{GIVEN } BBOOK \times \text{GIVEN } NAME)$. Strictly speaking, this expression is not part of our language but it is semantically equivalent to $\{ [[b : BBOOK] \wedge [n : NAME] \mid true] \bullet (b, n) \}$, which is part of the

[*NAME*, *DATE*, *BBOOK*]

$known : \mathbb{P}(BBOOK \times NAME)$ $birthday : \mathbb{P}(BBOOK \times NAME \times DATE)$	$\forall b : BBOOK \bullet$ $(\forall n : NAME; d_1, d_2 : DATE \bullet$ $(b, n, d_1) \in birthday \wedge (b, n, d_2) \in birthday \Rightarrow d_1 = d_2)$ \wedge $\{n : NAME \mid (b, n) \in known\} =$ $\{n : NAME \mid \exists d : DATE \bullet (b, n, d) \in birthday\}$
---	---

Fig. 7. Another birthday book specification

language. An additional transformation rule readily allows the above expression. The declaration of the global variable *known* is then translated to:

```
sig BBOOK { known: set NAME }
fact { known in { x: BBOOK, y: NAME | some b: BBOOK, n: NAME |
                { } and x->y = b->n } }
```

In this example, the fact just restates that *known* is a relation between *BBOOK* and *NAME* and so is unnecessary. Similarly, the declaration of *birthday* is translated to (having omitted the unnecessary fact):

```
sig BBOOK { birthday: set NAME->DATE }
```

The translation of the predicate part is straightforward. Only equality is handled somewhat unusually. The ISO Z standard's transformation and type inference rules transform $e_1 = e_2$ to $e_1 \circ \tau \in (\{e_2 \circ \tau\} \circ \mathbb{P} \tau)$. We want to be able to compare sets and relations for equality, for which τ is a powerset type, but the language defined in Figure 5 does not allow expressions of the resulting nested powerset types. So to support equality, we specifically allow membership of a singleton set without recording its type. After merging the signatures for *BBOOK*, the resulting Alloy is as follows:

```
sig NAME, DATE { }
sig BBOOK { known: set NAME, birthday: set NAME->DATE }
fact { all b: BBOOK |
      (all n: NAME, d1, d2: DATE |
       not (b->n->d1 in birthday and b->n->d2 in birthday)
       or d1 = d2)
      and
      {n: NAME | b->n in known} =
      {n: NAME | some d: DATE | b->n->d in birthday}}
```

Note that this is equivalent to the hand-translation of schema *BBook* to a signature *BBook* as shown in Figure 11. The only difference is that the constraints

on BBook are more concisely expressed using multiplicity keywords and signature facts; the constraints on BBOOK given here use Alloy's fact keyword instead. Possible instances of this model are given in Figure 4. Each instance represents a model for the Z specification in Figure 7.

4 Extending the Translation

We can increase the number of constructs that can be transformed into the language defined in Section 3.

4.1 More Z Constructs

A quantifier with explicit constraints on declarations can be transformed into one without:

$$\forall d \mid p_1 \bullet p_2 \implies \forall d \bullet p_1 \Rightarrow p_2$$

Such a rule should be applied along with the syntactic transformation rules provided by the ISO Z standard. The syntactic transformation rules are applied exhaustively, until no more rules can be applied. That is, the implication introduced by the above rule will be transformed by another rule into a predicate that uses negation and disjunction only, and the disjunction is further transformed into a predicate that uses negation and conjunction only.

The ISO Z Standard [8, Chapter 14] provides additional (so called semantic) transformation rules that define the equivalence of certain sentences of the annotated syntax. For example, rule 14.2.5.4 defines application expressions to be equivalent to a definite description:

$$e_1 e_2 \circ \tau \implies (\mu i : \text{carrier } \tau \mid (e_2, i) \in e_1 \bullet i)$$

In Alloy, an atom is represented by the singleton set containing this atom. This makes it possible to translate a definite description the same way as set comprehension:

$$\llbracket \mu [d \mid p] \bullet e \rrbracket = \llbracket \{ [d \mid p] \bullet e \} \rrbracket$$

The set resulting from a definite description should contain exactly one element, and the Alloy Analyzer could be used to check this. Alternatively, well-definedness of definite descriptions might be assumed when the translation to Alloy is performed. Other semantic transformation rules of the ISO Z standard that can be used to increase the constructs handled by our translation to Alloy are the transformation of free types paragraphs (rule 14.2.3.1) and tuple selection expressions (rule 14.2.5.1). Furthermore, references to and applications of toolkit names can be transformed away by expansion or unfolding according to their definitions.

4.2 Schemas

Manual translations of Z specifications [6,10,15] translate schemas depending on what they are used for. State schemas are usually translated to Alloy signatures; operation schemas are usually translated to Alloy predicates. An automatic translation could try to determine whether a schema is a state or operation schema but is it necessary to make such a distinction? Alloy does not have a fixed idiom for modelling state and operations. Jackson [9, Section 6.2.4], for example, gives an event based specification for a hotel locking system that uses events defined by signatures rather than operations defined by predicates.

A better approach for an automatic translation might be to translate a schema depending on how it is used throughout the Z specification to be translated. Alloy predicates allow reuse of constraints, which resemble the use of Z schemas as predicates. Thus, if a Z schema is used as a predicate, a translation of this schema to an Alloy predicate makes it possible to translate the use of the schema as a predicate straightforwardly by using the corresponding Alloy predicate in the translation. Alloy predicates can also be used when a Z schema is used as a declaration, as can be seen in the following alternative translation of the Z schemas given in Figure 11:

```

pred BBook[known: set NAME, birthday: NAME -> DATE] {
    known in NAME and birthday in NAME->DATE
    all n: NAME | lone n.birthday
    known = dom[birthday]
}

pred Init[known: set NAME, birthday: NAME -> DATE] {
    BBook[known, birthday] and no known
}

pred Add[known: set NAME, birthday: NAME -> DATE,
        known': set NAME, birthday': NAME -> DATE,
        name_in: NAME, date_in: DATE] {
    BBook[known, birthday] and BBook[known', birthday']
    ...
}

```

The authors currently favour a combined approach where a Z schema is represented in Alloy by both a signature and a predicate, and where signatures are shared if possible. For example, the Alloy predicates *BBook* and *Init* above have the same argument list, so can share the signature *S*:

```
sig S { known: set NAME, birthday: NAME -> DATE }
```

The signature can be used in declarations while the predicate can be used in constraints, simulating some of the flexibility that Z schemas provide. For example, the Z predicate $Init \subseteq BBook$ can now be translated by using the signature *S* and the Alloy predicates given above to:


```
{s: S | Init[s.known, s.birthday] } in
  {s: S | BBook[s.known, s.birthday] }
```

The operation Add can be written using signature S:

```
pred Add[s, s': S, name_in: NAME, date_in: DATE] {
  BBook[s.known, s.birthday] and BBook[s'.known, s'.birthday']
  ...
}
```

This is closer to the hand-translation provided in Figure 1 but all occurrences of *known*, *birthday*, *known'*, and *birthday'* in the predicate part of the Z schema *Add* must now be translated to *s.known*, *s.birthday*, *s'.known*, and *s'.birthday* respectively.

