

Festschrift

LNCS 5115

Jean-Raymond Abrial  
Uwe Glässer (Eds.)

# Rigorous Methods for Software Construction and Analysis

**Essays Dedicated to Egon Börger  
on the Occasion of His 60th Birthday**



 Springer

*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*

Jean-Raymond Abrial Uwe Glässer (Eds.)

# Rigorous Methods for Software Construction and Analysis

Essays Dedicated to Egon Börger  
on the Occasion of His 60th Birthday

Volume Editors

Jean-Raymond Abrial  
Marseille, France  
E-mail: jrabrial@neuf.fr

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

The illustration appearing on the cover of this book is the work  
of Daniel Rozenberg (DADARA).

Library of Congress Control Number: 2009942153

CR Subject Classification (1998): F.1, F.2.1-2, F.4.1, F.3, D.2.4, D.2-3, I.2.2

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

ISSN 0302-9743  
ISBN-10 3-642-11446-6 Springer Berlin Heidelberg New York  
ISBN-13 978-3-642-11446-5 Springer Berlin Heidelberg New York

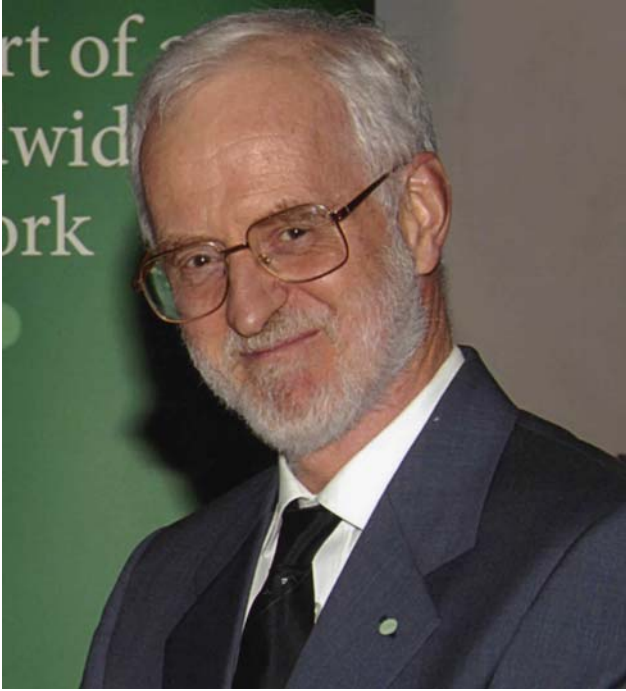
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication 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 2009  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper SPIN: 12821952 06/3180 5 4 3 2 1 0

## Preface



Egon Börger

# Tribute to Egon Börger on the Occasion of his 60<sup>th</sup> Birthday

Jean-Raymond Abrial<sup>1</sup> and Uwe Glässer<sup>2</sup>

<sup>1</sup> jrabrial@neuf.fr

<sup>2</sup> glaesser@cs.sfu.ca

Egon Börger was born on May 13, 1946, in Westfalia (Germany). After the classic baccalauréat, from 1965-1971 he studied philosophy, logic and mathematics at the Sorbonne (Paris, France), Institut Supérieur de Philosophie de Louvain (Belgium), Université de Louvain and Universität Münster (Germany), where he got his doctoral degree and in 1976 his “Habilitation” in mathematics. The themes of his doctoral dissertation, *Reduction classes in Krom and Horn formulae*, and of his “Habilitationsschrift,” *A simple method for determining the degree of unsolvability of decision problems for combinatorial systems*, have their root in the computational view of mathematical logic held at the time at the Institute for Logic and Foundations of Mathematics at the University of Münster, a tradition going back to (among others) Leibniz, Ackermann, Gödel, Post, Turing, Kleene, and associated in Münster with the names of the founder of the institute, Heinrich Scholz, and his followers Hans Hermes, Gisbert Hasenjäger and Dieter Rödding. This heritage determined the focus of Börger’s logical investigations in what nowadays is called computability and computational complexity theory and his early interest in applying methods from logic to solve problems in computer science.

Thus, it does not come as a surprise that from 1972 to 1976 Börger followed Edoardo Caianello’s call to help create the computer science department at the Università di Salerno (Italy), where he developed the curriculum for and taught the courses on Algorithms, Computational Complexity Theory, Semantics and Logic. After a short period (1976-1978) as Dozent of Mathematical Logic at the University of Münster, Börger became Professor for Theoretical Computer Science at the University of Dortmund (Germany), where he wrote his book on *Computability, Complexity, Logic* [1], which went through numerous editions, for over a decade became the main reference book for courses on the subject in German universities, and has been translated into English and Italian. Börger spent the academic year 1982–1983 as professor at the then new computer science department of the Università di Udine (Italy), and in 1985 accepted a computer science chair at the Università di Pisa (Italy), which he has held since then, rejecting various offers from other universities.

Through editing books and organizing workshops, summer schools, conferences, including various seminars at the Mathematical Research Institute in Oberwolfach and at Schloss Dagstuhl, Börger has been committed since the late 1970s to promoting a concrete interaction between logicians and computer

scientists, based upon his conviction that the major challenges for contemporary logic are to be found in applying logical methods in computer science. To provide an institutional basis for such an interaction, in 1986–1987 he founded together with his colleagues Michael Richter and Hans Kleine Büning the series of annual Computer Science Logic workshops. In their sixth edition, in San Miniato near Pisa, these meetings became the Annual Conference of the European Association for Computer Science Logic (<http://www.eacsl.org/>). The EACSL was founded on Börger’s initiative on July 14, 1992, by 37 computer scientists and logicians from 14 countries gathered in a Dagstuhl Seminar on Computer Science Logic Börger had organized together with his colleagues Richter, Kleine Büning, and Yuri Gurevich. From 1992 to 1997 Börger acted as first EACSL President.

Börger’s research activities in logic and complexity theory in the years 1969–1989 culminated in the book on *The Classical Decision Problem* [2], for which he wrote the first half, the one on the classification of undecidable classes of first-order logic formulae, co-authored by Erich Grädel who wrote the chapters on the complexity of the decidable classes, except for the section on the Shelah class that was written by Gurevich. The years 1986–1989 brought a shift of interest. They were characterized by a close cooperation between Börger and Gurevich on the eventual definition, by Gurevich in 1993 [3], of the notion of Abstract State Machines (ASMs)<sup>1</sup>. The idea grew out of Gurevich’s foundational concern about sharpening Turing’s thesis by a model of computation that explicitly recognizes the finiteness of computers, a theoretical effort that was crowned by success in 2000 when on the basis of three natural axioms Gurevich succeeded to prove that “Sequential Abstract State Machines capture sequential algorithms” [5].

Börger’s interest was triggered by an attempt to use ASMs to model the logic programming language Prolog. During his sabbatical from 1989 to 1990, spent at the IBM Scientific Center Heidelberg (Germany), in particular through his work in the ISO Prolog standardization committee, he recognized the potential of ASMs for building and verifying complex software-based systems in an effectively controllable manner, namely, by stepwise refinement of application-domain-focussed abstract ground models to executable code. Since then, he systematically pushed experiments to apply ASMs to real life, in particular industrial software-based systems. He triggered and led the effort of an international group of researchers which developed what is now known as the ASM method for high-level system design and analysis. He did this through multiple activities: through his own *research and publications* carried out at numerous research departments in Europe and the USA, through the *supervision of PhD students* in various European countries, through the definition and realization (including tool development) of academic and industrial *pilot projects* for building verifiable software in areas ranging from programming language implementation over train control to business processes [during sabbaticals at IBM 1989–1990,

---

<sup>1</sup> Details of the historical development can be found in the *AsmBook* [4, Ch.9].

Siemens Corporate Research and Development (Munich 1996, 1999), Microsoft Research (Redmond 2000), SAP Research (Karlsruhe, 2005)], through over 500 colloquium and conference *talks* worldwide and through the *organization* of:

- Seminars, e.g., the following Schloss Dagstuhl seminars:
  - *Methods for Semantics and Specification*, organized with Jean-Raymond Abrial (Paris), Hans Langmaack (University of Kiel, Germany), June 5–9, 1995. This seminar became known as the Steam-Boiler Seminar and resulted in a Springer LNCS State-of-the-Art Survey [6].
  - *Practical Methods for Code Documentation and Inspection*, organized with Paul Joannou (Ontario Hydro, Toronto, Canada), Dave Parnas (McMaster University, Canada), May 12–16, 1997.
  - *Requirements Capture/Documentation/Validation*, organized with Bärbel Hörger (Daimler-Benz Research, Germany), Dave Parnas (McMaster University, Canada), Dieter Rombach (Universität Kaiserslautern, Germany), June 14–18, 1999.
  - *Theory and Applications of Abstract State Machines*, organized with Andreas Blass (University of Michigan at Ann Arbor), Yuri Gurevich (Microsoft Research Redmond), March 4–8, 2002.
- Schools, e.g., the following summer schools:
  - *Informatica Matematica*, organized with Neil Jones (DIKU, University of Copenhagen), Scuola Matematica Interuniversitaria, Cortona (Italy) July 9–30, 1989.
  - *Specification and Validation Methods for Programming Languages and Systems*, organized with Alfredo Ferro (University of Catania), Lipari (Sicily), June 21–July 3, 1993. See [7].
  - *Architecture Design and Validation Methods*, organized with Ferro (University of Catania), Lipari (Sicily) June 23–July 5, 1997. See [8].
  - *Formal Methods for Engineering of Software*, organized with Furio Honsell and Simone Martine (both University of Udine), CISM, Udine (Italy), September 24–28, 2001.
  - *Software Technology*, organized with Ferro (University of Catania), Lipari (Sicily) July 1–13, 2002.
  - *Advances in Software Engineering*, organized with Ferro (University of Catania), Lipari (Sicily), July 8–21, 2007. See [9].
- Workshops, including the series of (bi-)annual international ASM workshops. This series was started in 1993 at the IFIP World Computer Congress [10, Stream C] in Hamburg (Germany). Börger proposed at this Dagstuhl seminar, whose results are reported in this volume, to merge the regular meetings of the three major state-based formal method user groups, ASMs, B, and Z. This led to the establishment of the ABZ Conferences, the first of which took place in 2008 in London (UK) [11], to be followed by the next one in 2010 in Québec (Canada).

This list shows some of the altogether 25 books and special journal issues Börger edited and of over 30 international conferences, workshops, and schools



he organized in logic (1969–1989) and computer science (since 1990). His publications comprise over 100 research papers in logic (27) and computer science (89) and over 40 papers of technical expository or of epistemological character, written in English, German, French, and Italian. His major publications on ASMs are a book on the method [4] and a book on a characteristic application of the method to Java and its JVM implementation. The latter book exhibits all the main features of the ASM method, namely, (a) *building an abstract ground model* (read: a precise definition) that can be justified to faithfully formalize the language and machine requirements in SUN’s manuals, (b) *horizontal and vertical refinements* leading from the ground model to executable code, (c) *validation* (by executing the models), and (d) *verification* (by mathematically proving or model checking properties of interest of the models, such as type safety, compiler correctness, and completeness, etc.) [12].

In recognition of his pioneering work in logic and its applications in computer science, Börger was awarded the prestigious *Humboldt Research Award* in 2007–2008.

## References

1. Börger, E.: Computability, Complexity, Logic (English translation of “Berechenbarkeit, Komplexität, Logik” from 1985. Vieweg-Verlag). Studies in Logic and the Foundations of Mathematics, vol. 128. North-Holland, Amsterdam (1989)
2. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer, Heidelberg (1997); Second printing in “Universitext”. Springer, Heidelberg (2001)
3. Gurevich, Y.: Evolving algebras 1993: Lipari Guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
4. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
5. Gurevich, Y.: Sequential Abstract State Machines capture sequential algorithms. ACM Trans. Computational Logic 1, 77–111 (2000)
6. Abrial, J.R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications. Specifying and Programming the Steam Boiler Control. LNCS, vol. 1165. Springer, Heidelberg (1996)
7. Börger, E. (ed.): Specification and Validation Methods. Oxford University Press, Oxford (1995)
8. Börger, E. (ed.): Architecture Design and Validation Methods. Springer, Heidelberg (2000)
9. Börger, E., Cisternino, A. (eds.): Advances in Software Engineering. LNCS, vol. 5316. Springer, Heidelberg (2008)
10. Pehrson, B., Simon, I.: Technology/foundations. In: IFIP 13th World Computer Congress 1994. Elsevier, Amsterdam (1994)
11. Börger, E., Bowen, J., Butler, M., Boca, P. (eds.): ABZ 2008. LNCS, vol. 5238. Springer, Heidelberg (2008)
12. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001)

# Rigorous Methods for Software Construction and Analysis

Dagstuhl Seminar 06191  
May 7–12, 2006

We survey here the key objectives and the structure of the Dagstuhl Seminar 06191, which was organized as a Festkolloquium on the occasion of Egon Börger's 60th birthday, in May 2006 in Schloss Dagstuhl, Germany.

Focusing on applied formal methods, the final seminar program covered a wide range of applied research spanning from theoretical and methodological foundations to practical applications of Abstract State Machines, B, and beyond, emphasizing universal methods and tools that, regardless of their application orientation, are still committed to the ideal of mathematical rigor.

Two overarching themes were:

- The persistent demand to foster further cross-fertilization between academic research and industrial development in the quest for innovative methods and tools to critically evaluate their potential in the light of new challenges as posed by new technological developments and paradigms in software engineering.
- The ever-present question of convergence of methods, clarifying their commonalities and differences to better understand how to combine related approaches for accomplishing the various tasks in modeling, simulation, and verification of complex hardware/software systems.

In total, 54 participants from 14 different countries and four different continents attended the seminar. In 12 sessions, comprising a total of 35 presentations, 34 technical ones and one about fellowships and awards of the Alexander von Humboldt Foundation, the following central topics, among other topics, were addressed:

- Methodological foundations of requirements specification and verification
- Characterization of specification languages and their logical foundations
- Advanced tool environments and systematic integration of tools
- Machine-assisted validation and verification
- Distributed algorithms and concurrent protocols
- Novel applications in public safety, security, and privacy
- Industrial case studies and experience reports
- The role of formal methods in computer science education

The technical talks were either 30, 45, or 60 minutes and often resulted in lively and fruitful discussions which were continued informally during the breaks. After-dinner sessions were the norm, even on Wednesday after returning from an afternoon excursion to the charming historic town of Trier.

Overall the program was fairly balanced. Roughly,

- One third of the talks were related to Abstract State Machines
- One third of the talks were related to B
- One third to other formal methods and software engineering contexts

Rather than a strict grouping of talks according to research communities, technical content, and other standard criteria, the organizers deliberately chose a mixed program with the intention to stimulate interactions across research communities and also between industry and academia. This strategy turned out to be successful, as was evident from the impressive attendance of basically all the sessions with only very few exceptions.

Over the course of the seminar, a number of spontaneous requests for additional talks were brought forward. While not all of them could be accommodated due to given schedule restrictions, such dynamics provided further evidence of the inspiring and open atmosphere that also helped forge new collaborations. Notably, there was a concrete proposal for organizing a joint working conference on ASM, B, and Z in London in 2008.

Last but not least, the tremendous hospitality of Schloss Dagstuhl made the participants feel comfortable and helped create a pleasant atmosphere that allowed everyone to fully concentrate on research contributions for more than 12 hours a day. The organizers would like to express their sincere appreciation for all the support and specifically thank the terrific Dagstuhl staff for their role in making this seminar so successful.

For the dissemination of results, revised and refereed versions of major contributions to the seminar were collected and published by Springer as an LNCS Festschrift.

October 2009

Jean-Raymond Abrial  
Uwe Glässer

## Referees

J.R. Abrial  
R. Banach  
J. P. Bowen  
M. Butler  
D. Cansell  
A. Cavarra  
A. Cisterino  
N. Evans  
R. Farahbod  
V. Gervasi  
U. Glässer  
S. Hallerstede

T.S. Hoang  
M. Leuschel  
F. Mehta  
D. Mery  
P. Müller  
W. Müller  
M.-L. Potet  
A. Prinz  
S. Rastkar  
E. Riccobene  
D. Runje  
J. N. Ruskiewicz

G. Schellhorn  
K.-D. Schewe  
S. Schneider  
C. Snook  
K. Stenzel  
B. Thalheim  
H. Treharne  
M. Vajihollahi  
L. Voisin  
Ch. Wallace  
J. Woodcock

# Table of Contents

Relaxing Restrictions on Invariant Composition in the B Method by Ownership Control <i>a la</i> SPEC# . . . . .	1
<i>Sylvain Boulmé and Marie-Laure Potet</i>	
Designing Old and New Distributed Algorithms by Replaying an Incremental Proof-Based Development . . . . .	17
<i>Dominique Cansell and Dominique Méry</i>	
Ten Reasons to Metamodel ASMs . . . . .	33
<i>Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra</i>	
An ASM-Characterization of a Class of Distributed Algorithms . . . . .	50
<i>Andreas Glausch and Wolfgang Reisig</i>	
Using Abstract State Machines for the Design of Multi-level Transaction Schedulers . . . . .	65
<i>Markus Kirchberg, Klaus-Dieter Schewe, and Jane Zhao</i>	
Validating and Animating Higher-Order Recursive Functions in B . . . . .	78
<i>Michael Leuschel, Dominique Cansell, and Michael Butler</i>	
A Systematic Verification Approach for Mondex Electronic Purses Using ASMs . . . . .	93
<i>Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius, and Wolfgang Reif</i>	
Management of UML Clusters . . . . .	111
<i>Peggy Schmidt and Bernhard Thalheim</i>	
A Step towards Merging xUML and CSP    B . . . . .	130
<i>Helen Treharne, Steve Schneider, Neil Grant, Neil Evans, and Wilson Ifill</i>	
CoreASM Plug-In Architecture . . . . .	147
<i>Roozbeh Farahbod, Vincenzo Gervasi, Uwe Glässer, and George Ma</i>	
JASMine: Accessing Java Code from CoreASM . . . . .	170
<i>Vincenzo Gervasi and Roozbeh Farahbod</i>	
A Modular Verification Methodology for C# Delegates . . . . .	187
<i>Peter Müller and Joseph N. Ruskiewicz</i>	

On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages .....	204
<i>Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack</i>	
Ten Commandments Ten Years On: Lessons for ASM, B, Z and VSR-net .....	219
<i>Jonathan P. Bowen and Michael G. Hinchey</i>	
<b>Author Index</b> .....	235

# Relaxing Restrictions on Invariant Composition in the B Method by Ownership Control *a la* SPEC#

Sylvain Boulmé and Marie-Laure Potet

Verimag, Grenoble, France  
Sylvain.Boulme@imag.fr,  
Marie-Laure.Potet@imag.fr

**Abstract.** This paper deals with modular verification of component invariants in the B Method. On the one hand, B imposes severe architecture restrictions that ensure soundness of component compositions with a few additional proof obligations. On the other hand, in the context of the verification of object oriented programs, SPEC# proposes a more expressive approach, but at the price of more complex specifications, and more numerous proof obligations. In this paper, we investigate an intermediate solution combining the advantages of both approaches.

## 1 Introduction

A project (or a development) in the B Method is constituted of several components, layered in abstract machine, refinements and implementation. A major feature of this component language is that each component can be developed and proved independently. Architecture restrictions of B ensure that composition of components is free: it only needs a few additional proofs. This is one of the keys that make the B Method scalable for large industrial applications.

In particular, at the level of invariants, this property is ensured because the invariant of a component can not be violated outside of its own operations. Hence, the users of a component  $M$  can assume the invariant of  $M$  without proving it. This invariant is established by the proofs obligations induced by the definition of  $M$ , independently of other components. In particular, when the invariant of  $M$  constrains variables of a component  $N$ , the preservation of  $M$ 's invariant is established under the *implicit assumption* that no third component calls an operation of  $N$  that modifies variables of  $N$  (which may violate the invariant of  $M$ ). This assumption is *correct* because of severe architecture restrictions.

In summary, B users can compose components without reproving their invariants, but they must deal with important architecture restrictions that ensure the soundness of reasonings involving invariants. Moreover, understanding how these restrictions ensure soundness is not trivial.

In [BP07], we have proposed a very simple (meta)model of invariant composition, inspired by SPEC# approach. This model is based on a dynamic notion

of *ownership* that characterizes which components can constrain other components through an invariant. Technically, invariant violations are monitored using a ghost variable associated to each component and called *component status*. The consistency of assumptions about invariants, with respect to the ownership dependencies, is controlled by proof obligations involving these component status. Hence, in [BP07], we show that B (without refinement) can be understood as a particular instance of this model. Moreover, this model inspired us an extension of B which authorizes more architectures and provides a better control on the initialization process of components. A little case study in [BP07] illustrates these features.