The issue of how to handle schema references and the schema calculus in general can be avoided by performing schema expansion prior to translating a Z specification to Alloy. More research is required to determine how practical this approach and other translation approaches mentioned here are.

5 Discussion and Conclusion

This work has been motivated by the desire to make the automatic analysis and visualisation that is available for Alloy accessible to Z users. Alloy, on the one hand, is a simple specification notation that is amenable to automatic analysis by the Alloy Analyzer, which gives immediate feedback by visualising typical scenarios and attempting to find counter-examples to properties that are believed to hold. Z, on the other hand, is a very expressive notation but lacks tool support to make validating specifications as easy and fun as the Alloy Analyzer.

To make use of tools developed for other formal notations, various translations from Z to other notations have been described. Z has been translated to SAL [175] to take advantage of the verification tools that SAL supports. Plagge and Leuschel [14] provide users of Z access to the PROB tool [11], an automated analysis tool-set for the B method. We believe that the visual feedback provided by the Alloy Analyzer is unique and valuable enough to justify yet another translation from Z. See also Aydal, Utting, and Woodcock [2] for a comparison of various modelling tools, including the Alloy Analyzer, ZLive (a Z animator) and ProZ.

In this paper, we have argued that a translation from Z to Alloy performed manually as given in Figure 1 is cumbersome and error-prone. We also raised the question whether such a translation preserves semantics. We have shown in Section 2 that the Z and corresponding Alloy specification given in Figure 1 are not semantically equivalent. In Figure 7 we provided an alternative representation of the birthday book state space in Z, which is semantically equivalent to the Alloy signature *BBook* given in Figure 1. These representations are not equivalent and we have given examples of properties that hold for the original Z specification in Figure 1 but do not hold for the alternative Z representation from Figure 7, nor the corresponding Alloy specification.

The main contribution of this paper is a formalisation of a semantics preserving translation of a subset of Z to Alloy. This provides the formal basis for our Z to Alloy translator, a practical tool that we are developing as part of CZT [12]. It can already handle specifications containing simple constructs and schemas as discussed in this paper. This paper has also considered several translation approaches for schemas. We are currently investigating how complex Z sets and relations (sets of sets) can be handled by our tool. Jackson [9, Chapter 3.2.3, page 41] shows how complex (non flat) relations can be reformulated as flat relations. This requires the introduction of new sets of atoms to model the constructs used within other sets or relations. This is similar to our handling of schemas and poses the same problems. Canonicalisation and generator axioms might be required to avoid spurious counter-examples for certain types of properties.

In the future, we plan to extend our tool to handle more Z constructs. In so doing, we plan to examine the relationship between different translation approaches and the properties that can be checked as well as what influence the translation has on the intelligibility of the resulting Alloy specification and the examples and counter-examples produced by the Alloy Analyzer. We also plan to prove formally the translation described in Section 3 to be semantics preserving.

Acknowledgements

The authors would like to thank Ian Toyn, Hugh Anderson, Mukhlis Matti for their helpful comments and inspiring discussions.

References

1. Arthan, R.: Proofpower, <http://www.lemma-one.com/ProofPower/>
2. Aydal, E.G., Utting, M., Woodcock, J.: A comparison of state-based modelling tools for model validation. In: Proceedings of Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. LNBP, vol. 11. Springer, Heidelberg (2008)
3. Bolton, C.: Using the Alloy analyzer to verify data refinement in Z. *Electronic Notes in Theoretical Computer Science* 137, 23–44 (2005)
4. Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.): ABZ 2008. LNCS, vol. 5238. Springer, Heidelberg (2008)
5. Derrick, J., North, S., Simons, T.: Issues in implementing a model checker for Z. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 678–696. Springer, Heidelberg (2006)
6. Estler, H.-C., Wehrheim, H.: Alloy as a refactoring checker? *Electronic Notes in Theoretical Computer Science* 214, 331–357 (2008)
7. Hewitt, M.A., O’Halloran, C.M., Sennett, C.T.: Experiences with PiZA, an animator for Z. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 37–51. Springer, Heidelberg (1997)
8. ISO/IEC 13568. Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. ISO/IEC (2002); First Edition 2002-07-01
9. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge (2006)

10. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Börger, et al. (eds.) [4], pp. 294–308
11. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.* 10(2), 185–203 (2008)
12. Malik, P., Utting, M.: CZT: A framework for Z tools. In: Treharne, et al. (eds.) [21], pp. 65–84
13. ORA Canada. Z/EVES version 1.5: An overview. In: Hutter, D., Traverso, P. (eds.) *FM-Trends 1998*. LNCS, vol. 1641, pp. 367–376. Springer, Heidelberg (1999)
14. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Davies, J., Gibbons, J. (eds.) *IFM 2007*. LNCS, vol. 4591, pp. 480–500. Springer, Heidelberg (2007)
15. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. *Formal Aspects of Computing* 20(1), 21–39 (2008)
16. Reeve, G., Reeves, S.: Experiences using Z animation tools. Technical Report 01/3/2001, Department of Computer Science, University of Waikato (2001)
17. Smith, G., Wildman, L.: Model checking Z specifications using SAL. In: Treharne, et al. (eds.) [21]
18. Spivey, J.M.: *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire (1992)
19. Spivey, M.: The fuzz type-checker for Z, <http://spivey.oriel.ox.ac.uk/mike/fuzz/>
20. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
21. Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.): *ZB 2005*. LNCS, vol. 3455. Springer, Heidelberg (2005)
22. Utting, M., Malik, P.: Unit testing of Z specifications. In: Börger, et al. (eds.) [4], pp. 309–322

B-ASM: Specification of ASM *à la* B

David Michel^{1,*}, Frédéric Gervais², and Pierre Valarcher²

¹ LIX, CNRS, Polytechnique School, 91128 Palaiseau, France
dmichel@lix.polytechnique.fr

² LACL, Université Paris-Est

IUT Sénart Fontainebleau, Dpt. informatique, 77300 Fontainebleau, France
{frederic.gervais,pierre.valarcher}@univ-paris-est.fr

Summary. We aim at extending the B language in order to build ASM programs which are correct with respect to B-like logical specifications. On the one hand, the main strengths of the B formal method are: i) the ability to express logical statements, and ii) the construction of a correct implementation by refinement. On the other hand, from our viewpoint, the striking aspects of ASM are the non-bounded outer loop that can reach the fixed point of a program and the power to express naturally any kind of (sequential) algorithms.

We introduce a new specification language, called B-ASM, attempting to bridge the gap between these two languages, by taking advantage of the strengths of each approach (B-ASM programs are defined in the same way as in ASM programs, but the language used to define transition functions is enriched with operations akin to some non-deterministic B substitutions). Our leitmotiv is to build an ASM which is correct with respect to a B-like specification. In that aim, we have extended the syntax and the semantics of B to take the non-bounded iteration into account. Moreover, the reuse of the well-founded theoretical relation of refinement from the B method is then straightforward. Rather than directly writing a complex ASM program, one can first specify the required logical properties of the program in a B-ASM specification. Then, we are able to build from the latter a correct ASM program, by proving the proof obligations (PO) associated to each refinement step. For instance, if we can determine a variant in the B-ASM specification for the outer loop, then the ASM program obtained by refinement is guaranteed to terminate.

Contribution. *Let M be a B-ASM machine, then we can construct an ASM which is correct with respect to M .*

At the end of the process a new B_0 program is obtained following strictly the syntax of a π program of an ASM, moreover the process has followed the proof correctness of B method refinement.

The challenge is now, to verify the efficiency of the new method in a real case study and of course, to develop tools.

* This author has been supported by the ANR-09-JCJC-0098-01 MaGiX project together with the Digiteo 2009-36 HD grant and région Île-de-France.

A Case for Using Data-Flow Analysis to Optimize Incremental Scope-Bounded Checking

Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry

The University of Texas at Austin
{dshao, dgopinath, khurshid, perry}@ece.utexas.edu

In software verification, *scope-bounded* checking of programs has become an effective technique for finding subtle bugs. Given bounds (that are iteratively relaxed) on input size and length of execution paths, a program and its correctness specifications are translated into a formula, which is solved using off-the-shelf solvers – a solution to the formula is a counterexample to the correctness specification.

The scalability and effectiveness of scope-bounded checking in bug finding critically depends on the capabilities of the underlying constraint solvers. Traditional approaches [1, 2] translate the bounded code segment of the program and its specification into *one* input formula. For non-trivial programs, the translated formulas can be quite complex and the solvers can fail to find a counterexample in a desired amount of time. When a solver times out, typically there is no information about the likely correctness of the program checked or the coverage of the analysis completed.

To scale scope-bounded checking, our previous work [3] introduced an *incremental* approach that uses the program’s *control-flow* as a basis of splitting the program and generating several sub-formulas, which represent simpler problem instances for the underlying solvers. The key insight of our incremental approach is a “sliding rule” that allows controlling the complexity of the sub-formulas to check based on the capabilities of the underlying solvers. Our previous work supports *splitting strategies* to embody the sliding rule. However, this work uses solely the program’s *control-flow* to define the strategies, and is therefore limited to the syntactical structure of the program and fails to exploit the program semantics.

Recently, we have developed a new approach that utilizes the program’s *data-flow*, specifically *variable-definitions*, to further reduce the solvers’ workload. Specifically, we split the program using different definitions of the same variable, which leads to a reduction in the number of variables in the resulting formulas.

Initial experimental results show that the use of data-flow provides a significant reduction in the number of variables in the encoded formulas over our previous control-flow-based incremental approach. We believe incremental algorithms hold much promise and their application with parallel algorithms in synergy is likely to scale scope-bounded checking to real applications.

Acknowledgment. This material is based upon work funded in part by NSF (Grants IIS-0438967, CCF-0702680, and CCF-0845628) and AFOSR (FA9550-09-1-0351).

References

- [1] Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: ISSTA (2006)
- [2] Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (2000)
- [3] Shao, D., Khurshid, S., Perry, D.E.: An incremental approach to scope-bounded checking using a lightweight formal method. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009: Formal Methods. LNCS, vol. 5850, pp. 757–772. Springer, Heidelberg (2009)

On the Modelling and Analysis of Amazon Web Services Access Policies

David Power, Mark Slaymaker, and Andrew Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

Cloud computing is a conceptual paradigm that is receiving a great deal of interest from a variety of major commercial organisations. By building systems which run within cloud computing infrastructures, problems related to scalability and availability can be reduced. At the time of writing, Amazon Web Services (AWS) [1] is one of the most widely used infrastructures. AWS consists of a number of different components, which can be used in combination or alone. One usage model is to use Elastic Compute Cloud instances to process information and to use the Simple Queue Service (SQS) to handle requests and responses.

If all of the sub-components of a system use the same security credentials, it is possible to restrict access using an ‘all-or-nothing’ approach. However, there are situations where more complex controls are appropriate. For this reason, the AWS access policy language was introduced, which enables access to be restricted based on a number of factors, such as the current time, the originating IP address, the action that is to be performed and the resource that is to be acted upon.

As the complexity of access control policies increases, there is a corresponding increase in the risk that a mistake might be made when defining these policies. In this paper we seek to reduce that risk with the appropriate application of formal methods. We use a hybrid approach of using both the Z specification language and the Alloy modelling language.

We have built formal models of the access policy language used within the AWS cloud computing infrastructure [2]. Specifically, we have explored policies written for SQS. Using the Alloy Analyzer we have been able to explore properties of specific combinations of policies. We have also been able to use the Alloy Analyzer to assist in the construction of new policies by using sets of requests which result in known access control decisions.

Previous work in this area has centred around simple access control systems such as Role-Based Access Control. Attempts at modelling the significantly more complex XACML have resulted in partial models. In this paper we present a complete model of the AWS access policy language, making it possible to analyse existing real-world systems.

References

1. Amazon.com: Amazon Web Services (2009), <http://aws.amazon.com/>
2. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On the modelling and analysis of amazon web services access policies. Technical Report RR-09-15, Oxford University Computing Laboratory (2009)

Architecture as an Independent Variable for Aspect-Oriented Application Descriptions

Hamid Bagheri and Kevin Sullivan

University of Virginia
151 Engineer's Way
Charlottesville, VA 22903 USA
{hb2j,sullivan}@virginia.edu

Software architecture researchers have long assumed that architecture independent application descriptions can be mapped to architectures in many styles, that results vary in quality attributes, and that the choice of a style is driven by consideration of such attributes. In our previous work [1], we demonstrated the feasibility of formally treating architectural style as an independent variable. Given an application description and architectural style description in Alloy [3], we *map* them to software architecture description that refines the given application in conformance with the given style. To represent a map, we extend a traditional architectural style description (in Alloy) with predicates for mapping application descriptions in a given style to architectural descriptions in the given style. These predicates take application descriptions as parameters and define relationships required to hold between them and computed architectural descriptions. Given an application description, and a map, Alloy computes corresponding architectural descriptions guaranteed to conform to the given architectural style. This paper extends our earlier work to aspect-oriented structures. In doing so, we describe an aspect-enabled application description style and a map taking application descriptions in this style to pipe-and-filter architectures. We use the Alloy Analyzer to compute architecture descriptions, represented as satisfying solutions to the constraints of a map given an application description. The *A2A* transformer application, developed in our research group, then converts the Alloy-computed architecture to an architecture description in a traditional architecture description language (ADL): here, AspectualACME [2].