This paper shows that in our model, the proof obligations involving component status can mostly be discharged by a dedicated static analyzer. This allows us to adapt the extension proposed in [BP07] as a conservative extension of B: an extension such that current developments in B (without refinement) are still valid without new proof obligations. This paper is organized as follows. Section 2 recalls some notions of the B Method and some of its restrictions. Section 3 introduces our model as a simple adaptation of SPEC# for B. Section 4 sketches our extension of B and illustrates its expressive power on a small example. Section 5 details how this extension is made conservative. The main idea is to consider component status as a *type information* and to replace proof obligations involving these variables by *type inference*. Hence, thanks to this static analysis, annotations about component status have not to be inserted manually and are checked automatically.

## 2 A Brief Presentation of the B Method

At first, we recall some basic notions about the B Method. The core language of B specifications is based on three formalisms: data are specified using set theory, properties are first order predicates and the behavioral part is specified by *Generalized Substitutions*. Generalized Substitutions are defined by the Weakest Precondition (*WP*) semantics, introduced by E.W. Dijkstra [Dij76], and denoted here by  $[S]R$ . Hence, “ $[x := e]R$ ” is the substitution of free occurrences of  $x$  in  $R$  by  $e$  and we have:

$$[x := e \parallel y := f]R \Leftrightarrow [z := f][x := e][y := z]R$$

with  $x$ ,  $y$  and  $z$  three distinct variables and  $z$  being free in  $R$  and  $e$ . Here are some *WP* other useful definition examples:

$[P \mid S]R \Leftrightarrow P \wedge [S]R$	pre-conditioned substitution
$[S_1 ; S_2]R \Leftrightarrow [S_1][S_2]R$	sequential substitution
$[S_1 \parallel S_2]R \Leftrightarrow ([S_1]R) \wedge ([S_2]R)$	bounded choice substitution

PRE  $P$  THEN  $S$  END is the syntactic form for  $P \mid S$ .

Generalized substitutions are equivalently characterized by two predicates,  $\text{trm}(S)$  and  $\text{prd}(S)$ , that respectively express the required condition for substitution  $S$  to terminate, and the relation between before and after states (denoted



respectively by  $v$  and  $v'$ ). Weakest precondition calculus includes termination: we have  $[S]R \Rightarrow \text{trm}(S)$ , for any  $R$ .

**Definition 1 (trm and prd predicates).**

$$\text{trm}(S) \Leftrightarrow [S]\text{true} \qquad \text{prd}(S) \Leftrightarrow \neg[S]\neg(v' = v)$$

## 2.1 Abstract Machine

As proposed by C. Morgan [MG90], B components correspond to the standard notion of state machines which define an initial state and a set of operations, acting on internal state variables. Moreover, an invariant is attached to an abstract machine: this is a property which must hold in observable states, i.e. states before and after operations calls. Roughly, an abstract machine has the following shape.

```

MACHINE  $M$ 
VARIABLES  $v$ 
INVARIANT  $I$ 
INITIALIZATION  $U$ 
OPERATIONS
   $o \leftarrow \text{nom\_op}(i) \hat{=}$ 
    PRE  $P$  THEN  $S$  END ;
  ...
END

```

$M$  is the component name,  $v$  a list of variable names,  $I$  a property on variables  $v$ , and  $U$  a generalized substitution. In the operation definition,  $i$  (resp.  $o$ ) denotes the list of input (resp. output) parameters,  $P$  is the precondition on  $v$  and  $i$ , and  $S$  is a generalized substitution which describes how  $v$  and  $o$  are updated.

Proof obligations attached to machine  $M$  consist in showing that  $I$  is an inductive property of component  $M$ :

**Definition 2 (Invariant proof obligations).**

$$\begin{array}{ll} (1) & [U]I \qquad \text{initialization} \\ (2) & I \wedge P \Rightarrow [S]I \qquad \text{operations} \end{array}$$

## 2.2 Invariants Composition

Abstract machines are combined through a few primitives. Here, we consider only clause INCLUDES, such that if  $M$  INCLUDES  $N$ , then  $M$  can be defined using any  $N$  operations and  $M$  invariant can constrain  $N$  variables. See [BP07] for considerations about clause SEES.

The first feature underlying invariant composition in the B Method is invariant preservation by encapsulated substitutions. A substitution  $S$  is an encapsulated substitution relative to a given component  $M$  if and only if variables of  $M$  are not directly assigned in  $S$ , but only through calls to  $M$  operations. Thus, any encapsulated substitution relative to  $M$  preserves, by construction, invariant of  $M$  (see [Pot02]). The second feature underlying invariant composition is a set of restrictions on component shared variables in inconsistent ways. Let us illustrate these restrictions on the following example. In this simplified example, we use abusively a sequence at the level of abstract machines, whereas such a sequence could only happen in a refinement:

MACHINE <i>Resource</i> VARIABLES $x$ INVARIANT $x \in \mathbb{N}$ INITIALIZATION $x := 0$ OPERATIONS $incr \hat{=} x := x + 1$ END	MACHINE <i>User</i> INCLUDES <i>Resource</i> INVARIANT $even(x)$ OPERATIONS $incr2 \hat{=}$ $BEGIN\ incr ; incr\ END$ END
---	---

Here, the following rule applies: a component  $M$  can call operations of a component  $N$  that modifies the state of  $N$  only if  $M$  includes  $N$  transitively. Moreover, syntactic restrictions ensure that INCLUDES dependency relation is a tree: it contains no cycle, and each machine can be included only once. These restrictions ensure that a component containing one of these sequences is syntactically incorrect:

(1)	$init ; incr2 ; incr ; incr2 ;$
(2)	$init ; incr2 ; incr ; incr ; incr2 ;$

Indeed, these sequences are rejected by  $B$  because they combine modifying operations of *Resource* and *User*. The rejection of sequence 1 prevents the second call of operation *incr2* to occur in a state where invariant of *User* is broken. But, sequence 2 is rejected although each operation-call happens in a state where the invariant attached to each operation definition is valid. In practical experiments, these restrictions make the design of architectures difficult [BB99, Hab01]. Actually, components can share variables only in a very limited way: at most one writer with several readers.

### 3 Adapting SPEC# Approach for Modules *a la B*

The SPEC# approach [BDF<sup>+</sup>04, LM04] proposes a flexible methodology for modular verification of objects invariants, which is based on a dynamic notion of ownership. An ownership relation describes which objects can constrain other objects, i.e. which object has an invariant depending on the value of another object. It is imposed that this relation is dynamically a forest (e.g. a set of disjoint trees). This permits the generation of proof obligations ensuring that an object is not modified while it is constrained by the invariant of another object. Dynamic ownership of an object can be transferred during execution, introducing some flexibility with respect to  $B$  restrictions. We directly present our adaptation of the SPEC# approach in the framework of  $B$  components and generalized substitutions style.

#### 3.1 Dependencies between Components and Admissible Invariants

In this section, we assume that architectures of components are structured by a binary relation between components, noted `owns` and called *static ownership*. This relation is assumed to have no cycle (its transitive closure `owns+` is irreflexive). It is related to the notion of admissible invariants (invariants that are

syntactically correct): all the free variables appearing in an admissible invariant must be declared in a component of  $\text{owns}^*({M})$  (the image of  $M$  through the reflexive and transitive closure of  $\text{owns}$ ). In other words, if  $M.\text{Inv}$  denotes the invariant of component  $M$  and  $M.\text{Var}$  the variables declared in  $M$ , then for all syntactically correct  $M$ , we assume that  $\text{free}(M.\text{Inv}) \subseteq \bigcup_{(M,N) \in \text{owns}^*} N.\text{Var}$ .

We use  $\text{owns}$  to give a hierarchical structure to validity of invariants: if the invariant of a component  $M$  can be *safely assumed*, then all the invariants components in  $\text{owns}^*({M})$  can also be *safely assumed*. Concretely, each component  $N$  is implicitly extended by a ghost variable  $N.\text{st}$ , belonging to  $\{\text{invalid}, \text{valid}, \text{committed}\}$ . Hence,  $N.\text{Inv}$  can be *safely assumed* if and only if  $N.\text{st} \neq \text{invalid}$ .

A component  $M$  is a *dynamic owner* of a component  $N$  if and only if  $(M, N) \in \text{owns}$  and  $M.\text{st} \neq \text{invalid}$ . The semantics of this ghost variable is the following:

- if  $N.\text{st} = \text{invalid}$ , then its invariant may be violated. Moreover,  $N$  has no dynamic owner. Hence, any modification on  $N$  variables is authorized.
- if  $N.\text{st} = \text{valid}$ , then its invariant is established, and it has no dynamic owner.
- if  $N.\text{st} = \text{committed}$ , then its invariant is established, and it has a single dynamic owner.

This semantics is formally expressed by the following definition.

**Definition 3 (Status meta-invariants).** *All components  $M$  and  $N$  have to verify the following meta-invariants:*

$\mathcal{MI}_1$	$M.\text{st} \neq \text{invalid} \Rightarrow M.\text{Inv}$
$\mathcal{MI}_2$	$M.\text{st} \neq \text{invalid} \wedge (M, N) \in \text{owns} \Rightarrow N.\text{st} = \text{committed}$
$\mathcal{MI}_3$	$M.\text{st} \neq \text{invalid} \wedge \text{owns}(\{M\}) \cap \text{owns}(\{N\}) \neq \emptyset \wedge M \neq N \Rightarrow N.\text{st} = \text{invalid}$

Let  $\mathcal{MI}$  be  $\mathcal{MI}_1 \wedge \mathcal{MI}_2 \wedge \mathcal{MI}_3$ .

The first meta-invariant states that a component invariant can be safely assumed if its status is different from  $\text{invalid}$ . Meta-invariant  $\mathcal{MI}_2$  imposes that when a component invariant is not  $\text{invalid}$  then components transitively owned by this component have to be declared as  $\text{committed}$ . Finally  $\mathcal{MI}_3$  ensures that each component has at most one unique dynamic owner.

### 3.2 Extending the Language of Generalized Substitution

**Assignment substitution.** In our language, assignment substitution is preconditioned, but it can occur outside of the component where the assigned variable is bound (there is *a priori* no variable encapsulation). We have:

subst	trm	prd
$N.\text{Var} := e$	$N.\text{st} = \text{invalid}$	$N.\text{st}' = N.\text{st} \wedge N.\text{Var}' = e$

Meta-invariants of definition 3 are preserved by this substitution, see [BP07].

**Substitutions  $\text{pack}(M)$  and  $\text{unpack}(M)$ .** Substitutions are extended with two new commands  $\text{pack}(M)$  and  $\text{unpack}(M)$ . The former requires the establishment of  $M$  invariant and the latter allows violation of  $M$  invariant. Status variables can only be modified via these commands. They can be invoked in  $M$  or outside of  $M$ . They are formally defined by:

subst	trm	prd
$\text{pack}(M)$	$\forall N.((M,N) \in \text{owns} \Rightarrow N.\text{st} = \text{valid})$ $\wedge M.\text{st} = \text{invalid}$ $\wedge M.\text{Inv}$	$\forall N.((M,N) \in \text{owns} \Rightarrow N.\text{st}' = \text{committed})$ $\forall N.(M \neq N \wedge (M,N) \notin \text{owns} \Rightarrow N.\text{st}' = N.\text{st})$ $\wedge M.\text{st}' = \text{valid}$ $\wedge M.\text{Var}' = M.\text{Var}$
$\text{unpack}(M)$	$M.\text{st} = \text{valid}$	$\forall N.((M,N) \in \text{owns} \Rightarrow N.\text{st}' = \text{valid})$ $\forall N.(M \neq N \wedge (M,N) \notin \text{owns} \Rightarrow N.\text{st}' = N.\text{st})$ $\wedge M.\text{st}' = \text{invalid}$ $\wedge M.\text{Var}' = M.\text{Var}$

Control of dynamic ownership appears in these commands. The precondition of  $\text{pack}(M)$  imposes that the components statically owned by  $M$  have no dynamic owners, then  $\text{pack}(M)$  makes  $M$  the dynamic owner of all the components that it statically owns. Of course,  $\text{unpack}(M)$  has the reverse effect. Here, let us note that the precondition of  $\text{unpack}(M)$  imposes that  $M$  has itself no dynamic owner. In other words, if we want to unpack a component  $N$ , for instance in order to modify it through a preconditioned assignment, we are first obliged to unpack its dynamic owner (and so on recursively). It is easy to prove that meta-invariants  $\mathcal{MI}_1$ ,  $\mathcal{MI}_2$  and  $\mathcal{MI}_3$  are preserved by  $\text{pack}$  and  $\text{unpack}$  substitutions.

Finally for each substitution  $S$  built from preconditioned assignment,  $\text{pack}$  and  $\text{unpack}$  new commands and other standard B substitutions (except assignment substitution) we have:

**Proposition 1 (Meta-invariants preservation).**  $\mathcal{MI} \wedge \text{trm}(S) \Rightarrow [S]\mathcal{MI}$

### 3.3 Revisiting Example 2.2

In example 2.2 the couple  $(User, Resource)$  must belong to the ownership relation  $\text{owns}$  because variables of the component  $Resource$  are constrained by the invariant part of the component  $User$ . Operations  $\text{incr}$  and  $\text{incr2}$  are now described in the following way:

$$\begin{aligned} \text{incr} &\hat{=} \text{PRE } Resource.\text{st} = \text{valid} \\ &\quad \text{THEN } \text{unpack}(Resource) ; x := x + 1 ; \text{pack}(Resource) \text{ END} \\ \text{incr2} &\hat{=} \text{PRE } User.\text{st} = \text{valid} \wedge Resource.\text{st} = \text{committed} \\ &\quad \text{THEN } \text{unpack}(User) ; \text{incr} ; \text{incr} ; \text{pack}(User) \text{ END} \end{aligned}$$

Starting in a state such that  $User.\text{st} = \text{valid}$  and  $Resource.\text{st} = \text{committed}$ , we can now label the sequence 2 of section 2.2 with  $\text{pack}$  and  $\text{unpack}$  substitutions such that termination of this sequence is provable:

$$\text{incr2} ; \text{unpack}(User) ; \text{incr} ; \text{incr} ; \text{pack}(User) ; \text{incr2}$$

## 4 A Component Approach

In a component approach, proofs are established at the level of operation definitions. All operations `PRE P THEN S END`, where status variables can occur free in  $P$  and where  $S$  can perform `unpack` and `pack` must satisfy:

$$MI \wedge P \Rightarrow \text{trm}(S)$$

In other words, preservation of components invariants must be expressed explicitly through `unpack` and `pack` command.

### 4.1 Several Forms for Definitions of Operations

In [BP07] we have given a systematic encoding of B operations and architecture rules in the Spec# approach. For instance, an *interface operation of a component*  $M$  is an operation which preserves the invariant of  $M$ . For instance, in B, the `owns` relation is assimilated to the `INCLUDES` clause. If we do not take into account clauses `SEES`, interface operations are of the form :

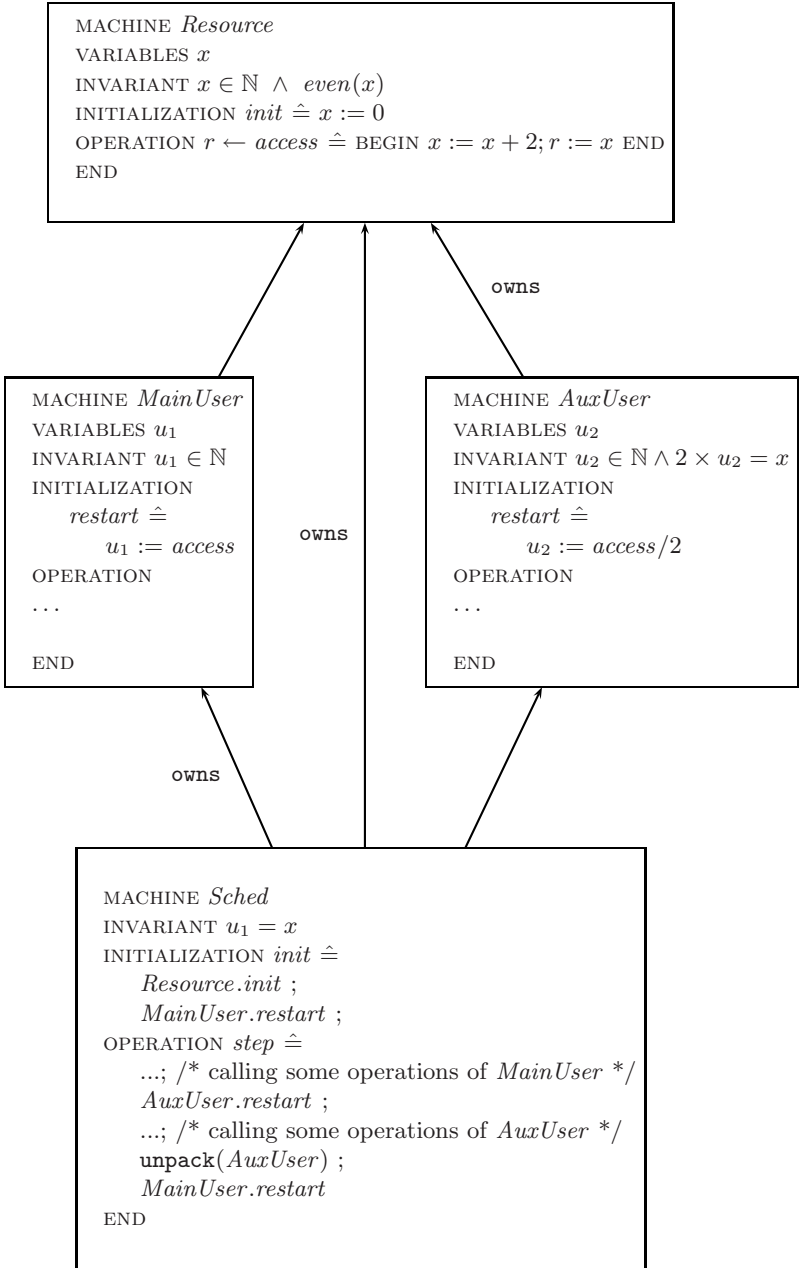
modifying case	PRE $P \wedge M.\text{st} = \text{valid}$ THEN <code>unpack(M) ; S ; pack(M)</code> END
read-only case	PRE $P \wedge M.\text{st} \neq \text{invalid}$ THEN $S$ END

### 4.2 Initialization and Reinitialization Process

In the B Method, all local initializations are implicitly combined to build a global initialization step. Because more sharing is now admitted, we make explicit the initialization process in using operations which establish invariants (on the contrary to interface operations which preserve invariant). At last, initializations can be invoked in any place, in order to reinitialize variables. Let  $U$  be the initialization substitution of component  $M$ . Several form of initialization operations are possible:

case 1	PRE $M.\text{st} = \text{invalid}$ THEN $U ; \text{pack}(M)$ END
case 2	PRE $M.\text{st} = \text{invalid} \wedge N.\text{st} \neq \text{invalid}$ THEN $U ; \text{pack}(M)$ END
case 3	PRE $M.\text{st} = \text{invalid} \wedge N.\text{st} = \text{invalid}$ THEN $N.\text{init} ; U ; \text{pack}(M)$ END

Case 1 corresponds to initialization of a stand-alone machine. Case 2 corresponds to an initialization depending on another initialization, like when  $M$  sees  $N$  in B. Case 3 corresponds to an initialization performing another initialization,  $N.\text{init}$ . Case 3 implicitly happens in B when  $M$  includes  $N$ : in particular, the elaboration of  $M$  invariant involves initial values of  $N$ .



**Fig. 1.** An example with two writers

### 4.3 An Example with Two Writers

We now study a four component architecture (see figure 4): a component *Resource* is shared between two components *MainUser* and *AuxUser* which are both used in *Sched*. Moreover, component *AuxUser* constrains *Resource* in its invariant whereas *MainUser* does not. Actually, *Sched* links in its invariant variables of *MainUser* to variables of *Resource*. This architecture can not be expressed in **B** because *MainUser* and *AuxUser* modify concurrently the variable of *Resource*. Let us now explain the differences between simple arrows and arrows labeled by **owns**. Simple arrows correspond here to pairs  $(M, N)$  such that  $M$  can call operations of  $N$ , read or write variables of  $N$  but does not constrain  $N$  in its invariant. Here, if simple arrows were replaced by **owns**-arrows, then *MainUser* and *AuxUser* could not be valid in the same time (because *Resource* can only have a single dynamic owner) and thus, *Sched* could never be packed.

In figure 4, initializations of component  $M$  perform implicitly a  $\text{pack}(M)$ , and operations of  $M$  are implicitly bracketed by substitutions  $\text{unpack}(M)$  and  $\text{pack}(M)$ . Moreover, preconditions on status have been omitted. For instance, the operation *Aux.restart* should be:

```
PRE  Resource.st = valid  $\wedge$  AuxUser.st = invalid
THEN u2 := access/2 ; pack(AuxUser) END
```

Finally, let us remark that the command  $\text{unpack}(AuxUser)$  is needed in *Sched.step* because, after *AuxUser.restart*, component *Resource* is committed. If the **unpack** substitution was omitted, the proof obligations relative to termination of the operation *step* would be false.

## 5 Static Analysis and Proof Obligations

The approach described in section 4 presents some important drawbacks. First, proof obligations are polluted with conditions on component status, which are mostly trivial. More dramatically, the user has to provide the preconditions on component status: this task is tedious and error prone. Indeed, the user can give a precondition  $P$  on component status such that  $P \wedge \mathcal{MI}_2 \wedge \mathcal{MI}_3$  is not satisfiable. Such a bad configuration can be discovered only at the end of the proof process, while attempting to prove the precondition of an operation call, whereas it may be the consequence of a bad architectural design.

To circumvent these drawbacks, we propose to check the consistency of component status using a static analysis, instead of proof obligations. Moreover, this static analysis infers preconditions over component status in operations and rejects operations which are suspected to have an unsound precondition. We benefit here from the fact that, in **B**, on the contrary to **SPEC#**, there is no alias on component names. The next sections introduce our status static analysis and how it is used in practice.

### 5.1 Introduction to Our Static Analysis

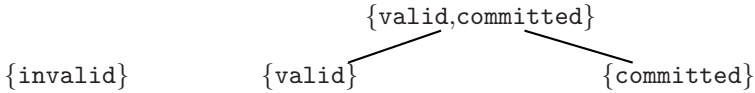
Actually, the design of such a static analysis needs to restrict the expressive power of our language. Here, we require that if a substitution is a branch (a

choice, an if-then-else, a while-loop), then **all branches perform the same observational transformation on the status variables**. For instance, if a branch is skip, others branches can only perform well-bracketed `unpack/pack` or `pack/unpack`. This is the main abstraction of our analysis.

Concretely, each substitution will be abstracted to a *status type*  $\mathbb{P} ! \mathbb{A}$  where  $\mathbb{P}$  is a conjunction of atomic status clauses about `st` variables and  $\mathbb{A}$  is a multiple affectation about `st` variables.

**Definition 4 (Syntax of status types).** *A status type is noted  $\mathbb{P} ! \mathbb{A}$  where:*

- $\mathbb{A}$  is called a *status affectation*, and is either `skip` or  $M.\text{st} := v \parallel \mathbb{A}_1$ , where component name  $M$  does not appear in  $\mathbb{A}_1$ , and  $v$  is `invalid`, `valid` or `committed`.
- $\mathbb{P}$  is called a *status precondition*, and is either `True` or  $M.\text{st} \in St \wedge \mathbb{P}_1$ , where component name  $M$  does not appear in  $\mathbb{P}_1$  and  $St$  belongs to the following partially ordered set:



The syntax chosen here suggests that status types  $\mathbb{P} ! \mathbb{A}$  can be interpreted itself as a preconditioned substitution  $\mathbb{P} \mid \mathbb{A}$ , working on variable `st` of components. We use this interpretation to express the equality between status types, and more generally the correctness of our type system. Indeed, equality between status types is defined from the equality between substitutions [Abr96], under the hypothesis of the meta-invariants  $\mathcal{MI}_2$  and  $\mathcal{MI}_3$ .

**Definition 5 (equality between status types under  $\mathcal{MI}_2 \wedge \mathcal{MI}_3$ ).**

$$\begin{array}{l}
 \mathbb{P}_1 ! \mathbb{A}_1 = \mathbb{P}_2 ! \mathbb{A}_2 \quad \text{if and only if} \\
 (\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P}_1) \mid \mathbb{A}_1 = (\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P}_2) \mid \mathbb{A}_2
 \end{array}$$

Due to the particular syntax of status preconditions, these propositions hold:

**Proposition 2.** *For a given architecture and two status preconditions  $\mathbb{P}_1$  and  $\mathbb{P}_2$ , it is decidable to check whether  $(\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P}_1) \Rightarrow \mathbb{P}_2$  is a tautology.*

**Proposition 3 (decidability of status precondition soundness).** *For a given architecture and a given status precondition  $\mathbb{P}$ , it is decidable to check whether  $\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P}$  is satisfiable.*

Then, the equality between status affectations is established using a normalized form, through the following function *Simplify*:

$$\begin{array}{l}
 \text{Simplify}(\text{skip}, \mathbb{P}) = \text{skip} \\
 \text{Simplify}(M.\text{st} := v \parallel \mathbb{A}, \mathbb{P}) = \text{Simplify}(\mathbb{A}, \mathbb{P}) \\
 \quad \text{if } \mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P} \Rightarrow M.\text{st} \in \{v\} \\
 \text{Simplify}(M.\text{st} := v \parallel \mathbb{A}, \mathbb{P}) = M.\text{st} := v \parallel \text{Simplify}(\mathbb{A}, \mathbb{P}) \\
 \quad \text{otherwise}
 \end{array}$$

Hence, we deduce from prop [2](#) and from the following proposition that equality between status type is decidable:



**Proposition 4 (equality between status types under  $\mathcal{MT}_2 \wedge \mathcal{MT}_3$ ).**

$\mathbb{P}_1 ! \mathbb{A}_1 = \mathbb{P}_2 ! \mathbb{A}_2$  if and only if

- $\mathcal{MT}_2 \wedge \mathcal{MT}_3 \wedge \mathbb{P}_1 \Rightarrow \mathbb{P}_2$  and  $\mathcal{MT}_2 \wedge \mathcal{MT}_3 \wedge \mathbb{P}_2 \Rightarrow \mathbb{P}_1$
- and,  $\text{Simplify}(\mathbb{A}_1, \mathbb{P}_1) = \text{Simplify}(\mathbb{A}_2, \mathbb{P}_2)$  modulo associativity and commutativity and  $\parallel$ .

Now, we illustrate the expected behavior of our typing rules on a few examples.

*Example 1 (Basic typing of sequences).* Assuming 3 components  $A, B, C$  such that  $A$  owns  $B$  and  $B$  owns  $C$ , then :

- substitution  $\text{unpack}(A) ; B.x := e$  must be rejected. Indeed, status precondition of  $B.x := e$  is  $B.\text{st} \in \{\text{invalid}\}$  whereas status affectation of  $\text{unpack}(A)$  is  $A.\text{st} := \text{invalid} \parallel B.\text{st} := \text{valid}$ .
- substitution  $\text{unpack}(A) ; \text{unpack}(B)$  must be abstracted to:  
 $A.\text{st} \in \{\text{valid}\} ! (A.\text{st} := \text{invalid} \parallel B.\text{st} := \text{invalid} \parallel C.\text{st} := \text{valid})$

*Example 2 (Typing approximations in branches).* Assuming a component  $A$  owning nothing, then :

- substitution  $\text{skip} \parallel (\text{unpack}(A) ; \text{pack}(A))$  must be abstracted to:  
 $A.\text{st} \in \{\text{valid}\} ! \text{skip}$
- substitution  $\text{skip} \parallel \text{unpack}(A)$  is rejected because the two branches perform incompatible observational effects on status.

## 5.2 Inference Rules of Our Static Analysis

As component status is now a typing information, we need to restrict a bit the language of substitutions. Substitutions can contain  $\text{pack}$  and  $\text{unpack}$  occurrences but formulae in preconditions or in guards can not reference component status. However, in order to allow the user to express preconditions over component status, we introduce a new substitution, called *status precondition* substitution and written  $\mathbb{P} ! S$  where  $\mathbb{P}$  is a precondition status and  $S$  a substitution. For the weakest-precondition calculus,  $\mathbb{P} ! S$  is interpreted as  $\mathbb{P} \mid S$ .

We now define the relation  $S \vdash \mathbb{P} ! \mathbb{A}$ , meaning that  $S$  is abstracted to  $\mathbb{P} ! \mathbb{A}$  by our static analyzer.

### Rules AFFECT, UNPACK and PACK

$$\begin{array}{c}
 \frac{}{M.x := e \vdash M.\text{st} \in \{\text{invalid}\} ! \text{skip}} \text{AFFECT} \\
 \\
 \frac{\mathbb{P} \equiv M.\text{st} \in \{\text{valid}\} \quad \mathbb{A} \equiv M.\text{st} := \text{invalid} \parallel (\parallel_{N \in \text{owns}(\{M\})} N.\text{st} := \text{valid})}{\text{unpack}(M) \vdash \mathbb{P} ! \mathbb{A}} \text{UNPACK} \\
 \\
 \frac{\mathbb{P} \equiv M.\text{st} \in \{\text{invalid}\} \wedge (\bigwedge_{N \in \text{owns}(\{M\})} N.\text{st} \in \{\text{valid}\}) \quad \mathbb{A} \equiv M.\text{st} := \text{valid} \parallel (\parallel_{N \in \text{owns}(\{M\})} N.\text{st} := \text{committed})}{\text{pack}(M) \vdash \mathbb{P} ! \mathbb{A}} \text{PACK}
 \end{array}$$

**Rule SEQ.** The sequence rule checks that status types can be sequentialized. When this is not the case, this rule is not applicable.

$$\frac{S_1 \vdash \mathbb{P}_1 ! \mathbb{A}_1 \quad S_2 \vdash \mathbb{P}_2 ! \mathbb{A}_2 \quad [\mathbb{A}_1] \mathbb{P}_2 \rightsquigarrow \mathbb{P}_3 \quad \mathbb{P}_1 \sqcap \mathbb{P}_3 \rightsquigarrow \mathbb{P} \quad \mathbb{A}_1 ; \mathbb{A}_2 \rightsquigarrow \mathbb{A}}{S_1 ; S_2 \vdash \mathbb{P} ! \text{Simplify}(\mathbb{A}, \mathbb{P})} \text{SEQ}$$

Notation  $[\mathbb{A}] \mathbb{P}_1 \rightsquigarrow \mathbb{P}_2$  means that there exists  $\mathbb{P}_2$  such that  $\mathbb{P}_2 \Leftrightarrow [\mathbb{A}] \mathbb{P}_1$ . The result of  $[\mathbb{A}] \mathbb{P}_1$  is undefined if a status variable  $M.st$  is set by a value which is not compatible with  $\mathbb{P}_1$ .  $FV(\mathbb{P})$  denotes the set of components names appearing in  $\mathbb{P}$ .

$$\frac{\frac{\text{[skip]} \mathbb{P} \rightsquigarrow \mathbb{P}}{v \in St} \quad \frac{[\mathbb{A}] \mathbb{P}_1 \rightsquigarrow \mathbb{P}_2 \quad M \notin FV(\mathbb{P}_1)}{[M.st := v \parallel \mathbb{A}] \mathbb{P}_1 \rightsquigarrow \mathbb{P}_2}}{[M.st := v \parallel \mathbb{A}] (M.st \in St \wedge \mathbb{P}_1) \rightsquigarrow \mathbb{P}_2}$$

Notation  $\mathbb{P}_1 \sqcap \mathbb{P}_2 \rightsquigarrow \mathbb{P}_3$  means that there exists  $\mathbb{P}_3$  such that  $\mathbb{P}_3 \Leftrightarrow \mathbb{P}_1 \wedge \mathbb{P}_2$ . The result of  $\mathbb{P}_1 \sqcap \mathbb{P}_2$  is undefined if  $\mathbb{P}_1$  and  $\mathbb{P}_2$  constrain some status variables in a contradictory way.

$$\frac{\frac{True \sqcap \mathbb{P} \rightsquigarrow \mathbb{P}}{St_1 \sqcap St_2 \rightsquigarrow St_3} \quad \frac{\mathbb{P}_1 \sqcap \mathbb{P}_2 \rightsquigarrow \mathbb{P}_3 \quad M \notin FV(\mathbb{P}_2)}{(M.st \in St \wedge \mathbb{P}_1) \sqcap \mathbb{P}_2 \rightsquigarrow M.st \in St \wedge \mathbb{P}_3}}{(M.st \in St_1 \wedge \mathbb{P}_1) \sqcap (M.st \in St_2 \wedge \mathbb{P}_2) \rightsquigarrow (M.st \in St_3) \wedge \mathbb{P}_3}$$

Finally, the sequence of two status affectations, noted  $\mathbb{A}_1 ; \mathbb{A}_2$  is defined below:

$$\frac{\frac{\text{skip} ; \mathbb{A} \rightsquigarrow \mathbb{A}}{\mathbb{A}_1 ; \mathbb{A}_2 \rightsquigarrow \mathbb{A}_3} \quad \frac{M \notin FV(\mathbb{A}_2)}{(M.st := v \parallel \mathbb{A}_1) ; \mathbb{A}_2 \rightsquigarrow M.st := v \parallel \mathbb{A}_3}}{(M.st := v_1 \parallel \mathbb{A}_1) ; (M.st := v_2 \parallel \mathbb{A}_2) \rightsquigarrow M.st := v_2 \parallel \mathbb{A}_3}$$

**Rule CHOICE.** As explained previously, for the choice both branches are required to produce the same observational effects under the hypothesis of the current precondition (and  $\mathcal{MI}_2 \wedge \mathcal{MI}_3$ ).

$$\frac{S_1 \vdash \mathbb{P}_1 ! \mathbb{A}_1 \quad S_2 \vdash \mathbb{P}_2 ! \mathbb{A}_2 \quad \mathbb{P}_1 \sqcap \mathbb{P}_2 \rightsquigarrow \mathbb{P} \quad \mathbb{P} ! \mathbb{A}_1 = \mathbb{P} ! \mathbb{A}_2}{S_1 \parallel S_2 \vdash \mathbb{P} ! \text{Simplify}(\mathbb{A}_1, \mathbb{P})} \text{CHOICE}$$

**Rules PRE, GUARD and ANY.** Furthermore, preconditions and guards, which are not allowed to express properties about component status anymore, are just skipped by the static analysis.

$$\frac{S \vdash \mathbb{P} ! \mathbb{A}}{Q \mid S \vdash \mathbb{P} ! \mathbb{A}} \text{PRE} \quad \frac{S \vdash \mathbb{P} ! \mathbb{A}}{Q \Longrightarrow S \vdash \mathbb{P} ! \mathbb{A}} \text{GUARD} \quad \frac{S \vdash \mathbb{P} ! \mathbb{A}}{@z \cdot S \vdash \mathbb{P} ! \mathbb{A}} \text{ANY}$$

**Rule ST\_PRE** For status precondition substitution, we simply consider the conjunction of the inferred precondition and the precondition given by the user. If the preconditions are not compatible then the whole substitution is rejected.

$$\frac{S \vdash \mathbb{P}_2 ! \mathbb{A} \quad \mathbb{P}_1 \sqcap \mathbb{P}_2 \rightsquigarrow \mathbb{P}}{\mathbb{P}_1 ! S \vdash \mathbb{P} ! \text{Simplify}(\mathbb{A}, \mathbb{P})} \text{ST\_PRE}$$

### 5.3 Correctness of the Typing Rules

Our static analysis aims to discard from proof obligations the reasoning about the status of components. Hence, the previous weakest-precondition calculus is now split in two parts: one part is our static analysis, and the other part is a new weakest-precondition calculus, written  $[S]^\dagger R$ , which does not perform substitution of `st` variables. The definition of  $[S]^\dagger R$  is similar to  $[S]R$  except for:

$$\begin{aligned} [\text{unpack}(M)]^\dagger R &= R \\ [\text{pack}(M)]^\dagger R &= M.\text{Inv} \wedge R \\ [\mathbb{P} ! S]^\dagger R &= R \end{aligned}$$

The correctness of the typing rules expresses that the cooperation of the type system and the new weakest-precondition calculus replace the previous weakest-precondition calculus.

**Proposition 5 (correctness of the typing rules).** *If  $S \vdash \mathbb{P} ! \mathbb{A}$ , then:*

$$\mathbb{P} \wedge \mathcal{MI} \wedge [S]^\dagger[\mathbb{A}]R \Rightarrow [S]R$$

### 5.4 The Typing Algorithm and Its Application

In conclusion, let us detail how our methodology is improved by using the static analysis. For an operation of the form  $P \mid S$ ,

1. we first infer the status type of  $S$ . The typing algorithm proceeds by induction over the syntactic structure of  $S$ . If the inference fails, then the operation is rejected because it may be inconsistent about component status.
2. then, we check that  $\mathbb{P}$  is sound for the considered architecture (according to proposition 3). If this is not the case, then the operation is also rejected.
3. at last, section 4 indicates that we must prove:  $(\mathcal{MI} \wedge P \wedge \mathbb{P}) \Rightarrow \text{trm}(S)$ . We recall here that invariant proof obligations are now embedded in substitutions through `pack` substitutions. Let  $\text{ISat}(\mathbb{P})$  be defined by:

$$\text{ISat}(\mathbb{P}) = \bigwedge_{M / (\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge \mathbb{P}) \Rightarrow M.\text{st} \in \{\text{valid}, \text{committed}\}} M.\text{Inv}$$

This formula on component variables is the conjunction of all invariants that are deduced from  $\mathcal{MI} \wedge \mathbb{P}$ . It is computable by proposition 2. Using proposition 5, the previous proof obligation is a consequence of the following one:

$$(P \wedge \text{ISat}(\mathbb{P})) \Rightarrow [S]^\dagger \text{True}$$

Hence, this discharges the user to reason about component status.

### 5.5 Revisiting Example of Section 4.3

We use our typing rules to check that example of figure 1 is accepted (as explained section 4.1, we consider that initializations are implicitly ended by `pack`,

and that interface operations are implicitly bracketed by `unpack/pack`). Moreover, termination proof obligations can also be discharged as expected. For instance, given the operations:

$$\begin{aligned} r \leftarrow \text{Resource.access} &\hat{=} \\ &\text{BEGIN } \text{unpack}(\text{Resource}) ; x := x + 2 ; r := x ; \text{pack}(\text{Resource}) ; \text{END} \\ \text{AuxUser.restart} &\hat{=} \\ &\text{BEGIN } u_2 := \text{access}/2 ; \text{pack}(\text{AuxUser}) \text{ END} \end{aligned}$$

Using structural rules, the derived status types are:

- $\text{Resource.access} \vdash \text{Resource.st} \in \{\text{valid}\} ! \text{skip}$
- $\text{AuxUser.restart}$
- $\vdash \text{Resource.st} \in \{\text{valid}\} \wedge \text{AuxUser.st} \in \{\text{invalid}\}$
- $! \text{AuxUser.st} := \text{valid} \parallel \text{Resource.st} := \{\text{committed}\}$

Finally, the proof of termination for  $\text{AuxUser.restart}$  imposes to prove

$$[x := x + 2 ; u_2 := x/2]^\dagger (u_2 \in \mathbb{N} \wedge 2 \times u_2 = x)$$

under the precondition  $\text{Resource.st} \in \{\text{valid}\} \wedge \text{AuxUser.st} \in \{\text{invalid}\}$ .

Using the definition of  $[S]^\dagger$  and the ISat operation we have to prove:

$$\text{even}(x) \Rightarrow ((x + 2)/2 \in \mathbb{N} \wedge 2 \times ((x + 2)/2) = x + 2)$$

This is true, using the fact that  $/$  is euclidean division.

## 6 Conclusion and Perspectives

In conclusion, this paper proposes an extension of  $\mathbf{B}$  (without refinement) which authorizes more architectures and provides a better control on the initialization process of components. In  $\mathbf{B}$ , given two components  $M$  and  $N$  such that  $M$  uses  $N$ , all operations of  $M$  share the same constraint on the use of  $N$ . With our extension, each operation of  $M$  may have its own constraint on  $N$ .

Our extension is conservative in the sense that proof obligations generated for current developments in  $\mathbf{B}$  (without refinement) correspond to current proof obligations. It is inspired by the  $\text{SPEC}\#$  methodology, but adapted for  $\mathbf{B}$  and uses a dedicated static analysis that discharges completely the user to reason about component status. Of course, our methodology does not eliminate architectural unsoundness. As in current  $\mathbf{B}$ , when the user is faced to such an unsoundness, he or she must correct it. Here, corrections may consist in adding `unpack` commands, or rewriting invariants, or reconsidering some part of the architecture.

However, this extension will present a practical interest only when it will support refinement. Here,  $\text{SPEC}\#$  may still inspire us: the treatment of (behavioral) subtyping may help for algorithmic refinement and the treatment of model fields may help for data refinement [LM06].