References

1. Bagheri, H., Song, Y., Sullivan, K.: Architecture as an independent variable. Technical report CS-2009-11, University of Virginia Department of Computer Science (November 2009)
2. Garcia, A., Chavez, C., Batista, T., Santanna, C., Kulesza, U., Rashid, A., Lucena, C.: On the modular representation of architectural aspects. In: Proceedings of the European Workshop on Software Architecture, Nantes, France. LNCS, pp. 82–97 (2006)
3. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11(2), 256–290 (2002)

ParAlloy: Towards a Framework for Efficient Parallel Analysis of Alloy Models

Nicolás Rosner, Juan P. Galeotti,
Carlos G. Lopez Pombo, and Marcelo F. Frias

Department of Computer Science, FCEyN, Universidad de Buenos Aires,
{nrosner, clpombo, jgaleotti, mfrias}@dc.uba.ar

Alloy [\[Jac02a\]](#) is a widely adopted relational modeling language. Its appealing syntax and the support provided by the Alloy Analyzer [\[Jac02b\]](#) tool make model analysis accessible to a public of non-specialists. A model and property are translated to a propositional formula, which is fed to a SAT-solver to search for counterexamples. The translation strongly depends on user-provided bounds for data domains called scopes – the larger the scopes, the more confident the user is about the correctness of the model. Due to the intrinsic complexity of the SAT-solving step, it is often the case that analyses do not scale well enough to remain feasible as scopes grow.

ParAlloy exploits the possibility of splitting the SAT formula, thus allowing for parallel SAT-solving of Alloy models. Three of its important characteristics are:

1. Its core component is a parallel solver for arbitrary propositional formulas –not necessarily in CNF– based on problem decomposition, and making a novel use of BEDs [\[AH02\]](#) for subproblem representation and manipulation, Minisat [\[ES03\]](#) for subproblem analysis, and MPI [\[SOHL⁺98\]](#) for inter-process communication.
2. Its Alloy-specific enhancements further improve (parallel) analyzability by using knowledge obtained from the models to assist splitting decisions.
3. For valid properties (the UNSAT case), the speedups allowed the analysis of Alloy properties (such as some assertions in [\[Zay06\]](#)) that exceed the current capabilities of the Alloy Analyzer. For invalid properties, test case generation or iterative model refinement (the SAT case), parallel analysis of search space paths often leads to much higher speedups, since its exhaustion is unnecessary.

References

- [AH02] Andersen, H.R., Hulgaard, H.: Boolean expression diagrams. *Information and computation* 179(2), 194–212 (2002)
- [ES03] Eén, N., Sörensson, N.: An extensible sat solver. In: Giunchiglia, E., Tacchella, A. (eds.) *SAT 2003*. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

- [Jac02a] Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11(2), 256–290 (2002)
- [Jac02b] Jackson, D.: *A micromodels of software: Lightweight modelling and analysis with Alloy*. Computer Science and Artificial Intelligence Laboratory. MIT, Cambridge (2002)
- [SOHL⁺98] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: *MPI: The complete reference*. MIT Press, Cambridge (1998)
- [Zav06] Zave, P.: Compositional binding in network domains. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) *FM 2006*. LNCS, vol. 4085, pp. 332–347. Springer, Heidelberg (2006)

Introducing Specification-Based Data Structure Repair Using Alloy

Razieh Nokhbeh Zaeem and Sarfraz Khurshid

University of Texas, Austin TX 78712, USA
{rnokhbehzaeem, khurshid}@ece.utexas.edu

While several different techniques utilize specifications to check correctness of programs before they are deployed, the use of specifications in deployed software is more limited, largely taking the form of runtime checking where assertions form a basis for detecting erroneous program states and terminating erroneous executions in failures. Recent approaches [1] proposed constraint-based repair where data structure constraints are used to repair erroneous states. However, data structure constraints are too weak a form of specification for error recovery in general. We have developed a specification-based approach for data structure repair, which allows repairing erroneous executions in deployed software by repairing erroneous states. The key novelty is our support for rich behavioral specifications, such as those that relate pre-states with post-states to accurately specify expected behavior and hence to enable precise repair.

We address the following repair problem: Let ϕ be a method postcondition that relates pre- and post-states such that $\phi(r, t)$ if and only if pre-state r and post-state t satisfy the post-condition. Given a valid pre-state u , and an invalid post-state s (i.e., $!\phi(u, s)$), mutate s into state s' such that $\phi(u, s')$.

Our approach views a specification as a non-deterministic implementation, which may permit a high degree of non-determinism. The Alloy tool-set [2] provides the enabling technology for writing specifications and systematically repairing erroneous states. One initial technique that we developed is to transform the repair problem into a constraint solving problem and leverage the Alloy tool-set as a solving machine, ignoring the erroneous state. Although this technique provides a correct output, it is likely infeasible for larger states. Our key insight to improve this technique is to use any correct state mutations by an otherwise erroneous execution to prune the non-determinism in the specification, thereby transmuted the specification to an implementation that does not incur a prohibitively high performance penalty. Moreover, using the faulty post-state as the starting point of the repair process avoids unnecessary perturbations during the repair process. We are working on extensions of this idea to build an effective and efficient repair framework that supports rich behavioral specifications.

Acknowledgment

This material is based upon work funded in part by the NSF (IIS-0438967, CCF-0702680, and CCF-0845628), and the AFOSR (FA9550-09-1-0351).

References

1. Elkarablieh, B., Garcia, I., Suen, Y.L., Khurshid, S.: Assertion-based Repair of Complex Data Structures. In: ASE (2007)
2. Jackson, D.: Software Abstractions: Logic, Language and Analysis. The MIT Press, Cambridge (2006)

Secrecy UML Method for Model Transformations

Wael Hassan¹, Nadera Slimani², Kamel Adi², and Luigi Logrippo²

¹ University of Ottawa, 4051D-800 King Edward, Ottawa, Ontario, K1N-6N5

² Université du Québec en Outaouais, Gatineau, Québec, Canada
wael@acm.org, {s1in02,Kamel.Adi,luigi}@uqo.ca

This paper introduces the subject of secrecy models development by transformation, with formal validation. In an enterprise, constructing a secrecy model is a participatory exercise involving policy makers and implementers. Policy makers iteratively provide business governance requirements, while policy implementers formulate rules of access in computer-executable terms. The process is error prone and may lead to undesirable situations thus threatening the security of the enterprise. At each iteration, a security officer (SO) needs to guarantee business continuity by ensuring property preservation; as well, he needs to check for potential threats due to policy changes. This paper proposes a method that is meant to address both aspects: the formal analysis of transformation results and the formal proof that transformations are property preserving. UML is used for expressing and transforming models [1], and the Alloy analyzer is used to perform integrity checks [3]. Governance requirements dictate a security policy, that regulates access to information. This policy is implemented by means of secrecy models. Hence, the SO defines the mandatory secrecy rules as a part of enterprise governance model in order to implement security policy. For instance, a secrecy rule may state: *higher-ranking officers have read rights to information at lower ranks*. Automation helps reduce design errors of combined and complex secrecy models [2]. However, current industry practices do not include precise methods for constructing and validating enterprise governance models. Our research proposes a formal transformation method to construct secrecy models by way of applying transformations to a base UML model (BM). For example, starting from the BM, with only three primitives: Subject/Verb/Object, we can generate RBAC0 in addition to SecureUML [2] model. By way of examples and by means of formal analysis we intend to show that, using our method, a SO is able to build different types of secrecy models and validate them for consistency, in addition to detecting scenarios resulting from unpreserved properties.