As discussed in the conclusion of [BP07], our extension of  $\mathbf{B}$  allows multiple writers. However, reasonings are preserved by refinement only when all ownership transfers between successive writers are performed via a reinitialization operation (as in example of section 4.3). Hence, this approach seems complementary with a rely-guarantee approach (see [BN04, NB04]).

## References

- [Abr96] Abrial, J.R.: *The B-Book*. Cambridge University Press, Cambridge (1996)
- [Ba99] Behm, P., et al.: *Météor: A successful application of B in a large project*. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) *FM 1999*. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
- [BA05] Badeau, F., Amelot, A.: *Using B as a High Level Programming Language in an Industrial Project: Roissy VAL*. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) *ZB 2005*. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
- [BB99] Büchi, M., Back, R.: *Compositional Symmetric Sharing in B*. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) *FM 1999*. LNCS, vol. 1708, p. 431. Springer, Heidelberg (1999)
- [BBP<sup>+</sup>03] Bert, D., Boulmé, S., Potet, M.-L., Requet, A., Voisin, L.: *Adaptable Translator of B Specifications to Embedded C programs*. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME 2003*. LNCS, vol. 2805. Springer, Heidelberg (2003)
- [BDF<sup>+</sup>04] Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: *Verification of object-oriented programs with invariants*. *Journal of Object Technology* 3(6), 27–56 (2004)
- [BN04] Barnett, M., Naumann, D.A.: *Friends need a bit more: Maintaining invariants over shared state*. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
- [BP07] Boulmé, S., Potet, M.-L.: *Interpreting invariant composition in the B method using the spec# ownership relation: A way to explain and relax B restrictions*. In: Julliand, J., Kouchnarenko, O. (eds.) *B 2007*. LNCS, vol. 4355, pp. 4–18. Springer, Heidelberg (2006)
- [Cle02] ClearSy. *Le Langage B. Manuel de reference, version 1.8.5*. ClearSy (2002)
- [Dij76] Dijkstra, E.W.: *A discipline of Programming*. Prentice-Hall, Englewood Cliffs (1976)
- [GP85] Gries, D., Prins, J.: *A New Notion of Encapsulation*. In: *Proc. of Symposium on Languages Issues in Programming Environments, SIGLPAN (1985)*
- [Hab01] Habrias, H.: *Spécification formelle avec B*. Hermès Science Publications (2001)
- [Hoa72] Hoare, C.A.R.: *Proof of correctness of data representations*. *Acta Informatica* 1, 271–281 (1972)
- [LM04] Leino, K.R.M., Müller, P.: *Object invariants in dynamic contexts*. In: Odersky, M. (ed.) *ECOOP 2004*. LNCS, vol. 3086, pp. 491–516. Springer, Heidelberg (2004)
- [LM06] Leino, K.R.M., Müller, P.: *A verification methodology for model fields*. In: Sestoft, P. (ed.) *ESOP 2006*. LNCS, vol. 3924, pp. 115–130. Springer, Heidelberg (2006)
- [LR98] Lanet, J.-L., Requet, A.: *Formal Proof of Smart Card Applets Correctness*. In: Schneier, B., Quisquater, J.-J. (eds.) *CARDIS 1998*. LNCS, vol. 1820. Springer, Heidelberg (2000)
- [MG90] Morgan, C., Gardiner, P.H.B.: *Data Refinement by Calculation*. *Acta Informatica* 27(6), 481–503 (1990)
- [MPHL06] Müller, P., Poetzsch-Heffer, A., Leavens, G.T.: *Modular Invariants for Layered Object Structures*. *Science of Computer Programming* (2006)

- [NB04] Naumann, D.A., Barnett, M.: Towards imperative modules: Reasoning about invariants and sharing of mutable state. In: LICS 2004, pp. 313–323. IEEE Computer Society Press, Los Alamitos (2004)
- [Pot02] Potet, M.-L.: Spécifications et développements formels: Etude des aspects compositionnels dans la méthode B. Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, décembre 5 (2002)
- [SL00] Sabatier, D., Lartigue, P.: The Use of the B method for the Design and the Validation of the Transaction Mechanism for smart Card Applications. *Formal Methods in System Design* 17(3), 245–272 (2000)

# Designing Old and New Distributed Algorithms by Replaying an Incremental Proof-Based Development

Dominique Cansell<sup>2</sup> and Dominique Méry<sup>1,\*</sup>

<sup>1</sup> Nancy-Université, Université Henri Poincaré Nancy1 & LORIA

<sup>2</sup> Université de Metz & LORIA

**Abstract.** The paper reports on practical experience with the event B method, when developing case studies, especially distributed algorithms, which are very complex to verify in practice. Using the event B method, we develop a famous distributed algorithm, namely the leader election protocol for an acyclic network, generally known as the IEEE 1394. The algorithm exists and the refinement helps us to model it entirely in an elegant way. The final model is very close to the real algorithm. Only the termination proof is missing, since it is a probabilistic algorithm, as well as the contention resolution, which is solved at a global abstract level. Modelling is clearly fundamental and complex; it should be carried out by persons able to use refinement and to manage abstractions or more precisely abstract models and proofs. Advantages of such an incremental development are multiple what we quote here and that will be explained in detail. We replay the development to improve the proof process and we obtain new distributed algorithms solving the leader election protocol problem. Two strategies are used to build the new algorithms; a first strategy is called the contention resolution; a second strategy is called the contention prevention and is based on a priority among possible nodes of the network. The two resulting algorithms are cheaper than the original IEEE 1394 protocol and neither acknowledgement, nor confirmation is required. We show how the techniques of localisation help in deriving the final distributed algorithm. The paper is an extended version of the complete development of the two new algorithms and it aims to emphasize methodological aspects related to the event B development.

**Keywords:** Formal method, B event-based method, refinement,safety, proofs, distributed systems.

## 1 Introduction

### 1.1 Playing with Proof-Based Developments

This paper shows how we re-use a previous incremental proof-based development of a distributed algorithm [3] to discover two other simpler correct-by-construction

---

\* Work of Dominique Cansell and Dominique Méry are supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

distributed algorithms: the resulting algorithm does not exist. Abstract models of the previous development express a context that leads us to the emergence of a simple idea using refinement and a tool which yields a new correct solution by construction. Our methodology uses the event B method and the Click'n'Prove tool [2], which together become a true laboratory for replaying developments.

The case study is the IEEE 1394 leader election protocol [3] in an acyclic graph. We start the new modelling from the two first models which explain the goal of the algorithm: election of a leader by submission. The real algorithm uses several messages between two nodes of the graph: a *submission* message is sent by a node  $x$  to a node  $y$ ; when  $y$  has received the submission message, an *acknowledgement* message is sent by  $y$  to  $x$ , which sends a *confirmation* message to  $y$ . Our new algorithms use only the *submission* messages.

## 1.2 Modelling, Design and Verification of Distributed Algorithms

A distributed system is a set of processors able to interact with each other. Three principal paradigms of interaction are known: Message Passing, Shared Memory, and Local Computations. In all three paradigms the processors are represented by nodes (vertices) and the interaction links are represented by edges (arcs) of a graph. In models based on communication, the processors interact through ‘atomic’ communication primitives: messages can be sent and received along the communication links (channels), or read/write operations can be executed on special storage registers associated with the communication links. In local calculi, interactions between nodes are described by graph rewriting rules. Different (sub) models of this kind can represent various forms of system architecture on different levels of abstraction and with different levels of detail in contact synchronisation.

The existence and description of distributed algorithms to solve a particular class of problems depends on the underlying network properties, including modes of communication and synchronisation of neighbour nodes, the identity or anonymity of processors, the initial states of the processors, or even knowledge about the topology of the network as a whole. Classical theoretical frameworks and notations, such as CSP, turn out to be cumbersome in describing certain distributed algorithms; they do not always reveal their characteristics and limitations and do not always support the formal proof of essential properties. In fact, CSP is a model for parallel and distributed computing based on the rendez-vous mechanism; it expresses concrete aspects. However, it is not a feature specific to CSP and distributed solutions are generally expressed in a too concrete programming language. Abstraction makes the design of distributed systems easier.

Many models related to local computation systems have been already introduced, as defined by Rosenstiehl et al. [15], Angluin [4], Yamashita and Kameda [11]. In [15] a synchronous model is considered, where vertices represent (identical) deterministic finite automata. The basic computation step is to compute the next state of each processor according to its state and the states of



its neighbours. In [4] an asynchronous model is considered. A basic computation step means that two adjacent vertices exchange their labels and then compute new labels. In [11], an asynchronous model is studied where a basic computation step means that a processor either changes its state and sends a message or it receives a message. In [7], networks are directed graphs coloured on their arcs; each processor changes its state depending on its previous state and on the states of its in-neighbours. All these models are concerned with local and global properties of networks and graphs.

In this context, the verification and design of distributed algorithms need to discover how the original algorithm has been designed; generally, verification techniques are based on the construction of abstractions of current distributed systems. In the case of distributed systems or algorithms, it is clear that the local states are elements of a global state and each site or each node has only a local view. Distributed systems are modelled using techniques based on non-determinism and fairness constraints. For instance, the I/O automata of Lynch [13] or TLA<sup>+</sup> [12] are two good examples of formalisms for expressing distributed systems or algorithms; the event B methodology provides concepts and tools for designing distributed algorithms by starting from very centralized views or models and then by introducing distribution.

### 1.3 Proof-Based Development for Distributed Algorithms

Proof-based development methods [15] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [15]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination.

A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [2].

At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations.

The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, mastering the proof complexity is a crucial issue. Even if a proof tool is available, its effective power is

limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development: this can be achieved by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements.

B Models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [10]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help to decrease the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques [14]; we have redeveloped the distributed algorithm [8] by proof-based refinement.

## 1.4 Summary of the Paper

Section 2 presents the event B method for developing distributed algorithms. Section 3 analyses the leader election problem and gives details on the two first common models of the development. In Section 4, we present the application of two strategies for addressing the question of the contention; the first class of solutions correspond to the contention resolution strategy and the second class of solutions is based on the avoidance of contention using priority counters. The two new algorithms are obtained from each class and do not require acknowledgements. Section 5 provides a sketch of the localisation techniques. Finally, section 6 concludes the paper.

## 2 Proof-Based Developments of Distributed Algorithms in Event B

Using the event B method, we have developed a famous distributed algorithm - the leader election protocol in an acyclic network namely the IEEE 1394 protocol. The complete development provides a sequence of B models, which give a progressive description of the final algorithm. The main result is the effective development of a complete algorithm with a documentation validated by proofs. B models provide explanations of the principles of the election; the final B model was very close to the real protocol. The proof of invariants was made easier by the use of the refinement and by applying the principle of the parachutist who is approaching a target and who is discovering the reality of the target. Only the termination proof is not given, since the termination of the algorithm is probabilistic; we are currently analysing the refinement of B models to introduce concepts related to time constraints that are mentioned in the final version of the IEEE 1394. This analysis is out of the scope of this paper. Modelling is a tough activity and must be led by experts using refinement and abstraction.

Advantages of such an incremental proof-based development are quoted here and are as follows:

- a progressive comprehension of the algorithmic process, because we used the refinement to introduce details of the algorithm in a gradual way.
- a discovery of errors of modelling detected and corrected via the proof.
- the presentation in a progressive way of such an algorithm allows our students and our colleagues to understand our development more easily and a posteriori. Our explanations are made easier by abstract descriptions.
- this can be achieved as explanations can be given to the customer, a priori, when a model is being constructed.
- evidence, that it is easier to prove abstract models. For technical refinements related to localisations, some proof obligations may no longer be automatically discharged because there are too many assumptions. In general this evidence is often technical and comprehensible. However, in general, our students manage to discharge automatic proof obligations and lightly interactive proof obligations: generally, only one remains unproved and is the most tricky one.

The IEEE 1394 case study enabled us to state a methodology of development for distributed algorithms and we reuse the first two models of the development [\[3\]](#):

- The starting phase is an abstract model - called the one-shot model - and it expresses the specification of the problem to solve. It has only one event, which is modelling the execution of the whole computation process; the event defines a pre/post specification in the event B modelling language. The event is a computation relation and does not express any execution model.
- A second phase of modelling gradually introduces the essence of the algorithm, namely the submission; the first refinement produces a refinement model based on an induction principle over a tree structure. This refinement model includes two events; one event corresponds to a refined version of the election, the second event models the induction step helping the tree-like structure to converge to a unique tree-like structure with a unique root node. The induction step is based on a submission principle - a node  $x$  sends a submission message to a node  $y$  and accepts the principle of submission (slave/master). We note that this is the most difficult proof to carry out in an interactive way and we claim that this proof would have been very difficult in a more concrete model.
- A third phase adds the management of messages between nodes; a node may send message to its neighbours and may receive messages from its neighbours. The network is still abstract and the information is still global. New events are defined to model the submission messages, the acknowledgement messages and the confirmation messages.
- A fourth phase localises global data and global variables; the knowledge per node is effectively local. The decision of localisation is taken late, since the technical proofs are easier in the lower abstract levels. If the localisation is

not possible, it may be possible that the algorithmic principles introduced in the previous refinement steps do not allow a distribution of data and actions. A solution would be to improve the previous refinement models.

The first phase is a very crucial step; the underlying mathematical theory should be precisely defined. In the IEEE 1394 case study, the definitions of trees, graphs, induction over trees, etc should have been stated in the set-theoretical language; the problem has been stated in the mathematical theory and theorems, such as the existence of a spanning tree for any acyclic graph. Many properties for data structures are generally proved in algorithmic books and should be integrated in a safe way; the proof tool helps us to discharge the proofs of these properties. The effort involved in these proofs, is important; it will be done only one time and will be integrated into a library of proved theories.

When the algorithm exists, the last model should be close to the algorithm; it helps us in our refinement steps; it allows us to understand how the algorithm is working and to extract at an abstract view of the algorithm. The main advantage of an incremental development is not to construct the final algorithm, but the incremental development collects much information about the algorithm that solves the problem and about the mathematical properties related to that problem.

It is possible to modify proved models or to start a new development from any model; for instance, the IEEE 1394 protocol leader election solves contention by a mechanism based on the choice of a timeout delay. In our first development, we decided to choose the higher node in the two competing nodes. This choice of refinement led to a new algorithm which is possible and correct with respect to the requirements. Another consideration is the complexity of the proof; is it possible to write a simpler proof of the property that is difficult to prove? How can we simply prove that the last step of the algorithm determines the leader among two possible candidates rather than allow two pairs of nodes in the network compete for the leadership? Can we find a more efficient algorithm? For instance, the efficiency of a distributed algorithm is based on the number of messages which are required for a given task. The analysis of the proved models shows us that a new invariant simplifies the interactive proofs. We present the first two models of the development [\[3\]](#).

### 3 Analysis of the Leader Election Problem

#### 3.1 Formalizing the Leader Election Problem in Event B

This first step is crucial in the event B development. It aims to state the leader election problem in a event B model, which expresses the properties of data structures and the existence of a solution. The data structures used are undirected acyclic graph. The solution exists if there is a directed spanning tree for each node considered as a root. The initial model is called ELECTION\_1 and simply gathers definitions:

DEFINITIONS	
$\text{tree}(r, N, t) \hat{=} \left( \begin{array}{l} r \in N \\ t \in N - \{r\} \rightarrow N \\ \forall q \cdot \left( \begin{array}{l} q \subseteq N \wedge \\ r \in q \wedge \\ t^{-1}[q] \subseteq q \\ \implies \\ N \subseteq q \end{array} \right) \end{array} \right)$	$\text{spanning}(r, t, g, N) \hat{=} \text{tree}(r, N, t) \wedge t \subseteq g$

- $N$  stands for the set of nodes in the graph.
- $t$  is a relation conforms to  $g$ ; it defines a tree over  $g$ .
- $r$  is the root of the tree defined by  $t$ .

and properties as follows:

$N$  is the (finite) set of nodes and  $g$  is a symmetric and irreflexive graph such that each node  $x$  is the root of a unique spanning tree  $f(x)$  of the graph.

PROPERTIES $g \subseteq N \times N$ $g = g^{-1}$ $\text{id}(N) \cap g = \emptyset$ $f \in N \rightarrow (N \leftrightarrow N)$ $\forall(r, t) \cdot (r \in N \wedge t \in N \leftrightarrow N \implies t = f(r) \Leftrightarrow \text{spanning}(r, t, g, N))$
--

$f$  states that a unique spanning tree for  $g$  can be assigned to each node;  $f$  is a total function over  $N$  defining the tree assigned to a given node. The first abstract model only has the event  $\text{elect}_1$  which expresses the election of  $l$  and builds the spanning tree  $s$  in one shot:

$\text{elect}_1 \hat{=} \text{ANY } x \text{ WHERE } x \in N \text{ THEN } l, s := x, f(x) \text{ END}$
---

### 3.2 First Algorithmic Model of the Leader Election Problem

The second model introduces the submission algorithm. A node  $x$  who has been asked to be the leader by each neighbour, is in fact the leader of the global network. The new refined event  $\text{elect}_2$  has a guard checking the condition of the election. The event  $\text{progress}_2$  can be triggered, when each neighbour of a given node  $x$  has asked him to be the leader, but if one neighbour  $y$  of  $x$  has not asked; the node  $x$  becomes the child of  $y$ . The progress is ensured by the convergence of the current forest towards a spanning tree, which exists by properties of the mathematical structure.

The leader election problem is characterized by definitions of acyclic undirected graphs and the event  $\text{elect}_1$  which expresses the proved [3] existence of a directed spanning tree for each node. The next goal is to refine the current model  $\text{ELECTION}_1$  to obtain a model  $\text{ELECTION}_2$  which is more algorithmic than  $\text{ELECTION}_1$ .

This refinement introduces the essence of the algorithm: the process of election is based on submission. The submission is atomic: two nodes  $x$  and  $y$ , where  $x$  has accepted the submission of all its neighbours except  $y$ , gives its submission to  $y$  and  $y$  accepts the submission. The model  $\text{ELECTION}_2$  introduces a new event which simulates the progressive computation of the directed spanning tree; the process models an inductive step and provides a simple expression of the convergence of a forest toward a tree. The model  $\text{ELECTION}_2$  has an event  $\text{progress}_2$  which helps the process to reach the tree. The guard is true for two nodes  $x$  and  $y$ , when  $x$  is the root of the forest and  $y$  is a neighbour of  $x$  under the relation defining the underlying graph structure. Its effect is to add the link which makes the node  $x$  the child of  $y$  in  $t$ . When  $x$  has accepted the submission of all its neighbours, the event  $\text{elect}_2$  can be triggered.

```

progress2 ≐
  ANY x, y WHERE
    x ↦ y ∈ g ∧
    x ∉ dom(t) ∧
    y ∉ dom(t) ∧
    g[{x}] = t-1[{x}] ∪ {y}
  THEN
    t := t ∪ {x ↦ y}
  END

```

```

elect2 ≐
  ANY x WHERE
    x ∈ N ∧
    g[{x}] = t-1[{x}]
  THEN
    l, s := x, t
  END

```

The difficulty is in proving the refinement of the event  $\text{elect}_1$  by the event  $\text{elect}_2$ . The problem is to prove that, when each neighbour of a node  $x$  is its child, then the resulting structure  $t$  is a spanning tree for  $g$  rooted by  $x$ :  $f(x)$  returns the tree  $t$ . The invariant expresses the fact that the set of spanning trees converge towards a spanning tree.

```

t ∈ N ↔ N
t ∩ t-1 = ∅
dom(t) ◁ (t ∪ t-1) = dom(t) ◁ g

```

The invariant states that  $t$  is a partial function over  $N$  ( $t$  is intended to model the functional relation called *be the child of*);  $t$  is asymmetric. The third sentence expresses the progression of the process and the fact that only one edge is chosen for the tree  $t$  (either  $x$  to  $y$  or  $y$  to  $x$ ); moreover,  $t$  is  $g$  up-to the reverse. The property  $x \notin \text{dom}(t)$  is required and is proven automatically by the automatic prover (predicate prover). The next difficult step is the proof of  $N \subseteq \{n | n \in N - \{x\} \wedge n \mapsto f(x)(n) \in t\} \cup \{x\}$  which is discharged using the induction principle over the spanning tree rooted by  $x$ ;  $f(x)$  and  $t$  are in

fact equal on  $N - \{x\}$  and  $x$  is not in their domain; hence,  $f(x)$  and  $t$  are equal on  $N$ .

The proof summary of the first model is 12 proof obligations generated by the tool and 5 proved by interaction.

The next proof is to show that there are at most two nodes  $x$  and  $y$  such that  $x$  and  $y$  are neighbours and are not yet in the tree  $t$  under construction: if  $y$  (resp.  $x$ ) requests to  $x$  to be its child ( $y \mapsto x \in g$  and  $x \notin \text{dom}(t)$ ), then  $y \mapsto x$  is appended to the set  $t$ , which is  $f(x)$  and symmetric for  $x$  and  $y$ . One concludes that  $t \subseteq f(x)$  and  $t \subseteq f(y)$ . Both properties  $x \notin \text{dom}(t)$  and  $y \notin \text{dom}(t)$  are discharged. The idea is to propose an invariant expressing this fact:

$$\forall x \cdot (x \in N - \text{dom}(t) \implies t \subseteq f(x))$$

In fact, the additional invariant tells us simply that a spanning tree always exists and that each node not in the domain of  $t$  is a candidate for the leadership and that node may become the leader. The leadership can be constrained by a variable attached to each node. A boolean variable, called *root*, forbids (freezes) the submission to the current node. Only one node should have a local variable *root* set to *true*, if one wants to avoid the deadlock. Consequently, if a node *wants* to be the leader, he/she can decide to wait requests from his/her neighbours; however, if two nodes are using the same strategy, the global system will deadlock. The refinement proof is based on the property  $\text{dom}(t) = N - \{x\}$ , whose proof is divided into the following steps:

1.  $x \notin \text{dom}(t)$ : automatically discharged by the predicate prover.
2.  $N \subseteq \text{dom}(t) \cup \{x\}$ : the proof is based on the induction principle over the spanning tree rooted by  $x$   $f(x)$ ; the universally quantified variable is instantiated by  $\text{dom}(t) \cup \{x\}$ . After this instantiation, the predicate prover automatically discharges the waiting goal.

This proof summary of the first refinement is 10 proof obligations generated by the tool, two of which require interaction. The proof of the two last nodes is based on:

$$\forall (x, y) \cdot \left( \begin{array}{l} x \mapsto y \in g \wedge \\ x \notin \text{dom}(t) \wedge \\ y \notin \text{dom}(t) \wedge \\ g[\{x\}] = t^{-1}[\{x\}] \cup \{y\} \wedge \\ g[\{y\}] = t^{-1}[\{y\}] \cup \{x\} \\ \implies \\ \text{dom}(t) = N - \{x, y\} \end{array} \right)$$

The property is proved using the same induction principle over spanning trees  $f(x)$  and  $f(y)$ ; the universally quantified variable is instantiated by  $\text{dom}(t) \cup \{x, y\}$ . The current state of the development refers to a first model expressing the one-shot election ELECTION\_1 and a second model ELECTION\_2 refining

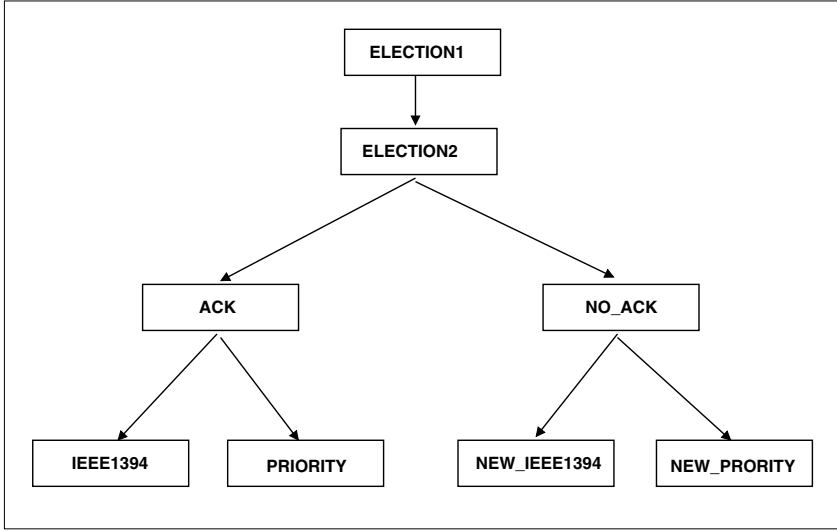


Fig. 1. Proof-based development for the leader election problem

ELECTION<sub>1</sub> (see the figure 1). Deadlock-freeness is proved by adding the condition that at least one event is triggered; the graph is connected and only two nodes can be in contention. In fact, the global process is a set of trees converging to a tree.

The next step is crucial, since it prepares the distribution of models by introducing messages. The localisation of variables should be delayed as far as possible, while refining models: proofs are generally less complex on abstract models. Before we give the next refinements, we should recall that the first proof-based development of the IEEE 1394 leader election protocol [3] was based on a description using a C-like programming language, where we had misunderstood how the contention problem was solved. The contention problem is either solved, or avoided:

- avoidance of contention is ensured by a mechanism of priority between nodes; since at most two nodes may be in contention, we elect the node with the highest priority to be the leader; the solution requires a total ordering over nodes and the identification of each plugged component: refinement model PRIORITY (see [9] for details of the development).
- resolution of contention is solved by a probabilistic mechanism; when two nodes are in contention, both go back to the state before sending a message to the other node, as long as the contention state is happening; there may exist an infinite sequence of contention states but under probabilistic assumption, the contention is solved: IEEE1394 [3].

Both solutions use acknowledgements and we have located both solutions under the node **ACK**. We now show that acknowledgement messages can be removed



and we can partially replay both designs of algorithm and derive two new solutions without acknowledgements (see, in the figure 1, the two solutions under the node **NO\_ACK**).

## 4 The Leader Election Protocol without Acknowledgement

The development of the two new distributed algorithms shares events, which are independant on the contention problem. In order to improve the presentation, the first subsection describes events not related to the contention management; the second subsection addresses the contention resolution and the use of unnamed nodes; the last subsection derives a solution without acknowledgement and is based on the priority counters for avoiding the contention.

### 4.1 Contention-Free Development Part

In our initial work [3], we introduced events for managing messages during the second refinement. Since messages were followed by one acknowledgement (*ack*), we should also model confirmations (*t*). We discovered that there were confirmations, when we localised the graph and the variables. The call for the workshop dedicated to the protocol IEEE 1394 asked whether confirmations were necessary. At that time, we answered yes. The current work answers no, because we can assert that acknowledgements, and then confirmations, are not required. On the other hand, we can not remove the sending of messages. The event **progress** is introduced and the node  $x$  sends a message to  $y$ ; the reader will notice that the message is sent only, if  $x$  did not yet accept the submission of  $y$  ( $y \mapsto x \notin t$ ) and that  $x$  did not send yet its message ( $x \notin \text{dom}(m)$ ):

<pre> send_msg3  ≐   ANY x, y WHERE     x ↦ y ∈ g ∧     g[{x}] = t<sup>-1</sup>[{x}] ∪ {y} ∧     y ↦ x ∉ t ∧     x ∉ dom(m)   THEN     m := m ∪ {x ↦ y}   END </pre>
--

<pre> progress3  ≐   ANY x, y WHERE     x ↦ y ∈ m - t ∧     y ∉ dom(m)   THEN     t := t ∪ {x ↦ y}   END </pre>
---

The abstract event **progress** expresses that a node  $y$ , whose neighbour  $x$  has asked to be its child, has got the message and has accepted the parenthood of  $x$ . In the previous version, the node  $y$  receives the message of  $x$  ( $x \mapsto y \in m$ ) and  $y$  did not yet accept it ( $x \mapsto y \notin t$ );  $y$  can also be in the situation of sending a request itself and it did not send a message ( $y \notin \text{dom}(m)$ ). Invariants are based on invariants of our previous work, but are much simpler and are simpler because *ack* and *t* are not in our models.

The first three lines express typing information and inclusion properties: messages follow the route of the graph. The fourth line expresses that, if  $x$  has sent a message to  $y$  ( $x \mapsto y \in m$ ), then  $x$  is in a submission status and  $x$  accepts that  $y$  is its parent ( $g[\{x\}] = t^{-1}[\{x\}] \cup \{y\}$ ).

$$\begin{array}{l} m \in N \leftrightarrow N \\ t \subseteq m \\ m \subseteq g \\ \forall (x, y) \cdot (x \mapsto y \in m \implies g[\{x\}] = t^{-1}[\{x\}] \cup \{y\}) \end{array}$$

## 4.2 Contention Resolution

$$\begin{array}{l} x \mapsto y \in m - t \\ y \in \text{dom}(m) \end{array}$$

The guard of `progress3` is not validated, when  $x$  has sent a message to  $y$  and when  $y$  has sent a message.

The current events are not enough for reacting to this state and the current model is completed by a new event `contention3`, which discovers the contention status and which reacts to the previous guard. We have now two new events for managing contention: `contention3` discovers the contention and `solve_contention3` solves the contention. A contention channel called  $c$  is used to control the contention status.

$$\begin{array}{l} \text{contention}_3 \hat{=} \\ \text{ANY } x, y \text{ WHERE} \\ \quad x \mapsto y \in m - (t \cup c) \wedge \\ \quad y \in \text{dom}(m) \\ \text{THEN} \\ \quad c := c \cup \{x \mapsto y\} \\ \text{END} \end{array}$$

$$\begin{array}{l} \text{solve\_contention}_3 \hat{=} \\ \text{ANY } x, y \text{ WHERE} \\ \quad x \in N \wedge y \in N \wedge \\ \quad c = \{x \mapsto y, y \mapsto x\} \\ \text{THEN} \\ \quad m := m - c \quad || \\ \quad c := \emptyset \\ \text{END} \end{array}$$

In the new invariant, if  $x$  has sent a message to  $y$  and if  $y$  has sent a message  $y \in \text{dom}(m)$ , then  $y$  has sent its message to  $x$  (no other choice) and this message was not yet confirmed ( $y \mapsto x \in m - t$ ). It includes properties satisfied by the variable  $c$ .

$$\forall (x, y) \cdot \left( \begin{array}{l} x \mapsto y \in m - (t \cup c) \\ y \in \text{dom}(m) \end{array} \implies y \mapsto x \in m - t \right)$$

$$\begin{array}{l} c \subseteq m \\ t \cap c = \emptyset \\ \forall (x, y) \cdot (x \mapsto y \in c \implies y \mapsto x \in m - t) \end{array}$$

The proof summary of the new refinement is: 33 proof obligations are generated by the tool and no interaction is necessary. At this point, we know that

confirmations are not necessary and confirmations can not be justified by lossy channels. The number of messages is three times less than in the first development. As long as a node which has sent a message, does not get a message from another node, there is no contention; if there is a problem on the protocol, the timeout mechanism allows us to reinitialise the protocol for a new election. The current model is called NEW\_IEEE1394 and it can be refined to localise variables.

### 4.3 Contention Prevention

Another way to deal with the contention problem is to use a priority technique, which will avoid the contention. The solution was discovered after a misunderstanding of the IEEE 1394 initial solution and we did not know it. In fact, the solution was mentioned by N. Lynch [13] and was first proposed by D. Anluin [4].

When two nodes are in contention (and at most two nodes can be in contention, proved mechanically and formally), each node can not send an acknowledgement to the other node; one of them should not be able to send this ack and the other one must do it. The main idea is to introduce a *unique counter* called *ctr* and it means that each node is uniquely identified and must be identifiable. The assumption is that there is a mechanism for naming, *ctr* is a total injection from Nodes into natural numbers. In a real network, one can assume that equipment might be uniquely identified by an unique address, for instance, but it is not the general rule.

The new event is called `avoid_solve_cnt3`. Like for `send_ack3`, it adds the pair  $x \mapsto y$  to  $t$ .

```

avoid_solve_cnt =
  ANY  $x, y$  WHERE
     $x \mapsto y \in m - t \quad \wedge \quad y \in \text{dom}(m) \wedge \text{ctr}(x) < \text{ctr}(y)$ 
  THEN
     $t := t \cup \{x \mapsto y\}$ 
  END

```

The two differences with the guard of event `progress3` concern the condition  $y \in \text{dom}(m)$ , which is true in `avoid_solve_cnt` and false in `progress3` and the guard  $\text{ctr}(x) < \text{ctr}(y)$  is added to the event `avoid_solve_cnt`. Since *ctr* is an injection, both nodes  $x$  and  $y$  can not be triggered concurrently. The proof of the invariant requires the following additional invariant:

$$\forall (x, y) \cdot \left( \left( \begin{array}{l} x \mapsto y \in t \wedge \\ y \mapsto x \in m \end{array} \right) \implies \text{ctr}(x) < \text{ctr}(y) \right)$$

The proof summary of the current refinement is: 13 proof obligations are generated by the tool and no interaction is necessary. The current model is called NEW\_PRIORITY and it can be refined to localise variables.

## 5 Localisation of Events and Data

The final step is to localise information on the events and the data. For instance, we have localised the graph  $g$  by the constants  $n$  (for neighbour) and the variable  $t$  with the variable  $b$  like in our previous work [3]. The gluing invariant is required to localise information. For instance, the event `elect` is a refined version of the initial event of the model `ELECTION_1` and the guard is expressed using  $n(x)$  and  $b(x)$  which are related to the definition of  $g$  and  $t$  by the gluing property (for  $n$ ) and by the gluing invariant (for  $b$ ).

$$\begin{aligned}
 n &\in N \rightarrow \mathbb{P}(N) \\
 \forall x \cdot (x \in N &\implies n(x) = g[\{x\}]) \\
 b &\in N \rightarrow \mathbb{P}(N) \\
 \forall x \cdot (x \in N &\implies b(x) = t^{-1}[\{x\}])
 \end{aligned}$$

$$\begin{aligned}
 \text{elect} &\hat{=} \\
 &\text{ANY } x \text{ WHERE} \\
 &\quad x \in N \wedge \\
 &\quad n(x) = b(x) \\
 &\text{THEN} \\
 &\quad l := x \\
 &\text{END}
 \end{aligned}$$

The final refinement of `NEW_IEEE1394` leads to the modelling of wires. For each node, the variable  $D$  specifies the state (*high* or *low*) of the wire between each node, the variable  $D$  specifies the state (*high* or *low*) of the wire between neighbours. If  $y \in \text{dom}(D(x))$ , then there exists a wire between  $x$  and  $y$ .  $D(x)(y) = \text{low}$  if  $x$  does not accepted the submission from  $y$ . Variable  $B$  is a function with gives the state of a node. This state allows us to determine if the node has sent a message. We have substituted variables  $d$  (resp.  $bm$ ) by  $D$  (resp. by  $B$ ).  $M$  stands for the messages, which are effectively moving in the real network, since we have the property  $m = M \cup t \cup c$ . We have introduced this model in several refinement steps.

$$\begin{aligned}
 d(x) &= n(x) - b(x) \\
 r(x) &= \text{card}(d(x)) \\
 D &\in N \rightarrow (N \leftrightarrow \{\text{low}, \text{high}\}) \\
 \forall x \cdot (x \in N &\implies \text{dom}(D(x)) = n(x)) \\
 \forall x \cdot (x \in N &\implies d(x) = D(x)^{-1}[\{\text{low}\}])
 \end{aligned}$$

$$\begin{aligned}
 B &\in N \rightarrow \{\text{low}, \text{high}\} \\
 bm &= \text{dom}(m) \\
 bm &= B^{-1}[\{\text{high}\}]
 \end{aligned}$$

$$\begin{aligned}
 \text{send\_msg}_9 &\hat{=} \\
 &\text{ANY } x, y \text{ WHERE} \\
 &\quad x \in N \wedge \\
 &\quad B(x) = \text{low} \wedge \\
 &\quad y \in D(x)^{-1}[\{\text{low}\}] \wedge \\
 &\quad r(x) = 1 \\
 &\text{THEN} \\
 &\quad M := M \cup \{x \mapsto y\} \quad || \\
 &\quad B(x) := \text{high} \\
 &\text{END}
 \end{aligned}$$

$$\begin{aligned}
 \text{progress}_9 &\hat{=} \\
 &\text{ANY } x, y \text{ WHERE} \\
 &\quad x \mapsto y \in M \wedge \\
 &\quad B(y) = \text{low} \\
 &\text{THEN} \\
 &\quad D(y)(x) := \text{high} \quad || \\
 &\quad r(y) := r(y) - 1 \quad || \\
 &\quad M := M - \{x \mapsto y\} \\
 &\text{END}
 \end{aligned}$$

The proof summary of the fourth refinement is: 20 (9+11) proof obligations are generated by the tool and 7 (3+4) among them require interaction. We

do not give the complete set of events of the last models but note that the localisation is systematic, as long as information of each node contain localisable information.

## 6 Conclusion and Future Work

The current work is a general presentation of work on the leader election problem in a distributed environment. We have used the development of the classical leader election protocol of the IEEE 1394 and we propose an improvement of the classical solution. Two new algorithms are developed incrementally by replaying previous developments and we show that acknowledgements are not needed in the leader election protocol. We have shown that a proof-based development can be easily reused to improve and/or discover new distributed algorithms. The new algorithm is still working like a submission algorithm but the traffic of messages is simpler. We replay from our previous development starting from the model which introduces messages. The call for the workshop dedicated to the protocol IEEE 1394 asked whether confirmations were necessary. At that time, we answered yes. The current work answers no, because we can assert that neither acknowledgements, nor confirmations are required. On the other hand, we can not remove the sending of messages.

Proof summary for the first three models of the four leader election algorithms is given in the array as follows:

Models	Number of proof obligations	Number of interactive steps
ELECTION_1	2	1
ELECTION_2	10	2
IEEE1394	41	0
PRIORITY	31	2
NEW_IEEE1394	33	0
NEW_PRIORITY	13	0
TOTAL	120	5

The localisation refinement requires the same proof obligations as those in the initial development in [3] and in [9] for PRIORITY. Future work will apply the incremental development on other distributed algorithms.

**Acknowledgments.** We thank J.-R. Abrial for his fruitful advices and comments. Rosemary Monahan reads and annotates a current version; she adds english language in our text; thanks for your advices. Finally, a referee gives a very detailed report on our paper and allows us to improve the understanding of the final version; we thank you.

## References

1. Abrial, J.-R.: *The B book - Assigning Programs to Meanings*. Cambridge University Press, Cambridge (1996)
2. 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)
3. Abrial, J.-R., Cansell, D., Méry, D.: A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing* 14(3), 215–227 (2003)
4. Angluin, D.: Local and global properties in networks of processors. In: *Proceedings of the 12th Symposium on theory of computing*, pp. 82–93 (1980)
5. Back, R.: On correct refinement of programs. *Journal of Computer and System Sciences* 23(1), 49–68 (1979)
6. Bjørner, D., Henson, M.C. (eds.): *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, Heidelberg (2007)
7. Boldi, P., Vigna, S.: Computing anonymously with arbitrary knowledge. In: *Proceedings of the 18th ACM Symposium on principles of distributed computing*, pp. 181–188 (1999)
8. Cansell, D., Méry, D.: Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. *Theoretical Computer Science* (2006)
9. Cansell, D., Méry, D.: *The event-B Modelling Method: Concepts and Case Studies*, pp. 33–140. Springer, Heidelberg (2007); See [6]
10. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press, Cambridge (2000)
11. Kameda, T., Yamashita, M.: Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems* 7(1), 69–89 (1996)
12. Lamport, L.: *Specifying Systems: The TLA<sup>+</sup> Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading (2002)
13. Lynch, N.: *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco (1996)
14. Moreau, L., Duprat, J.: A Construction of Distributed Reference Counting. *Acta Informatica* 37, 563–595 (2001)
15. Rosenstiehl, P., Fiksel, J.-R., Holliger, A.: Intelligent graphs. In: Read, R. (ed.) *Graph theory and computing*, pp. 219–265. Academic Press, New York (1972)

# Ten Reasons to Metamodel ASMs

Angelo Gargantini<sup>1</sup>, Elvinia Riccobene<sup>2</sup>, and Patrizia Scandurra<sup>1</sup>

<sup>1</sup> Dip. di Ingegneria dell'Informazione e Metodi Matematici,  
Università degli Studi di Bergamo, Italy

`angelo.gargantini@unibg.it`, `patrizia.scandurra@unibg.it`

<sup>2</sup> Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy  
`elvinia.riccobene@unimi.it`

**Abstract.** Model-Driven Engineering (or MDE) is an emerging approach for system development which refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. MDE puts emphasis on bridges between different working contexts and on the integration of bodies of knowledge differently developed. We discuss the mutual advantages that the integration of MDE and Abstract State Machines (ASMs) would provide: MDE can gain rigour and preciseness, while ASMs get a standard abstract notation and a general framework for a wide tool interoperability.