References

1. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. *Artif. Intell.*, 70–118 (2005)
2. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. *Softw. Eng. Methodol.*, 39–91 (2006)
3. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On Challenges of Model Transformation from UML to Alloy. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)

Improving Traceability between KAOS Requirements Models and B Specifications

Abderrahman Matoussi¹ and Dorian Petit²

¹ LACL, University Paris-Est

`abderrahman.matoussi@univ-paris12.fr`

² LAMIH, University Lille Nord de France UVHC

`dorian.petit@univ-valenciennes.fr`

The aim of this paper is to give some feedback about the B specification [1] of a localization software component which is one of the most critical parts in the land transportation system. The main difficulties when we develop a localization component is: (i) to find the correct algorithm that merges positioning data (ii) to take into account all the properties we have to deal with. At this stage, we think that a semi formal model such as KAOS [2], a goal-based requirements engineering method, will be very useful in order to have guidelines on how to do. For that, we will just focus on the architecture of the B specifications and how using KAOS help us to build it. Since goals play an important role in requirements engineering process, rather than establishing traceability from the KAOS requirements model as a whole, we propose to establish traceability from individual goals that are part of the KAOS goal model. The main idea is to specify a correspondence rule between each concept of the goal model and B elements. Up to now, we consider only functional goals of type *Achieve* [2]. A B machine is associated to each goal. This machine contains an operation that “realizes” the goal; i.e. it describes the “work” to perform to reach the goal, in terms of generalized substitutions. The refinement of a goal is represented by a B refinement machine that refines the machine; the abstract operation is refined by a concrete one. This operation is built by combining operations of the machines that correspond to the sub-goals of the more abstract goal and are included in the B machine via the inclusion relationship. The nature of the combination depends on the goal refinement pattern (Milestone, AND, OR). The reader can refer to [3] for more details. The main contribution of our approach is that it establishes the first brick toward the construction of the bridge between the non-formal and the formal worlds as narrow and concise as possible. Furthermore, by discharging the proof obligations generated by the B refinement process, we can prove some properties of consistency on the goal model. Regarding the different KAOS goal model concepts, we need now to consider the translation of the concepts of domain properties and non functional goals.

References

1. Abrial, J.R.: The B-Book: Assigning programs to meanings. CUP (1996)
2. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
3. Matoussi, A., Laleau, R., Petit, D.: Bridging the gap between KAOS requirements models and B specifications. Technical Report TR-LACL-2009-5, LACL, University of Paris-Est (2009),
<http://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2009-5.pdf>

Code Synthesis for Timed Automata: A Comparison Using Case Study

Anaheed Ayoub¹, Ayman Wahba², Ashraf Salem¹, and Mohamed Sheirah²

¹ Mentor Graphics Egypt

² Ain Shams University

There are two available approaches to automatically generate implementation code from timed automata model. The first approach is implemented and attached to TIMES tool [1]. We will call this approach "TIMES approach". While the second approach is based on using B-method [2] and its available code generation tool [3]. We will call this approach "B-method approach". We select the model of the production cell to be used as a case study for the comparison between these two approaches. The same production cell model has been used against both approaches. The B-method approach generates platform independent code [4]. So we select the generated code using TIMES to be platform independent too for the comparison purpose. For the B-method approach, we use the deterministic semantic of timed automata which is used for TIMES code generation as given in [5]. This semantic controls the selection of the next executed function. The using of this deterministic mechanism is generally not needed for the code generated by the B-method approach. But we use it as it is the implemented mechanism for the TIMES approach. So we select to use it for comparison purpose. By running the implementation code generated using the B-method approach, it works fine as far as we run and no property violation could be found. On the other hand the code generated using TIMES approach runs successfully for the first 10 action transitions and then it progresses the time infinitely. This means that the system deadlocked, so it violates the first property of the model. While the first property is to guarantee that the system is deadlock free. This deadlock is due to the mishandling of the committed and urgent states [6]. The introduced comparison gave a result that the approach based on the using of B-method generates a verified code (by mean of simulation) and handles more timed automata features.

References

1. <http://www.it.uu.se/research/group/darts/times/papers/manual.pdf>
2. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. http://www.tools.clearsy.com/index.php5?title=Tutorial_ComenC
4. Ayoub, A., Wahba, A., Salem, A., Taher, M., Sheirah, M.: Automatic Code Generation from Verified Timed Automata Model. To be appear in proceeding of IADIS Applied Computing, Italy (2009)
5. Amnell, T., Fersman, E., Pettersson, P., Sun, H., Yi, W.: Code Synthesis for Timed Automata. Nordic Journal of Computing 9(4) (2002)
6. <http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf>

Towards Validation of Requirements Models

Atif Mashkoor¹ and Abderrahman Matoussi²

¹ LORIA – Nancy Université
{firstname.lastname}@loria.fr

² LACL – Université Paris-Est
{firstname.lastname}@univ-paris12.fr

The use of formal methods for software development is escalating over the period of time. The input to this formal specification phase is often the documents obtained during the requirements analysis activity which are either textual or semi-formal. Now there is a traceability gap between analysis and specification phases as verification of the semi-formal analysis model is difficult because of poor understandability of lower level of formalism of verification tools and validation of the formal specification is difficult for customers due to their inability to understand formal models. Our objective is to bridge this gap by a gradual introduction of formalism into the requirement model in order to facilitate its validation. We analyse our requirements with KAOS (Knowledge Acquisition in autOMated Specification) [1] which is a goal-oriented methodology for requirements modeling, then we translate the KAOS goal model, following our derived precise semantics [3], into an Event-B [2] formal specification, and finally we rigorously animate the obtained specification in order to validate its conformance to original requirements with the approach defined in [4]. The whole scenario is summed up by figure 1. By the validation of a semi-formal requirement model by rigorous animation of its formal counterpart, we reap benefits at two levels: customers can be involved into the development right from the start and consequently the requirement errors can be detected right on the spot. At theoretical level, we have obtained some initial results at analysis, specification and validation levels independently, however we hope that our proposed combined approach is also feasible collectively.

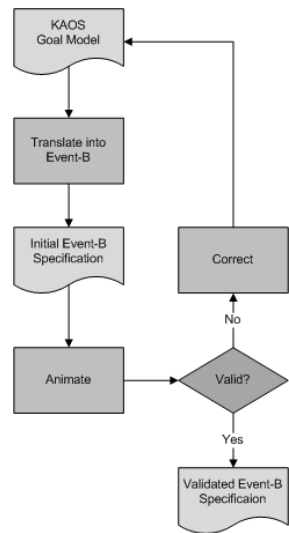


Fig. 1. The requirements validation process

References

1. Van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. CUP (2009)
3. Matoussi, A.: Expressing KAOS Goal Models with Event-B. In: Doctoral Symposium of 16th International Symposium on Formal Methods, Eindhoven, The Netherlands (2009)
4. Mashkoor, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems, York, UK (2009)

A Proof Based Approach for Formal Verification of Transactional BPEL Web Services

Idir Aït Sadoune and Yamine Aït Ameur

LISI/ENSMA-UP

Téléport 2 -1, avenue Clément Ader, BP 40109, 86961, Futuroscope-Poitiers, France
{idir.aitsadoune,yamine}@ensma.fr

The Service-Oriented Architectures (SOA) are increasingly used in various application domains. Nowadays various Services operate on the Web and access various critical resources such as databases. These services are called transactional web services when they perform transactional actions. This kind of Services must verify the relevant constraints related to transactional systems. In our work, we focus on web services described with BPEL [1].