## Introduction

Model-driven Engineering (MDE) [12] is an emerging approach for software development and analysis where models play the fundamental role of first-class artifacts. *Metamodelling* is a key concept of the MDE paradigm and it is intended as a modular and layered way to endow a language or a formalism with an abstract notation, so separating the abstract syntax and semantics of the language constructs from their different concrete notations. Furthermore, metamodelling allows to settle a “global framework” to enable otherwise dissimilar languages (of possibly different domains, the so called *Domain-specific languages*) to be used in an interoperable manner in different *technical spaces*, namely working contexts with a set of associated concepts, knowledge, tools, skills, and possibilities. Indeed, it allows to establish precise *bridges* (or *projections*) among the metamodels of these different domain-specific languages to automatically execute model transformations.

To achieve the goal of a global interoperability and merging of bodies of knowledge with rigor and preciseness, an integration of the MDE paradigm with formal methods is necessary [23].

This is a position paper mainly aimed at explaining the feasibility and the advantages of the proposed integration in the context of the Abstract State Machines (or ASMs).

We strongly believe that applying the MDE approach to ASMs is worthwhile at least for the following ten reasons, later discussed: (R1) to have a standard abstract notation as unified representation of ASM concepts independent of any

particular implementation platform; (R2) to have a graphical representation of ASMs also useful for teaching purposes; (R3) to have an interchange format among ASM tools; (R4) to have standard libraries and APIs to use in new or existing tools and programs; (R5) to automatically derive a family of languages, visual/textual notations and their parsers; (R6) to allow tool interoperability; (R7) to help the integration of existing tools; (R8) to help the development of new tools; (R9) to export ASMs to other environments and permit the integration with other specialized external notations and tools (for instance for property verification and testing); and last but not least, (R10) to complement the MDE with a formal approach.

The overall goal of our project is to develop a unified abstract notation for ASM and a general framework for a wide interoperability and integration of tools around ASMs. We started by defining the *AsmM*, a metamodel for ASMs, in [28,10]. We have, therefore, developed the ASMETA framework [10] as an instantiation of the metamodelling framework for ASMs, to create and handle ASM models exploiting the advantages offered by the metamodelling approach and its related facilities (in terms of derivatives, libraries, APIs, etc.). ASMETA provides a global infrastructure for interoperability of ASM application tools (new and existing ones) including ASM model editors, ASM model repositories, ASM model validators, ASM model verifiers, ASM simulators, ASM-to-Any code generators, etc.

Each reason stated above may not suffice to justify the effort of developing a complex metamodel as *AsmM*, but we hope that all together will convince even the most skeptical reader that the application of the MDE approach to ASMs is worthwhile. Note that not only ASMs would benefit from this approach: we expect that new synergies arise and that the cooperative interaction among ASMs and MDE creates an enhanced combined effect - as outlined in R10.

## R.1 To Have a Standard Abstract Notation

The success of the ASM as a system engineering method able to guide the development of hardware and software systems, from requirements capture to their implementation, is nowadays widely acknowledged [14]. The increasing application of the ASMs for academic and industrial projects has caused a rapid development of tools for ASM model analysis, namely simulation, property verification, and test generation. Among these tools we can cite *AsmGofer* [43], *AsmL* [8], *Xasm* [50], *TASM* [45], *ASM workbench* [15,49], *CoreASM* [18], *ATGT* [11] and other tools based on model checkers and theorem provers [25,49,26,42,20].

To encode ASM models, each tool uses a different syntax strictly depending on the implementation environment (C for XASM, Gofer for *AsmGofer*, .NET for *AsmL*, etc.), adopts a different internal representation of ASM models, and provides proprietary constructs which extend the basic mathematical concepts of ASMs. There is no agreement around a common standard ASM language. The result is that a practitioner willing to use an ASM tool needs to know its



own syntax and that most ASM researchers still use their own ASM notation, normally not defined by a grammar but in terms of mathematical concepts. Moreover, due to the lack of abstractness of the tool languages, the process of encoding ASM models is not always straightforward and natural, and one needs to map mathematical concepts, like ASM states (namely universes and functions defined on them), into types and structures provided by the target language.

To achieve the goals of developing a unified abstract notation for ASM, a notation independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs, and tackling the problem of ASM tool interoperability and integration, we exploited the *metamodelling* approach suggested by the MDE.

According to the MDE terminology, a metamodel defines the *abstract syntax* of a language, i.e. the structure of the language, separated from its *concrete notation*. A metamodel-based abstract syntax definition has the great advantage of being suitable to derive from the same metamodel (through mappings or projections) different alternative concrete notations, textual or graphical or both, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on. Therefore, a metamodel could serve as *standard representation* of a formal notation, establishing a common terminology to discriminate pertinent elements to be discussed, and therefore, helps to communicate understandings, especially if – as in the case of ASMs – the formal method is still evolving and the community is too much heterogeneous to easily come to an agreement on a unique textual notation. Note that the goal of achieving a standard and lean syntax for ASM specifications is shared with the CoreASM project [18].

In [10,28], a complete metamodel for ASMs is presented. As MDE framework, we adopted the OMG’s metamodelling platform. The *AsmM* (Abstract State Machines Metamodel) results into class diagrams developed using the MOF (the OMG’s metalanguage to define metamodels) modelling constructs (classes, packages, associations). We developed the metamodel in a *modular* and *bottom-up* way. We started separating the ASM static part represented by the *state*, namely domains, functions and terms, from the dynamic part represented by the *transition system*, namely the ASM rules. Then, we proceeded to model Basic ASMs, Turbo ASMs, and Multi-Agent (Sync/Async) ASMs, so reflecting the natural classification of abstract state machines.

The complete metamodel is organized in one package called **ASMETA**, which is further divided into four packages as shown in Fig. 1. Each package covers different aspects of ASMs. The dashed gray ovals in Fig. 1 denote the packages representing the notions of *State* and *Transition System*, respectively.

The **Structure** package defines the architectural constructs (modules and machines) required to specify the backbone of an ASM model. The **Definitions** package contains all basic constructs (functions, domains, constraints, rule declarations, etc..) which characterize algebraic specifications. The **Terms**

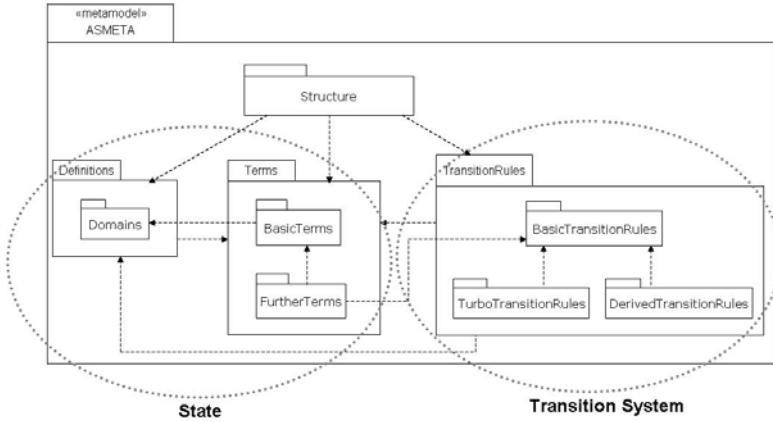


Fig. 1. Package structure of the ASM Metamodel

package provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. The **TransitionRules** package contains all possible transition rules schemes of Basic and Turbo ASMs. All transition rules derived from basic and turbo ones (e.g. the case-rule and the while-rule) are contained in the **DerivedTransitionRules** package.

Each class of the metamodel is equipped with a set of relevant *constraints*, OCL (version 2.0 [34]) invariants written to fix how to meaningfully connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. All OCL constraints have been syntactically checked by using the OCL checker OCLE [36].

In order to make AsmM able to support the languages of existing ASM tools, we have enriched the metamodel with particular forms of domains, special terms and derived rule schemes taken from these languages (see [28] for details). We have borrowed some extended terms including conditional terms and comprehension terms from ASM-SL, maps, sets and sequences from AsmL. Named rules with parameters (**RuleDeclaration**) appear in ASM-SL, while the concepts of submachine computation, iteration, and recursion, modelled in the AsmM respectively by the classes **SeqRule** and **IterateRule**, can be found in XASM as well as in AsmL (with an Object Oriented style, though). The agent representation in the AsmM is similar to the agents of AsmGofer, although Agents is an abstract domain in our metamodel, while Agent is an integer domain in AsmGofer. In Sect. R.9, we clarify how the metamodel is able to capture all foreseeable features of a possible ASM language; therefore, AsmM can be used as standard reference syntax.

## R.2 To Have a Graphical Abstract Notation

People often claim that formal methods are too difficult to put in practice due to their mathematical-based foundation. In this direction an abstract and visual

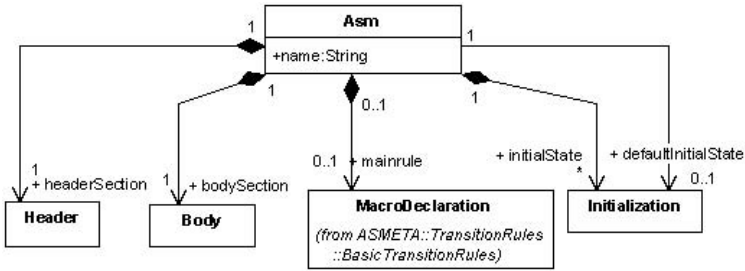


Fig. 2. Backbone

representation<sup>1</sup>, like the one provided by a MOF-compliant metamodel, delivers a more readable view of the modelling primitives offered by a formal method, especially for people, like students, who do not deal well with mathematics but are familiar with the standard MOF/UML. Therefore, the *AsmM* can be considered a complementary approach to [14] for the presentation of ASMs.

We here give evidence of how the metamodel can be useful to introduce ASMs. We present only a very small fragment of the *AsmM* whose complete description can be found in [28,10].

Fig. 2 shows the backbone of a *basic ASM*. An instance of the root class *Asm* represents an entire ASM specification. According to the working definition given in [14], a basic ASM has a *name* and is defined by a *Header* (to establish the signature), a *Body* (to define domains, functions, and rules), a *main rule*, and a set of initial states (instances of the *Initialization* class). Executing a basic ASM means executing its *main rule* starting from one specified *initial state*, i.e. the one denoted by the association end *defaultInitialState*.

The composite relationships between the class *Asm* (the *whole*) and its component classes (the *parts*) assures that each part is included in at most one *Asm* instance.

An ASM without a main rule and without a set of initial states (see the multiplicity of association ends *mainRule* and *initialState*) is called *module*<sup>2</sup> which is useful to syntactically structure large specifications.

The *Header* (see Fig. 3) consists of some *import clauses* and an optional *export clause* to specify the names which are imported from or exported to other ASMs (or ASM modules), and of its *signature* containing the *declarations* of the ASM domains and functions. Every ASM is allowed to use only identifiers (for domains, functions and transition rules) which are defined within its signature or imported from other ASMs or ASM modules.

The initialization of an ASM consists of a set of *initial states*. The class *Initialization* (not described here, see [28,10] for details) models the notion

<sup>1</sup> It should be noted that the visual representation for the (abstract) syntax of the language has not to be confused with a possible graphical notation for ASM specifications, which we refer to in Sect. R.5.

<sup>2</sup> This definition of module differs slightly from the module concept outlined in Chap. 2 of [14]; but, it has been accorded with the authors.

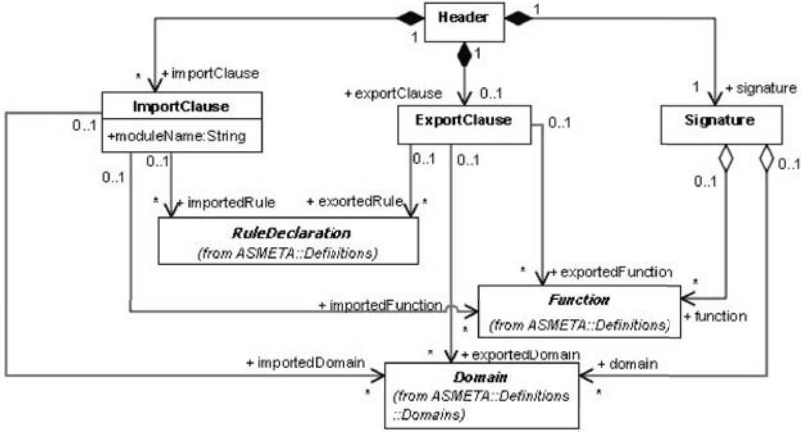


Fig. 3. Header

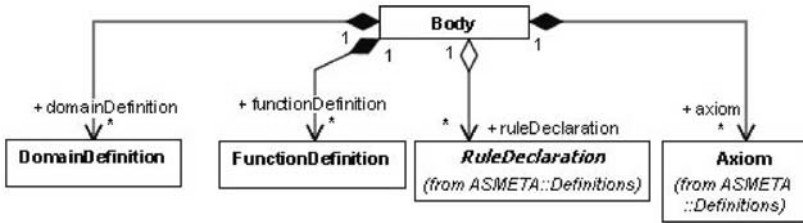


Fig. 4. Body

of an initial state. All possible initial states are linked (see Fig. 2) to an ASM by the association end `initialState` and one initial state is elected as `default` (see the association end `defaultInitialState`).

The `Body` (see Fig. 4) of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations of transition rules and definitions of axioms for constraints one wants to assume for some domains, functions, and transition rules of the ASM.

### R.3 To Have an Interchange Format

The interoperability among ASM tools can be (at least partially) achieved by a common interchange format. The work in [7] represents the first and the only attempt in this respect; it was based on the use of a pure XML format and unfortunately it has never been completed.

Whenever a language or formalism is specified in terms of a MOF-compliant metamodel, the MOF enables a standard way to generate an XMI (XML Metadata Interchange) [34] interchange format for models in that language. The main purpose of XMI is to provide an easy interchange of data and metadata between

modelling tools and metadata repositories in distributed heterogeneous environments. The XMI format is not for human consumption and it is not to be confused with the “concrete syntax” used by modelers to write their models. It has to be intended, instead, as an effective hard code to be automatically generated for interchanging purposes only.

To tackle the problem of ASM tool interoperability, we exploit the mechanism of deriving a specific XMI format from the metamodel. format, given as XML document type definition file (commonly named DTD), has been generated automatically from the AsmM in the MDR framework. First, we have drawn the AsmM with the Poseidon UML tool (v. 4.2) and saved it in the UML-XMI format. Then, we have converted it to the MOF 1.4 XMI by means of the UML2MOF transformation tool provided by the Netbeans MDR project. Finally, we have loaded the MOF model in the MDR framework of Netbeans and according to the rules specified by the MOF 1.4 to XMI 1.2 mapping specification [51], the DTD for AsmM models was generated.

In Section R.6 we discuss the role of the ASM-XMI format for interchanging ASM models among tools.

## R.4 To Have Standard Libraries

Applications and tools endowed with MOF-compliant metamodels, can have their *Java Metadata Interface* (JMI) [32] automatically generated. The JMI standard is based on the MOF 1.4 specification and defines a Java application program interface (API) for the creation, storage, access and manipulation of metadata in a MOF-based instance repository.

From the AsmM in the MDR framework we also automatically generate a JMI library for AsmM models (see [28] for more details). JMI can be used in client programs written in Java which want to manipulate ASM models (to read model structure, to modify parts of the specification and create new elements), as well as by tool developers to speed up the creation of new tools supporting ASMs. In Sections R.6, R.7, R.8 we show how the JMI can be useful for tool interoperability, integration of existing tools, and development of new tools.

Besides the XMI and JMI libraries already discussed, other libraries can be developed from MOF-compliant metamodels to provide additional facilities. Among them, we mention CMI (CORBA Metadata Interface) [17] for bridging with the middleware CORBA space.

## R.5 To Derive Concrete Notations and Their Parsers

A MOF-compliant metamodel allows to derive different alternative concrete notations, textual or graphical. Initially, we investigated the use of tools like HUTN (Human Usable Textual Notation) [31] or Anti- Yacc [19] which are capable of generating text grammars from specific MOF-based repositories. Nevertheless, we decided not to use them since they do not permit a detailed customization of the generated language and they provide concrete notations merely suitable

for object-oriented languages. There are better MOF-to-grammar tools now, like xText [22] of OpenArchitectureWare or TCS of AMMA [3], which we may consider to adopt in the future.

In [27] we define general rules on how to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we use these mapping rules to define an EBNF grammar from the AsmM for an ASM textual notation. The resulting language, called AsmetaL[3], is completely independent from any specific platform and allows a natural and straightforward encoding of ASM models. We design AsmM without any specific implementation platform in mind. The language derived from it does not contain any platform-dependent concept. Instead, the language of CoreASM explicitly contains directives for importing plug-ins written in Java, and the AsmL permits the use of the Microsoft .NET library.

In [27], we also provide guidance on how to assemble a JavaCC file given in input to the JavaCC parser generator [2] to automatically produce a parser for the EBNF grammar of the AsmetaL. This parser is more than a grammar checker: it is able to process ASM models written in AsmetaL, to check for their consistency w.r.t. the OCL constraints of the metamodel, and to create instances of the AsmM in a MDR MOF repository through the use of the AsmM-JMIs.

The complete AsmetaL grammar is reported in [28] and is also available in [10] together with the AsmetaL parser.

We have validated the metamodel and the AsmetaL notation to assess their usability and capability to encode ASM models. To this purpose, we have asked to a non ASM expert to port some specifications from [14] and other ASM case studies to AsmetaL. The task was completed within three man-months.

Up to now we have about 140 ASM specifications encoded in AsmetaL and available in [10]. We are strongly confident that AsmetaL satisfies all the desired requirements of expressivity, abstractness and easiness of use.

Note that concrete notations derived from metamodels can be also graphical. For instance, the Eclipse Graphical Modeling Framework (GMF) [4] provides a generative component and runtime infrastructure for developing graphical editors based on Eclipse Modelling Framework (EMF) [11] and the eclipse Graphical Editing Framework (GEF). The GMF follows a novel approach which suggests to derive modelling tools, like graphical model editors, from metamodels [35].

## R.6 To Allow Tool Interoperability

The existing ASM tools for model validation and verification, have been developed by encoding an ASM formal model into the language of the implementation environment and exploiting the computation engine and validation/verification algorithms and techniques of the implementation system to compute ASM runs and prove properties. Since each tool usually covers well only one aspect of the whole system development process, at different steps modelers and practitioners

---

<sup>3</sup> A preliminary version of the AsmetaL language can be found in [41], under the name of AsmM-CS (AsmM Concrete Syntax).

would like to switch tools to make the best of them while reusing information already entered about their models. However, ASM tools are loosely coupled and have syntaxes strictly depending on the implementation environment. This makes the integration of tools hard to accomplish and prevents ASMs from being used in an efficient and tool supported manner during the software development life-cycle. Therefore, a way to support tools interoperability is of great interest for the ASM community and can be achieved by the combination of standards like MOF, XMI (R.3), and JMIs (R.4).

Basically, all ASM artifacts/tools can be classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying to the AsmM metamodel appropriate projections from MOF to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the AsmM metamodel and related derivatives. Finally, integrated artifacts/tools are external and existing tools that are connected to the ASM metamodeling environment.

Fig. 5 shows a scenario of interoperability among ASM tools as suggested by our approach. Generated/based tools (like Tool A in the figure) can access ASM models through the APIs (like the AsmM JMIs) in a MOF repository (like the SUN MDR [5]) where ASM models reside. They can also exchange ASM models in the XML/XMI standard format: a XMI reader and writer provided by MDR can be used to load/save an ASM model from/into a XML file.

Integrated tools can interoperate in different ways. Some tools (like Tool B in the figure) can exchange ASM models in the XML/XMI standard format and verify their validity with respect to the given AsmM XMI DTD. Tool providers only need supply their tools with appropriate plug-ins capable of importing and/or exporting the XMI format for the AsmM (by using XMI

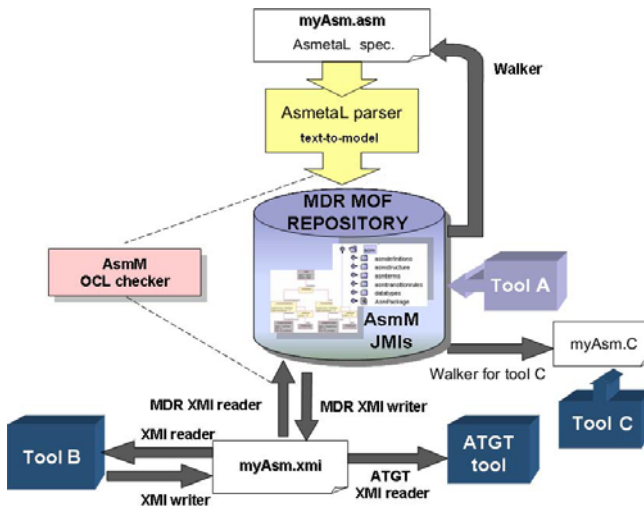


Fig. 5. ASM model interchange through XMI and APIs



readers/writers). Other tools (like Tool C in the figure) may keep their input data formats: in this case walkers must be developed to translate ASM models from the repository to the tool proprietary formats. Mixed approaches are also possible, as the one adopted in modifying the ATGT tool, as explained in Section [R.7](#).