In the BPEL language, a composite Web Service is implemented by a process that consists of activities such as the messaging activities *invoke* and *reply*, used for interacting with other web services and the structured activities *sequence*, *flow* and *scope*, acting as containers for their nested activities. BPEL provides some support for transactions through its *fault* and *compensation* handlers undoing the effects of completed activities.

In most related work [2,3], validation of the web services composition and workflow shows how to model transactional behaviors and involves the verification of behavioral properties. In our work, we sketch a methodology showing how Event.B models [4,5] obtained by the approach described in [6] can be used to prove web services transactional properties. Transactional services that access and manage critical resources are isolated in a *scope* element with compensation and fault handlers. When modelling fault and compensation handlers by a set of events, it becomes possible to model and check the properties related to transactional web services. These properties are encoded in the INVARIANTS clause in order to guarantee consistency of the manipulated resources.

References

1. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Standard (April 2007), <http://docs.oasis-open.org/wsbpel/2.0/0S/wsbpel-v2.0-0S.html>
2. He, Y., Zhao, L., Wu, Z., Li, F.: Formal Modeling of Transaction Behavior in WS-BPEL. In: International Conference on Computer Science and Software Engineering, CSSE 2008 (2008)
3. Guidi, C., Lucchi, R., Mazzara, M.: A Formal Framework for Web Services Coordination. ENTCS 180(2), 55–70 (2007)

4. Metayer, C., Abrial, J.-R., Voisin, L.: Event-B Language. Project IST-511599. RODIN (2005)
5. ClearSy: Rodin (2007),
http://www.methode-b.com/php/travaux_r&d_methode_b_projet_RODIN_fr.php
6. Aït-Sadoune, I., Aït-Ameur, Y.: A Proof Based Approach for Modelling and Verifying Web Services Compositions. In: 14th IEEE International Conference on Engineering of Complex Computer Systems ICECCS 2009, Potsdam, Germany, June 2-4, pp. 1-10 (2009)

On an Extensible Rule-Based Prover for Event-B

Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh

ECS, University of Southampton, Southampton SO17 1BJ, UK
{im06r,mjb,ae2,ra3}@ecs.soton.ac.uk

Abstract. Event-B [1] is a formalism for discrete system modelling. The Rodin platform [2] provides a toolset to carry out specification, refinement and proof in Event-B. The importance of proofs as part of formal modelling cannot be emphasised enough, and as such, it is imperative to provide effective tool support for it. An important aspect of this support is the extensibility of the prover, and more pressingly, how its soundness is preserved while allowing extensibility. Rodin has a limited support for adding rules as this requires (a) a deep understanding of the internal architecture and (b) knowledge of the Java language. Our approach attempts to provide support for user-defined proof rules. We initially focus on supporting rewrite rules to enhance the rewriting capabilities of Rodin. To achieve this objective, we introduce a *theory* construct distinct from contexts and machines. The theory construct provides a platform for the users to define rewrite rules both conditional and unconditional. As part of rule definition, users decide whether the rule is to be applied automatically or interactively. Each defined rule gives rise to proof obligations that serve to verify its conservativity. In this respect, it is required that validity and well-definedness are preserved by rules. After the conservativity of all rules contained in a theory is established, the theory can then be deployed and available to the proving activity. In order to apply rewrite rules, it is necessary to single out applicable rules to any given sequent. This is achieved through a pattern matching mechanism which is implemented as an extension to Rodin. Our approach has two advantages. Firstly, it offers a uniform mechanism to add proof rule without the need to write Java code. Secondly, it provides a means to verify added rules using proof obligations. Our work is still in progress, and research has to be carried out to (a) cover a larger set of rewrite and inference rules, and (b) provide guidelines to help the theory developer with deciding whether a given rule should be applied automatically.

References

1. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.* 77(1-2), 1–28 (2007)
2. Butler, M., Hallerstede, S.: The Rodin Formal Modelling Tool. In: BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London (December 2007)

B Model Abstraction Combining Syntactic and Semantic Methods

Jacques Julliand¹, Nicolas Stouls², Pierre-Christophe Bué¹,
and Pierre-Alain Masson¹

¹ LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex
{bue, julliand, masson}@lifc.univ-fcomte.fr

² Université de Lyon, INRIA
INSA-Lyon, CITI, F-69621, Villeurbanne, France
nicolas.stouls@insa-lyon.fr

Abstract. In a model-based testing approach as well as for the verification of properties by model-checking, B models provide an interesting solution. But for industrial applications, the size of their state space often makes them hard to handle. To reduce the amount of states, an abstraction function can be used, often combining state variable elimination and domain abstractions of the remaining variables. This paper illustrates a computer aided abstraction process that combines syntactic and semantic abstraction functions. The first function syntactically transforms a B event system M into an abstract one A , and the second one transforms a B event system into a Symbolic Labelled Transition System (SLTS). The syntactic transformation suppresses some variables in M . This function is correct in the sense that A is refined by M . A process that combines the syntactic and semantic abstractions has been experimented. It significantly reduces the time cost of semantic abstraction computation. This abstraction process allows for verifying safety properties by model-checking or for generating abstract tests. These tests are generated by a coverage criteria such as all states or all transitions of an SLTS.

Keywords: Model Abstraction, Syntactic Abstraction, Refinement.

The full version of this short paper is available as a research report:

Reference

- [JSBM09] Julliand, J., Stouls, N., Bué, P.-C., Masson, P.-A.: B model abstraction combining syntactic and semantics methods. Research Report RR2009-04, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, 15 pages (November 2009)

A Basis for Feature-Oriented Modelling in Event-B

Jennifer Sorge, Michael Poppleton, and Michael Butler

Electronics and Computer Science, University of Southampton
{jhs06r, mmp, mjb}@ecs.soton.ac.uk

Feature-oriented modelling is a well-known approach for Software Product Line (SPL) development. It is a widely used method when developing groups of related software. With an SPL approach, the development of a software product is quicker, less expensive and of higher quality than a one-off development since much effort is re-used. However, this approach is not common in formal methods development, which is generally high cost and time consuming, yet crucial in the development of critical systems. With the increase of more complex critical systems, it becomes more important to apply formal methods to the development cycle, and we propose a method that allows the application of SPL development techniques to formal methods. This results in faster and cheaper development of formal systems.

Our method combines Event-B [1] and feature models [2]. A feature in a feature model represents a requirement of the product family and is formally described in Event-B using special feature modelling patterns. A feature represented in Event-B is referred to as component. We develop composition rules, which allow components to be composed. Special composition proof obligations allow the verification of the composition.

The feature model is formed by features which may be associated with Event-B components. A subset of features from the feature model can be selected to form a feature model instance, thereby selecting several of these Event-B components. These components are composed pair-wise, and composition POs can be discharged to prove properties and to ensure consistency of the composition. The final Event-B machine represents the formal specification which is associated with the feature model instance and is obtained by composing these components.

The motivation of our work is to allow product line development for critical systems. We use traditional product line methods, i.e. feature modelling, and link it with the formal method Event-B.

Future work is focussed on amending feature diagrams to reflect Event-B components more precisely. Refinement patterns will also be addressed.

References

1. Abrial, J.-R.: Modeling in Event-B: Systems and Software Engineering. Cambridge University Press, Cambridge (2009)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

Using Event-B to Verify the Kmelia Components and Their Assemblies

Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix

COLOSS Team
LINA CNRS UMR 6241 - University of Nantes
{firstname.lastname}@univ-nantes.fr

Component-based software engineering is a practical approach to address the issue of building large software by combining existing and new components. However, building reliable software systems from components requires to verify the consistency of components and the correctness of their assemblies.

A Kmelia component is equipped with invariants and with pre/post-conditions defined on services. A Kmelia assembly defines a set of links between required and provided services of various components, with respect to their pre/post-conditions [12]. Among the formal analysis necessary to ensure complete correctness, we consider: (i) the component invariant consistency vs. pre-/post-conditions of services; (ii) the Kmelia assembly link contract correctness, that relates services which are linked in the assemblies. We use the notion of contract as in the classical works and results such as *design-by-contract* or *specification matching*: on the one hand the pre-condition of a required service is stronger than the pre-condition of the linked provided service; on the other hand the post-condition of the provided service is stronger than the post-condition of the linked required service. This motivates the choice for using Event-B to check the consistency of Kmelia components and the correctness of their assembly. We show how to generate the necessary Event-B models from parts of the Kmelia specifications. We design Event-B patterns to guide the translation and build the necessary proof obligations. Then, we describe how the proofs of the Event-B models are linked with the attempted proofs at the Kmelia level: each Kmelia component is proved to be consistent by checking the Event-B invariant preservation on separate models for observable parts and required/provided services. To check each assembly link, appropriate Event-B models are built and then Event-B refinement proof obligations are generated and discharged.

The refinement technique of Event-B is used to manage both the structuring of the generated Event-B models and also the proofs to be discharged. Yet we have applied the technique to small and medium size case studies. The results of the current work constitute one more step for rigorously building components and assemblies using the Kmelia framework.

References

1. Attiogbé, C., André, P., Ardourel, G.: Checking Component Composability. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 18–33. Springer, Heidelberg (2006)
2. André, P., Ardourel, G., Attiogbé, C., Lanoix, A.: Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies. In: FACS 2009 (to be published)

Starting B Specifications from Use Cases

Thiago C. de Sousa¹ and Arylido G. Russo Jr²

¹ University of São Paulo

thiago@ime.usp.br

² AeS Group

agrj@aes.com.br

The B method [1] is gaining visibility in formal methods community due to excellent support for refinement. However, the traceability between the requirements and the formal model is still an issue of this method. To overcome this issue, different solutions have been proposed by researchers. In [2], the authors have presented a traceability between KAOS requirements and B. A mixed solution using natural language and UML-B has been proposed by [3]. However, these approaches use non-standard artifacts for requirement specifications, which we consider a disincentive for convincing designers to adopt formal methods since they must spend time to learn them. So, we propose an approach for mapping requirements to B models from use cases [4], which can be considered as the *de-facto* industry standard for requirement specifications. We propose that use case scenario sentences must be written using a controlled natural language (CNL) described according our use case transaction definition, which is based on Ochodek's transaction model [5]. We consider that a transaction is a sequence of four steps actions in a scenario, which starts from the actors request (U) and finishes with the system response (SR). The system validation (SV) and system expletive (SE) actions must also occur within the starting and ending action. The actions help to find out the B components. So, from SV actions we extract the preconditions and from SE actions we derive the operations names and the postconditions. We are not interested in the automatic translation of use cases for formal specifications since there are many natural language ambiguity problems. The intention of our work is to take the use cases as a guideline for starting B specifications. Our main goal is to create a new and complete development process (including deliverables artifacts), namely *BeVelopment*, for B focusing on agility/usability and we believe that use cases seem to be a good start point.

References

1. Abrial, J.-R.: The B-Book: assigning programs to meaning. C.U.P (1996)
2. Ponsard, C., Dieul, E.: From requirements models to formal specifications in B. In: ReMo2V CEUR Workshop Proceedings, vol. 241. CEUR-WS.org (2006)
3. Jastram, M., Leuschel, M., Bendisposto, J., Russo Jr., A.: Mapping Requirements to B models. DEPLOY Deliverable (2009) (unpublished manuscript)
4. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-oriented software engineering: A use case driven approach. Addison-Wesley, Reading (1992)
5. Ochodek, M., Nawrocki, J.R.: Automatic Transactions Identification in Use Cases. In: Meyer, B., Nawrocki, J.R., Walter, B. (eds.) CEE-SET 2007. LNCS, vol. 5082, pp. 55–68. Springer, Heidelberg (2008)

Integrating SMT-Solvers in Z and B Tools

Alessandro Cavalcante Gurgel^{1,*}, Valério Gutemberg de Medeiros Jr.²,
Marcel Vinicius Medeiros Oliveira¹, and David Boris Paul Déharbe¹

¹ Departamento de Informática e Matemática Aplicada, UFRN, Brazil

² Instituto Federal de Educação, Ciência e Tecnologia, IFRN, Brazil

An important frequent task in both Z and B is the proof of verification conditions (VCs). In Z and B, VCs can be predicates to be discharged as a result of refinement steps, some proof about initialization properties or domain checking. Ideally, a tool that supports any Z and B technique should automatically discharge as many VCs as possible. Here, we present ZB2SMT^[1], a Java package designed to clearly and directly integrate both Z and B tools to the satisfiability module theory (SMT) solvers such as veriT^[2], a first-order logic (FOL) theorem prover that accepts the SMT syntax^[3] as input. By having the SMT syntax as target we are able to easily integrate with further eleven automatic theorem provers. ZB2SMT is currently used by Batcave^[4], an open source tool that generates VCs for the B method and CRefine^[5], a tool that supports the *Circus* refinement calculus. Much of the VCs generated to validate the refinement law applications, are based on FOL predicates. Hence, CRefine uses the ZB2SMT package to automatically prove such predicates. The package integrates elements of Z and B predicates in a common language and transforms these predicates into SMT syntax. In this process, a SMT file is generated containing the predicate and some definitions. It is sent to a chosen SMT solver which yields a Boolean value for the predicate or it can be sent to several SMT solvers in a parallel approach. In order to improve the performance of the proof system, ZB2SMT has a module that can call different instances of solvers at different computers, according to a configuration file. It improves the proof process by allowing different strategies to be performed in parallel, reducing the verification time.