A modeler can also start writing her/his ASM specification in AsmetaL and then, through the connection to the repository provided by the parser, transform it, for example, into the XMI interchange format.

## R.7 To Help the Integration of Existing Tools

We here discuss how we modified the ATGT tool [\[11\]](#) in order to make it AsmM-compliant. ATGT takes an ASM specification (written using the AsmGofer syntax) and produces a set of test predicates, translates the original ASM specification to Promela (the language of the model checker SPIN used to generate tests), and generates a set of test sequences by exploiting the counter example generation of the model checker.

ATGT is written in Java. It already (see Fig. [6](#)) has its own parser for AsmGofer files, which reads a specification and builds an internal representation of the model in terms of Java objects. The tool functionalities are delegated to three components (Test predicate generator, Tests generator, ASM to Promela) which read the data of the loaded ASM specification and perform their tasks.

In our approach, ATGT keeps its own data structures to represent the ASM models and other information necessary for the services it provides. In this way we do not modify the three most critical components, which continue to process data in the old representation.

To make ATGT capable of reading AsmM models, we first added a new component, the JMI/XMI reader, which is automatically derived from the metamodel by using MDR Netbeans. This JMI/XMI reader parses a XMI file containing the ASM specification the user wants to load and produces the JMI objects representing the loaded ASM. Then we added a module, called JMI queries, which queries those JMI objects and builds the equivalent model in terms of ATGT internal data. The JMI queries are very similar to the AsmGofer parser already

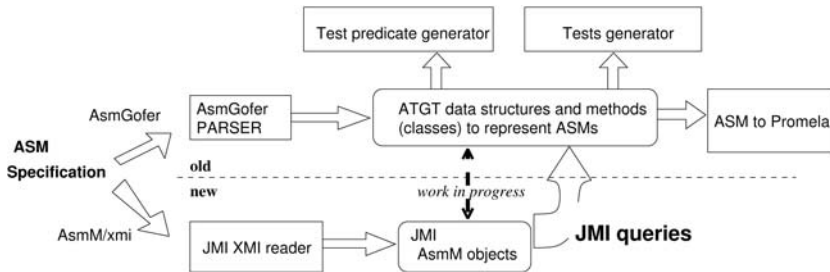


Fig. 6. Adapting ATGT to the AsmM



in ATGT, except that they read the information about the ASM model from JMI data instead of a file.

Although we did not exploit the power of the metamodel inside ATGT and we simply made ATGT AsmM-compliant, the result is worthwhile and the effort is limited: adding this new feature to ATGT required about two man-months. If we started today from scratch to develop ATGT, we would use directly JMI to represent ASM models, since JMI offers a stable and clean interface that is derived from the metamodel (see Sect. [R.4](#)). The use of JMI would avoid the burden of writing internal libraries for representing ASM models. For this reason, we have started working on making the internal representation of ASM models that ATGT adopts equivalent to the JMI, in order to eventually integrate JMI directly in ATGT (work in progress in Fig. [6](#)).

Further advances in the MDE direction [\[13\]](#) would be replacing the ASM to Promela and the AsmGofer parser components by model transformations from the AsmM (as *pivot metamodel*, see Sect. [R.9](#)) to Promela metamodel and from Gofer metamodel to the AsmM, provided that such metamodels for Promela and Gofer (linked to their concrete syntax) exist.

## R.8 To Help the Development of New Tools

MDE helps developers to build new tools by providing an interchange format [\(R.3\)](#), standard libraries [\(R.4\)](#) and several possible maps to concrete syntaxes and parsers [\(R.5\)](#). By exploiting these technologies, a developer who is interested in developing a new tool for ASMs, does not need to write a parser, an internal representation of ASMs and an interchange format (if he/she wishes to export, import files from other tools). In particular, the development of a grammar can be very time consuming and error prone - specially if one wants to be able to read complete ASM specifications. Internal representations of ASMs are normally bound to the parser which is being defined, and a small change in the parser may require an update of the internal libraries and refactoring of the code. All these tasks can require a good deal of time and effort, although they are not relevant for the particular technique or algorithm being developed. In the MDE approach, the developer needs to understand the metamodel (for example by reading its graphical representation - [R.2](#)) and then focus on the functionalities he/she intend to support with the new tool.

For instance, we have developed a general-purpose ASM *simulation engine* [\[29,10\]](#), called AsmetaS, to make AsmM models executable. This tool is an example of Tool A (see Fig. [5](#)) since essentially it is an interpreter which navigates through the MOF repository where ASM models are instantiated (as instances of the AsmM metamodel) to make its computations. We do not have to deal with basic functionalities such as parsing, abstract syntax trees, type checking, etc., since they are already provided by the MOF-environment. We have focused the development on those classes necessary to simulate an ASM, and the construction of the update set has required only the definition of the class UpdateSet representing an update set and the class UpdateSetBuilder building an update

set. `UpdateSetBuilder` introduces a method `UpdateSet m(R r)` which, for every class `R` representing a rule, builds the update set for the rule `r` of class `R`.

The architecture of this interpreter is very simple and consists in only 20 classes. It also allows a modular and incremental development. A first prototype (available at [10]) has been developed in only three man months and is able to interpret basic, turbo without submachine calls, and synchronous multiagent ASMs.

## R.9 To Integrate ASMs with Other Notations/Tools

In the MDE direction, `AsmM` can be seen as the *pivot metamodel* toward a systematic integration among ASM tools and between ASMs and external tools.

According to the view presented in [13], a pivot metamodel of a given formalism or language  $L$  is intended as a platform-independent modelling language which abstracts a certain number of general concepts about  $L$ . The integration among tools supporting  $L$  can be achieved by providing, for the notation  $L'$  (a dialect of  $L$ ) of each tool  $T_L$ , a *metamodel* – seen as a platform-specific modelling language – and *model transformations* to the pivot and from the pivot to the  $L'$ -metamodel. Hence, the metamodel of the notation  $L^i$  of a tool  $T_L^i$  can be linked to the metamodel of the notation  $L^j$  of another tool  $T_L^j$  by the composition of the two transformations from  $L^i$ -metamodel to the pivot and from the pivot to the  $L^j$ -metamodel. In this way, the interoperability between tools  $T_L^i$  and  $T_L^j$  is achieved by translating PSM (Platform-specific Model) models written in  $L^i$  to  $L^j$  and vice versa.

In the ASM context, `AsmM` can be adopted as pivot metamodel and would allow the integration among ASM tools at the level of metamodels. For example, if we take `AsmGofer` as tool  $T_L^i$  and `AsmL` as tool  $T_L^j$  and we had defined the corresponding metamodels together with precise *transformation bridges* from/to `AsmM`, we may map an `AsmGofer`-PSM into an `AsmL`-PSM.

Similarly, one can integrate the language  $L$  or one of its tools  $T_L$  with a tool using a notation  $M$  by providing a bridge between the pivot metamodel of  $L$  to the metamodel of  $M$ . In the ASM context, the `AsmM` may allow the integration between ASMs and tools like the model checkers `Spin` or `SMV`, provided that the metamodels for their notations exist.

For most notations  $M$ , however, the metamodel does not exist, and  $M$  is simply pure text. In this case the bridge must be built between the metamodel of  $L$  and a textual notation, and this can be done by using MOF-to-grammar tools, like `xText` [22] of `OpenArchitectureWare` or `TCS` of `AMMA` [3]. In the ASM context, we may “compile” ASM models into a programming language, like Java, by applying a `AsmM`-to-Java transformation to the input ASM model.

## R.10 To Complement the MDE with a Formal Approach

In the previous sections, we have discussed some advantages that ASMs can gain from MDE. We believe that the MDE paradigm can also gain rigor and

preciseness from the integration with ASMs as formal method. The semantics specification of domain-specific modeling languages (defined in terms of a meta-model), for example, is an open problem in the MDE approach. The OMG meta-modelling framework provides, by means of *metamodels* and *UML profiles* (UML metamodel extensions for a particular application domain), standard techniques to define the abstract syntax and static semantics (the OCL constraints) of a Domain-specific language. However, it lacks of any standard and rigorous support to provide the dynamic (operational) semantics, which is usually given in natural language. This lack has several negative consequences, as confirmed by existing work in literature which aims at formalizing the UML semantics.

Techniques and approaches to the precise and pragmatic definition of behavioral semantics for domain-specific languages are still under development. One promising method, called *semantic anchoring* relies on the use of well-defined *semantic units* of simple, well-understood constructs (like a finite state machine) and on the use of model transformations that map higher level modeling constructs into configured semantic units. This approach has been followed, for example, by the authors in [16,46], where AsmL is used as a common semantic framework to define the semantic domain of Domain-specific languages.

We believe that any formalism proposed as semantic framework must address the following important characteristics: (i) it should be formal and powerful enough to rigorously define the operational semantics of complex real languages, (ii) it should be executable in order to validate the metamodels' semantics, (iii) it should be endowed with a metamodel-based definition conforming to the meta-modelling framework in order to allow the applicability of model transformation tools, and (iv) it should be able to work at high levels of abstraction. According to these requirements, the ASM formalism seems to be a good candidate.

Similarly to the approach in [16], we propose ASMs as semantic framework to define the (operational) semantics of metamodel-based languages. The key idea is a smooth integration of the AsmM metamodel with the OMG framework in order to provide a means to rigorously define the operational semantics of metamodel-based languages and, in particular, of UML extensions (profiles), in a way which permits us to uniformly link abstract syntax, expressed in the MOF metalanguage, and detailed semantics, expressed in ASMs (here promoted as metalanguage, too) of languages.

In practice, this integration may be done as shown in Fig. 7. At the meta-model level, the MOF core constructs i.e., the Infrastructure Library, have to be mapped into ASM concepts. This may be done by defining a set of transformation rules,  $T_{\text{MOFToASM}}$ , from the Infrastructure Library metamodel to the AsmM metamodel.

At the metamodel level, a metamodel or a UML profile  $L_{\text{MOF}}$  of a given language  $L$  is translated by  $T_{\text{MOFToASM}}$  to a ground ASM-compliant metamodel  $L_{\text{AsmM}}^1$  of  $L$  made of multi-sorted first-order structures, i.e. sets with relations, functions and constraints, representing classes and associations of the source  $L_{\text{MOF}}$  metamodel.  $L_{\text{AsmM}}^1$  needs to be complemented with the semantic aspects of the language  $L$ . The *computational model* which reflects the operational

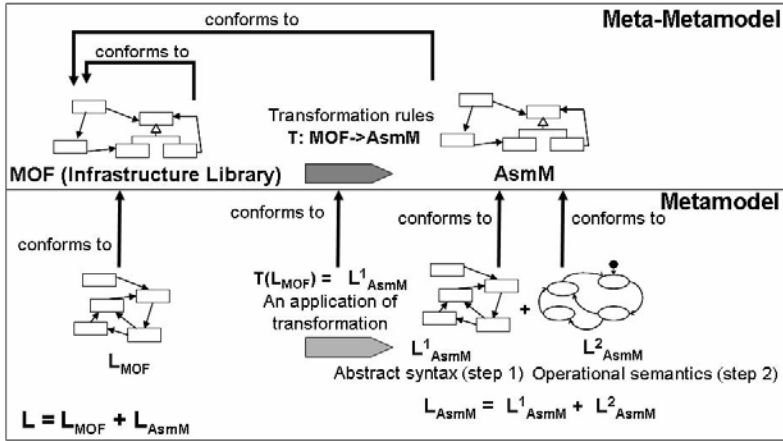


Fig. 7. An integrated framework for metamodel-based language specification

semantics of  $L$ , say  $L_{AsmM}^2$ , is defined through ASM transition rules. The static structures of the ASM signature  $L_{AsmM}^1$  is further refined and enriched with dynamic aspects, e.g., designating some specific entities to be ASM agents, and introducing new functions, which, however, may be present in the original metamodel expressed in terms of OCL query/operations. We say  $L_{AsmM} = L_{AsmM}^1 + L_{AsmM}^2$  the final result of this modelling activity.

Note that, the process of applying the  $T_{MOFToASM}$  can be fully automatized by means of a *transformation engine* like the ATL in the AMMA platform [3], Xactium XMF Mosaic [6], etc. However, a certain human effort is still required to capture in terms of ASM transition rules the behavioural aspects of the given language.

We have applied the proposed methodology to a UML profile for the SystemC language [39] - as part of the definition of a model-based SoC (System-on-chip) design flow for embedded systems [21,40] - to define the operational semantics of the SystemC Process State Machines, an extension of the UML statecharts used to model the reactive behaviour of the SystemC processes.

Although the proposed approach has been first identified and tested for the OMG's framework, it could be easily extended and applied to other metamodeling frameworks.

## Related Work

Concerning the definition of a concrete language for ASMs, other previous proposals exist. The Abstract State Machine Language (AsmL) [8] developed by the Foundation Software Engineering group at Microsoft is the greatest effort in this respect. AsmL is a rich executable specification language, based on the theory of ASMs, expression- and object- oriented, and fully integrated into the

.NET framework and Microsoft development tools. However, AsmL does not provide a semantic structure targeted for the ASM method. “One can see it as a fusion of the Abstract State Machine paradigm and the .NET type system, influenced to an extent by other specification languages like VDM or Z” [52]. Adopting a terminology currently used, AsmL is a platform-specific modeling language for the .NET type system. A similar consideration can be made also for the AsmGofer language [43]. An AsmGofer specification can be thought, in fact, as a PSM (platform-specific model) for the Gofer environment.

Other specific languages for the ASMs, no longer maintained, are ASM-SL [15], which adopts a functional style being developed in ML and which has inspired us in the language of terms, and XASM [50] which is integrated in Montages, an environment generally used for defining semantics and grammar of programming languages. Recently other simulation environments for ASMs have been developed, including the CoreAsm [18], an extensible execution engine developed in Java, and TASM (Timed ASMs) [45], an encoding of Timed Automata in ASMs.

Concerning the metamodeling technique for language engineering, we can mention the official metamodels supported by the OMG [37] for MOF itself, for UML [47], for OCL, etc. Academic communities like the Graph Transformation community [30,44,48] and the Petri Net community [38], have also started to settle their tools on general metamodels and XML-based formats. A metamodel for the ITU language SDL-2000 has been also developed [24]. Recently, a metamodel for the AsmL language is available in the XMI format at [9] as part of a zoo of metamodels defined by using the KM3 meta-language [33]. However, this metamodel is not appropriately documented or described elsewhere, so this prevent us to evaluate it for our purposes.

## References

1. Eclipse Modeling Framework, <http://www.eclipse.org/emf/>
2. Java Compiler Compiler, <https://javacc.dev.java.net/>
3. The AMMA Platform, <http://www.sciences.univ-nantes.fr/lina/at1/>
4. The Eclipse Graphical Modeling Framework, <http://www.eclipse.org/gmf/>
5. The, M.D.R. (Model Driven Repository) for NetBeans, <http://mdr.netbeans.org/>
6. The Xactium XMF Mosaic, <http://www.modelbased.net/www.xactium.com/>
7. Anlauff, M., Del Castillo, G., Huggins, J., Janneck, J., Schmid, J., Schulte, W.: The ASM-Interchange Format XML Document Type Definition (ASM-DTD), <http://www.first.gmd.de/~ma/asmdtd.html>
8. The ASML Language, <http://research.microsoft.com/foundations/AsmL/>
9. The AsmL metamodel in the Atlantic Zoo (2006), <http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/#AsmL>
10. The Abstract State Machine Metamodel and its tool set, <http://asmeta.sf.net/>
11. ATGT: ASM tests generation tool, <http://cs.unibg.it/gargantini/project/atgt/>
12. Bézivin, J.: On the Unification Power of Models. Software and System Modeling (SoSym) 4(2), 171–188 (2005)

13. Bézivin, J., Brunelière, H., Jouault, F.J., Kurtev, I.: Model Engineering Support for Tool Interoperability. In: The 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica (2005)
14. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
15. Del Castillo, G.: The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models Tool Demonstration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 578–581. Springer, Heidelberg (2001)
16. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
17. OMG, CORBA, <http://www.corba.org/>
18. The CoreASM Project, <http://www.coreasm.org/>
19. Hearnden, D., Raymond, K., Steel, J.: Anti-Yacc: MOF-to-text. In: Proc. of EDOC, pp. 200–211 (2002)
20. Dold, A.: A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universitat Ulm (July 1998)
21. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A SoC Design Methodology Based on a UML 2.0 Profile for SystemC. In: Proc. of Design Automation and Test in Europe (DATE 2005). IEEE, Los Alamitos (2005)
22. Efftinge, S.: oAW xText - A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit (2006)
23. Mens, T., et al.: Challenges in software evolution. In: International Workshop on Principles of Software Evolution, IWPSSE 2005 (2005)
24. Fischer, J., Piefel, M., Scheidgen, M.: A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In: Fourth SDL And MSC Workshop (SAM 2004), pp. 208–223 (2004)
25. Gargantini, A., Riccobene, E.: Encoding Abstract State Machines in PVS. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 303–322. Springer, Heidelberg (2000)
26. 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)
27. Gargantini, A., Riccobene, E., Scandurra, P.: Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In: 3M4MDA 2006 workshop at the European Conference on MDA (2006)
28. Gargantini, A., Riccobene, E., Scandurra, P.: Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan (2006)
29. Gargantini, A., Riccobene, E., Scandurra, P.: A Metamodel-based Simulator for ASMs. In: 14th International ASM Workshop, Grimstad, Norway, June 7-9 (2007)
30. Holt, R., Schürr, A., Sim, S.E., Winter, A.: Graph eXchange Language, <http://www.gupro.de/GXL/index.html>
31. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01, <http://www.uml.org/>
32. Java Metadata Interface Specification, Version 1.0. (2002), <http://java.sun.com/products/jmi/>
33. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)

34. OMG. The Model Driven Architecture (MDA), <http://www.omg.org/mda/>
35. Nyttun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 268–283. Springer, Heidelberg (2006)
36. OCL Environment (OCLE), <http://lci.cs.ubbcluj.ro/ocle>
37. The Object Management Group (OMG), <http://www.omg.org>
38. Petri Net Markup Language (PNML), <http://www.informatik.hu-berlin.de/top/pnml>
39. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A UML 2.0 profile for SystemC: toward high-level SoC design. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 138–141. ACM, New York (2005)
40. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A model-driven design environment for embedded systems. In: Proc. of the 43rd annual Conference on Design Automation (DAC 2006), pp. 915–918. ACM Press, New York (2006)
41. Scandurra, P., Gargantini, A., Genovese, C., Genovese, T., Riccobene, E.: A Concrete Syntax derived from the Abstract State Machine Metamodel. In: 12th International Workshop on Abstract State Machines (ASM 2005), Paris, France, March 8-11 (2005)
42. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science* 3(4), 377–413 (1997)
43. Schmid, J.: *AsmGofer*, <http://www.tydo.de/AsmGofer>
44. Taentzer, G.: Towards common exchange formats for graphs and graph transformation systems. In: Padberg, J. (ed.) UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS (2001)
45. The Timed Abstract State Machine (TASM) Language and Toolset, <http://esl.mit.edu/html/tasm.html>
46. Thibodeaux, R.: The Specification of Architectural Languages with Abstract State Machines. In: 14th International ASM Workshop, Grimstad, Norway, June 7-9 (2007)
47. OMG. The Unified Modeling Language (UML), <http://www.uml.org>
48. Varró, D., Varró, G., Pataricza, A.: Towards an XMI-based model interchange format for graph transformation systems. Technical report, Budapest University of Technology and Economics, Dept. of Measurement and Information Systems (September 2000)
49. Winter, K.: Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)* 3(5), 689–701 (1997)
50. XASM: The Open Source ASM Language, <http://www.xasm.org>
51. OMG, XMI Specification, v1.2, <http://www.omg.org/cgi-bin/doc?formal/2002-01-01>
52. Gurevich, Y., Rossman, B., Schulte, W.: Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27 (March 2004)



# An ASM-Characterization of a Class of Distributed Algorithms

Andreas Glausch and Wolfgang Reisig

Humboldt-Universität zu Berlin

Institut für Informatik

{glausch,reisig}@informatik.hu-berlin.de

**Abstract.** Conventional computation models restrict to particular data structures to represent states of a computation, e.g. natural numbers, sequences, stacks, etc. Gurevich's *Abstract State Machines (ASMs)* take a more liberal position: *any* first-order structure may serve as a state. In [7] Gurevich characterizes the expressive power of *sequential ASMs*: he defines the class of *sequential algorithms* by means of only a few, amazingly general requirements and proves this class to be equivalent to sequential ASMs.