Acknowledgments. This work was partially supported by INES (www.ines.org.br), funded by CNPq grant 573964/2008-4 and by CNPq grants 553597/2008-6, 550946/2007-1, and 620132/2008-6.

References

1. Bouton, T., Caminha, D., de Oliveira, B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: CADE-22, pp. 151–156 (2009)
2. Marinho, E.S., Medeiros Jr., V.G., Tavares, C., Déharbe, D.: Um ambiente de verificação automática para o método B. In: SBMF 2007 (2007)

* The ANP supports the work of the author through the prh22 project.

¹ Freely available at <http://www.consiste.dimap.ufrn.br/projetos/zb2smt>.

3. Oliveira, M.V.M., Gurgel, A.C., de Castro, C.G.: CRefine: Support for the *Circus* Refinement Calculus. In: 6th IEEE on SEFM, pp. 281–290. IEEE, Los Alamitos (2008)
4. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2 (2006)

Formal Analysis in Model Management: Exploiting the Power of CZT

James R. Williams, Fiona A.C. Polack, and Richard F. Paige*

Department of Computer Science, University of York, UK YO10 5DD
{jw,fiona,paige}@cs.york.ac.uk

Software engineering diagrams are hard to verify and formally analyse, often due to inadequately defined diagram semantics: the semantics often does not enable formal analysis, or may be underspecified to a degree that does not allow useful properties to be checked.

The AUtoZ tools (jamesrobertwilliams.co.uk/autoz.php) provide formalisation in the style of commercially-acceptable model management [3]. AUtoZ is an automated framework based on Amálio's GeFoRME, the generative framework for rigorous model-driven engineering [1]. GeFoRME is designed to give semantically-adaptable support to the construction of formal models from diagrams.

Formal methods tools often produce messages aimed at expert users of the tool and relate to line numbers of the formal specification; mapping these messages back to components in UML diagrams is not trivial. To address this, we are creating an AUtoZ instance that targets the *Community Z Tools* (CZT) project (czt.sourceforge.net). The *ZML* sub-project of CZT [2] supports XML markup for Z. CZT tools annotate the ZML file, for instance with issues raised by formal analysis. Exploiting the fact that, in model engineering, a diagrammatic model must conform to a metamodel (that defines abstract syntax and some semantics), and that ZML has a well-defined metamodel, generic associations can be made at the metamodel level. Therefore we can link elements in the UML and ZML models. Traceability links are thus a side-effect of the Z generation.

By combining AUtoZ with CZT's flexible, open-source formal support mechanisms, a complete tool chain has been designed which can overcome many of the problems of interfacing formal analysis with traditional diagram-based software engineering.

References

1. Amálio, N.: Generative frameworks for rigorous model-driven development. PhD thesis, Computer Science, York, UK (2007)
2. Utting, M., et al.: ZML: XML Support for Standard Z. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 437–456. Springer, Heidelberg (2003)
3. Williams, J., Polack, F.: Automated formalisation for verification of diagrammatic models. In: FACS 2009. ENTCS (2009)

* This research was supported by the EPSRC, through the Large-Scale Complex IT Systems project, EP/F001096/1.

Author Index

- Abdallah, Chaouki T. 132
Abrial, Jean-Raymond 319
Adi, Kamel 400
Aït Ameer, Yamine 405
Aït Sadoune, Idir 405
Altenhofen, Michael 47
André, Pascal 410
Arcaini, Paolo 61
Ardourel, Gilles 410
Attigobé, Christian 410
Ayoub, Anaheed 403
- Bagheri, Hamid 395
Börger, Egon 20
Bué, Pierre-Christophe 408
Butler, Michael 189, 231, 407, 409
- Catano, Nestor 259
Cavalcanti, Ana 334
Craig, Iain 20
- Déharbe, David 203, 217
Déharbe, David Boris Paul 412
de Sousa, Thiago C. 411
D'Ippolito, Nicolás 160
Dunne, Steve 302
- Edmunds, Andrew 407
- Farahbod, Roozbeh 47
Filali-Amine, Mamoun 245
Frias, Marcelo F. 160, 396
- Galeotti, Juan P. 160, 396
Gargantini, Angelo 61
Gervais, Frédéric 391
Gomes, Bruno 203
Gopinath, Divya 392
Groves, Lindsay 377
Groß, Gudmund 189
Gurgel, Alessandro Cavalcante 412
- Hallerstede, Stefan 273, 287
Hasan, Osman 2
Hassan, Waël 400
- Hassine, Jameleddine 34
Heileman, Gregory L. 132
Hoang, Thai Son 319
- Iliasov, Alexei 174
Ilic, Dubravka 174
Ireland, Andrew 189
- Jackson, Daniel 1, 118
Jacob, Jeremy L. 105
Julliand, Jacques 408
- Kang, Eunsuk 1
Khoury, Joud 132
Khurshid, Sarfraz 392, 398
- Laibinis, Linas 174
Lanoix, Arnaud 410
Lanzarotti, Esteban 160
Latvala, Timo 174
Lawall, Julia 245
Lenihan, Clare 377
Leuschel, Michael 231, 287
Logrippe, Luigi 400
Lopez Pombo, Carlos G. 396
- Maamria, Issam 407
Malik, Petra 377
Mashkoo, Atif 404
Masson, Pierre-Alain 408
Matoussi, Abderrahman 401, 404
Medeiros Jr., Valério Gutemberg de 412
Mera, Sergio 160
Merle, Philippe 91
Michel, David 391
Moraes, Katia 203
Moreira, Anamaria 203
- Near, Joseph P. 118
Nokhbeh Zaeem, Razieh 398
- Oliveira, Marcel Vinicius Medeiros 412
- Paige, Richard F. 414
Perry, Dewayne E. 392
Petit, Dorian 401

- Plagge, Daniel 287
Polack, Fiona A.C. 414
Poppleton, Michael 409
Power, David 349, 394
- Reynolds, Mark C. 146
Rezazadeh, Abdolbaghi 407
Riccobene, Elvinia 61, 75
Romanovsky, Alexander 174
Rosner, Nicolás 396
Rueda, Camilo 259
Russo Jr, Aryldo G. 411
- Salem, Ashraf 403
Scandurra, Patrizia 75
Seinturier, Lionel 91
Shao, Danhua 392
Sheirah, Mohamed 403
Simpson, Andrew 349, 363, 394
Slaymaker, Mark 349, 394
- Slimani, Nadera 400
Sorge, Jennifer 409
Stouls, Nicolas 408
Sullivan, Kevin 395
- Tahar, Sofiène 2
Tiberghien, Alban 91
Troubitsyna, Elena 174
Turner, Edd 231
- Valarcher, Pierre 391
Varpaaniemi, Kimmo 174
Vernon, Michael 334
- Wahba, Ayman 403
Williams, James R. 414
Wu, Nicolas 363
- Zeyda, Frank 302, 334