In this paper we generalize Gurevich's result to *distributed algorithms*: we define a class of distributed algorithms by likewise general requirements and show that this class is covered by a distributed computation model based on sequential ASMs.

## 1 Introduction

Conventional computation models include a distinguished notion of *states*. For example, a state of a Turing machine is captured by the head position, the internal control state, and the tape inscription. Another example is the  $\lambda$ -calculus where each state is represented by a  $\lambda$ -expression. In both cases the representation of states is based on a restricted set of data structures: alphabets, natural numbers, and sequences over alphabets.

Gurevich's Abstract State Machines (ASMs) [6] feature a more liberal representation of states: each state is a *structure*, a notion well known from first-order logic. As usual, a structure comprises a nonempty set  $U$  (its *universe*) together with finitely many functions defined over  $U$ , each with a fixed arity. No additional properties are required: *any* structure may serve as a state of an ASM. For example, a state may include uncomputable functions or real-valued functions such as *sin* and *log*.

As a consequence, a computation of an ASM may *a priori* employ mathematical concepts such as real numbers, vectors, graphs, geometrical objects, etc. (Of course, this also holds for classical data structures such as stacks, lists, and queues.) Conventional computation models such as Turing machines usually require a particular encoding in order to represent such concepts, if possible at all.



The freedom to include arbitrary data structures as part of the state of an ASM allows for a natural and flexible modeling of the states of a system. For this reason ASMs have been extended to a successfully applied design and analysis methodology [4]. By use of stepwise refinement and composition of ASMs, large-scale systems from real-world have been modeled and analyzed formally. For example, the operational semantics of the SDL-2000 standard officially is defined by an ASM model [9], and the correctness and completeness of a Java standard compiler has been proven based on a formal ASM model [13].

## 2 Scope and Contribution of This Paper

Classically, ASMs employ a simple pseudo-code like syntax to describe state changes [6]. Gurevich revealed that ASMs may also be characterized independently of any concrete syntax: In [7] Gurevich defines the class of *sequential algorithms* and proves this class to be equivalent to the class of *sequential ASMs* (c.f. also [12]). Blass, Gurevich, and others identified further, more general classes of algorithms – including *nondeterministic*, *parallel*, and *interactive* versions – and showed that each class is equivalent to a corresponding class of ASMs [11, 2, 3, 8].

In this paper we contribute to this work by a corresponding result for *distributed algorithms*: we define a class of distributed algorithms by five simple and general requirements and show that this class is equivalent to a distributed computation model based on ASMs.

Distributed algorithms significantly differ from the aforementioned variants of algorithms: a sequential/nondeterministic/parallel/interactive algorithm computes for each state  $S$  a successor state  $S'$ , i.e. state changes occur sequentially ordered and may involve the complete state  $S$ . By contrast, a distributed algorithm performs *concurrent* and *locally bounded* changes of the state. In order to represent such state changes we adapt the idea of *actions* with locally limited cause and effect, and the notion of *distributed runs*, in the tradition of Petri [10], Pratt [14], and Gurevich [6]: a distributed run is a set of action occurrences, partially ordered by causal dependencies.

The variant of distributed algorithms examined in this paper still has some limitations. For the sake of simplicity, we restrict attention to communication via a shared state, and do not consider message passing explicitly. Furthermore, we do not consider dynamic instantiation or disposing of agents. The main contribution of this paper is a first step towards an ASM-based theory of distributed algorithms, leaving much room for generalizations.

The rest of this paper is organized as follows: Section 3 recalls elementary notions on structures and introduces the general framework of *actions* and *distributed runs*. Section 4 defines the notion of *distributed algorithms*. Section 5 introduces *distributed ASMs*, an operational computation model for distributed algorithms. Finally, we show in Sect. 6 that distributed ASMs are expressive enough to capture every distributed algorithm as discussed in Sect. 4.

### 3 The Basic Framework

In this section we motivate and exemplify the basic notions employed in the rest of this paper. We recall some basic notions on structures, and introduce the concept of actions and distributed runs.

As usual, a *signature*  $\Sigma$  is used to address the functions of a structure:  $\Sigma$  consists of finitely many functions symbols  $\mathbf{f}_1, \dots, \mathbf{f}_k$  and determines for each  $\mathbf{f}_i$  an arity  $n_i \in \mathbb{N}$ . A structure  $S$  consisting of the functions  $f_1, \dots, f_n$  is a  $\Sigma$ -*structure* if the arity of  $f_i$  is  $n_i$  for  $i = 1, \dots, n$ , i.e. the arities of the symbols in  $\Sigma$  and the arities of the functions in  $S$  coincide. In this case the function  $f_i$  is the *interpretation* of the symbol  $\mathbf{f}_i$  in  $S$ , denoted by  $\mathbf{f}_{iS}$ .

As already indicated in the introduction, each state of an ASM is a structure. We therefore use the notions structure and state interchangeably. As a running example throughout this paper we consider the following structure  $Q$ , with universe  $U = \{1, 2, 3\}$ , consisting of two 0-ary functions  $\mathbf{a}_Q$  and  $\mathbf{b}_Q$ , and two unary functions  $\mathbf{inc}_Q$  and  $\mathbf{val}_Q$ :

$$\begin{array}{llll} \mathbf{a}_Q = 1 & \mathbf{b}_Q = 2 & \mathbf{inc}_Q(1) = 2 & \mathbf{val}_Q(1) = 1 \\ & & \mathbf{inc}_Q(2) = 3 & \mathbf{val}_Q(2) = 2 \\ & & \mathbf{inc}_Q(3) = 1 & \mathbf{val}_Q(3) = 3. \end{array}$$

We represent a signature by a sequence of function symbols followed by a sequence of their respective arities. Obviously, the signature of  $Q$  is  $\Sigma_Q = (\mathbf{a}, \mathbf{b}, \mathbf{inc}, \mathbf{val}, 0, 0, 1, 1)$ .

In order to represent distributed computation on a  $\Sigma$ -structure  $S$ , we require means to describe locally bounded and concurrent changes of  $S$ . It turns out useful not to consider  $S$  as a monolithic entity but as a set of  $\Sigma$ -*molecules*, each of which consisting of a location and a value. A *location* of  $S$  consists of a  $n$ -ary function symbol  $\mathbf{f}$  and a  $n$ -ary argument tuple  $\bar{a}$  over the universe of  $S$ . For example  $(\mathbf{val}, [1])$  is a location of the structure  $Q$  (we enclose the argument tuple in square brackets for the sake of readability). Obviously, each location  $(\mathbf{f}, \bar{a})$  of  $S$  defines a unique value  $v = \mathbf{f}_S(\bar{a})$ . The triple  $(\mathbf{f}, \bar{a}, v)$  is a  $\Sigma$ -*molecule* of  $S$  (or simply *molecule* if this causes no confusion). For example,  $(\mathbf{val}, [1], 1)$  is a molecule of the structure  $Q$ . Intuitively, this molecule states that “the function denoted by  $\mathbf{val}$  maps the argument tuple  $[1]$  to the value 1”.

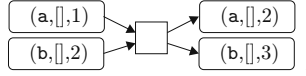
A structure  $S$  is completely described by its set of molecules. For example, the above structure  $Q$  is represented by the following set of molecules:

$$\begin{aligned} Q = \{ & (\mathbf{a}, [], 1), (\mathbf{b}, [], 2), \\ & (\mathbf{inc}, [1], 2), (\mathbf{inc}, [2], 3), (\mathbf{inc}, [3], 1), \\ & (\mathbf{val}, [1], 1), (\mathbf{val}, [2], 2), (\mathbf{val}, [3], 3) \}. \end{aligned}$$

Calling them “updates”, Gurevich employed molecules already in [6] to describe differences between structures.

A structure  $S$  is changed locally by applying an *action* which replaces some of the molecules of  $S$ . For example, the above structure  $Q$  can be changed by replacing the molecules  $(\mathbf{a}, [], 1)$  and  $(\mathbf{b}, [], 2)$  by the molecules  $(\mathbf{a}, [], 2)$  and  $(\mathbf{b}, [], 3)$ . This

yields another structure  $Q'$ , interpreting the symbols  $a$  and  $b$  by 2 and 3, respectively. We apply the graphical notation of Petri nets and outline this action by



In general, a  $\Sigma$ -action (or simply *action* if  $\Sigma$  is clear from the context) is a pair  $a = (a^{\text{in}}, a^{\text{out}})$ , where  $a^{\text{in}}$  and  $a^{\text{out}}$  are sets of  $\Sigma$ -molecules such that

- the locations of the molecules in  $a^{\text{in}}$  and  $a^{\text{out}}$  coincide, (1)
- both  $a^{\text{in}}$  and  $a^{\text{out}}$  are *consistent*: a set of molecules is consistent if it does not contain two different molecules with the same location. (2)

Hence, an action  $a$  only modifies the values of the molecules, but not their locations (1), and both  $a^{\text{in}}$  and  $a^{\text{out}}$  determine at most one value for each location (2).

An action  $a$  then performs a *step*  $S \xrightarrow{a} S'$  by replacing the molecules  $a^{\text{in}}$  by the molecules  $a^{\text{out}}$ , thus changing the state  $S$  to a new state  $S'$ . More precisely, for two states  $S$  and  $S'$  we write  $S \xrightarrow{a} S'$  to denote that

- $a^{\text{in}} \subseteq S$ , (3)
- $S' = (S \setminus a^{\text{in}}) \cup a^{\text{out}}$ , (4)
- $S$  and  $S'$  have the same universe. (5)

That is,  $a^{\text{in}}$  constitutes the *local cause* of  $a$  (3), whereas  $a^{\text{out}}$  constitutes the *local effect* of  $a$  (4). Furthermore, actions are required to preserve the universe of the state (5).

Two actions replacing disjoint molecules do not interfere with each other, and may therefore be applied concurrently. This implies the notion of *distributed run*: a distributed run is a partially ordered set of action occurrences. We represent this partial order by an inscribed *occurrence net*, a notion well known from the theory of Petri nets (see (11)). Figure 1 outlines an example of a distributed run  $R$ : each square (called *transition*) represents the occurrence of one action

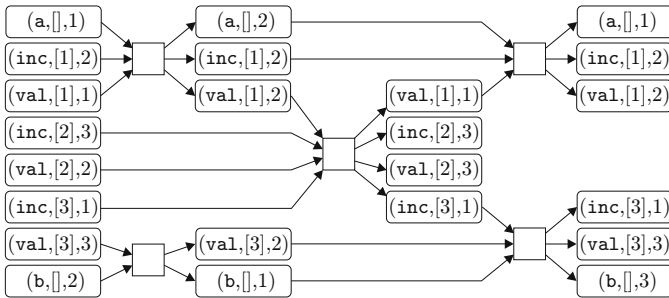


Fig. 1. The distributed run  $R$

replacing the molecules attached to the incoming arcs by the molecules attached to the outgoing arcs. An important property of occurrence nets is that each molecule is attached to at most one incoming and at most one outgoing arc. Consequently, each molecule is accessed resp. modified by at most one action, i.e. molecules cannot be accessed or modified concurrently. The transitive closure of the arcs then induces a partial order on the transitions, i.e. a partial order on the action occurrences.

In Fig. 1, the rounded rectangles inscribed by the molecules are called *places*. Observe that the leftmost places in  $R$  hold the molecules of the structure  $Q$ , which is the *initial state* of this run. Further (global) states of this run are identified by the notion of *cut*. A cut  $C$  is a maximal set of unordered places of  $R$  such that only finitely many transitions precede the places in  $C$ . Figure 2 shows three different cuts,  $C_1$ ,  $C_2$ , and  $C_3$ , each of which represented by a dashed line. The inscriptions of the places in a cut  $C$  then always constitute a  $\Sigma$ -structure  $S$ , as exemplified in Fig. 2. This structure  $S$  is the global state of the run after the occurrence of all actions preceding the places in  $C$ . Note that the cuts  $C_1$  and  $C_2$  in Fig. 2 represent a *step*, as only a single action occurs between  $C_1$  and  $C_2$ .

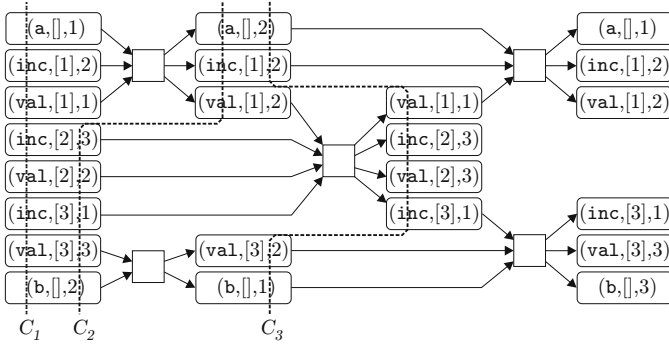


Fig. 2. Three different cuts  $C_1$ ,  $C_2$ , and  $C_3$  of the distributed run  $R$

In general, a *distributed run* over a signature  $\Sigma$  is an occurrence net with places inscribed by molecules such that

- the molecules at the places without an incoming arc constitute a  $\Sigma$ -structure  $S_0$ , (the *initial state* of the run)
- all molecules contain only elements of the universe of  $S_0$ ,
- each transition  $t$  represents an action  $a$ : the places at the incoming/outgoing arcs of  $t$  are inscribed by the molecules in  $a^{\text{in}}/a^{\text{out}}$ .

Due to lack of space, we skip the (somewhat technical) formal definition of distributed runs and cuts, and rely on their intuitive graphical notation as shown in Fig. 1. For the technical details, we refer to [5].

## 4 Distributed Algorithms

Based on the framework introduced in the previous section, this section defines the class of *distributed algorithms* by five requirements which are fairly general, nevertheless simple and intuitive. We briefly discuss the reasonability for each of these requirements. In addition, for the ASM expert, we briefly present for each requirement its relationship to Gurevich's original requirements to sequential algorithms [7].

### 4.1 A State of a Distributed Algorithm Is a Structure

As emphasized by Tarski already, mathematical structures are general enough to faithfully describe any static mathematical entity on any level of abstraction. Consequently, it is legitimate to assume that every state of an algorithm can also be described naturally by a structure. As an algorithm should always have a finite representation, a single signature (with a finite set of symbols) suffices for all states. Furthermore, in a computational framework, it is common to mark some of the states as initial. This leads to the following first requirement:

**Requirement D1 (state requirement).** *A distributed algorithm  $\mathcal{D}$  determines a non-empty set  $\mathcal{S}_{\mathcal{D}}$  of states and a non-empty set  $\mathcal{I}_{\mathcal{D}} \subseteq \mathcal{S}_{\mathcal{D}}$  of initial states. All states in  $\mathcal{S}_{\mathcal{D}}$  are structures over the same signature  $\Sigma_{\mathcal{D}}$ .*

This requirement reflects Gurevich's *abstract state* requirement, where each state of a sequential algorithm is required to be a  $\Sigma$ -structure.

### 4.2 Distributed Algorithms Perform Actions

According to common intuition about distributed computing, state changes in a distributed algorithm occur locally bounded and occasionally concurrent. The concept of *actions* as introduced in Sect. 3 captures such state changes in a natural and general way. Therefore, the state changes of a distributed algorithm  $\mathcal{D}$  can always be described naturally by a set  $\mathcal{A}_{\mathcal{D}}$  of actions.

Of course, applying an action of  $\mathcal{D}$  to a state of  $\mathcal{D}$  should always yield a state of  $\mathcal{D}$  again. Furthermore, for technical reasons, we require every action of  $\mathcal{D}$  be executable in at least one state of  $\mathcal{D}$ .

**Requirement D2 (action requirement).** *A distributed algorithm  $\mathcal{D}$  determines a set  $\mathcal{A}_{\mathcal{D}}$  of actions over signature  $\Sigma_{\mathcal{D}}$  such that for each action  $a \in \mathcal{A}_{\mathcal{D}}$  holds:*

- if  $S \in \mathcal{S}_{\mathcal{D}}$  and  $S \xrightarrow{a} S'$  then  $S' \in \mathcal{S}_{\mathcal{D}}$ ,
- there is a state  $S \in \mathcal{S}_{\mathcal{D}}$  such that  $S \xrightarrow{a} S'$ .

For a state  $S \in \mathcal{S}_{\mathcal{D}}$  and an action  $a \in \mathcal{A}_{\mathcal{D}}$ ,  $S \xrightarrow{a} S'$  is a *step* of  $\mathcal{D}$ . A *distributed run* of  $\mathcal{D}$  is a distributed run  $R$  (see Sect. 3) such that the initial state of  $R$  is an initial state of  $\mathcal{D}$  and each action occurring in  $R$  is an action of  $\mathcal{D}$ .

Requirement **D2** is an adaption of Gurevich’s *sequential time* requirement, which demands each sequential algorithm to determine a set of global steps, represented as a next-state function  $\tau$ . We replace global steps by local actions here.

### 4.3 Distributed Algorithms Respect Isomorphism

As usual, a bijective mapping  $i : U_R \rightarrow U_S$  between the universes of two  $\Sigma$ -structures  $R$  and  $S$  is an *isomorphism between  $R$  and  $S$*  (written  $i : R \rightarrow S$ ) iff  $i(\mathbf{f}_R(u_1, \dots, u_n)) = \mathbf{f}_S(i(u_1), \dots, i(u_n))$  for all  $n$ -ary function symbols  $\mathbf{f}$  in  $\Sigma$  and all  $u_1, \dots, u_n \in U_R$ . Isomorphic structures only differ in the concrete representation of the elements of the universe, whereas the functions of both structures are essentially the same.

For an algorithm the concrete representation of elements should be inessential. For example, the Euclidean algorithm computes the greatest common divisor regardless whether the integers are represented by transistor states on a chip or by ink on a paper. In general, a distributed algorithm should not distinguish isomorphic states, i.e. should behave “isomorphic” at isomorphic states.

More precisely, if an action  $a$  can occur in a state  $R$  isomorphic to a state  $S$ , a corresponding isomorphic action can occur in  $S$ . To formalize this, we extend any isomorphism  $i : R \rightarrow S$  canonically to molecules, sets of molecules, and actions. The third requirement then reads:

**Requirement D3 (isomorphism requirement).** *The sets of states  $\mathcal{S}_{\mathcal{D}}$  and  $\mathcal{J}_{\mathcal{D}}$  of a distributed algorithm  $\mathcal{D}$  both are closed under isomorphism. Furthermore, if  $R \xrightarrow{a} R'$  is a step of  $\mathcal{D}$  and  $i : R \rightarrow S$  is an isomorphism, then there exists also a step  $S \xrightarrow{i(a)} S'$  of  $\mathcal{D}$ .*

This requirement is an adoption of a part of Gurevich’s *abstract state* requirement which demands the next-state function  $\tau$  to map isomorphic states to isomorphic next-states.

### 4.4 Actions of Distributed Algorithms Operate Autonomously

The next requirement can be formulated simply and intuitively convincing: each action  $a$  of a distributed algorithm only uses elements that  $a$  has access to, i.e.  $a$  operates autonomously on data. In other words, an action of a distributed algorithm does not obtain elements from nowhere.

It remains to clarify the above meaning of *use* and *have access*. In technical terms, an action  $a$  uses an element  $x$  if  $x$  occurs in an argument tuple in one of the molecules of  $a$  (where it is used to access a location), or if  $x$  occurs as a value in one of the molecules in  $a^{\text{out}}$  (where it is used as a newly assigned value). As the locations of  $a^{\text{in}}$  and  $a^{\text{out}}$  are identical, we define:  $a$  uses  $x$  iff there is a molecule  $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{out}}$  such that  $x \in \{u_1, \dots, u_n, v\}$ .

Furthermore,  $a$  has access to the value of a molecule  $m \in a^{\text{in}}$  if  $a$  has access to the elements in the argument tuple of  $m$ . This leads to the following inductive definition:

- for each molecule  $(\mathbf{x}, [], v) \in a^{\text{in}}$ ,  $a$  has access to  $v$ ,
- if  $a$  has access to  $u_1, \dots, u_n$  and  $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{in}}$  for  $n \geq 1$ , then  $a$  has access to  $v$ .

The fourth requirement is now quite obvious:

**Requirement D4 (autonomicity requirement).** *For each action  $a$  of a distributed algorithm  $\mathcal{D}$ ,  $a$  has access to each element that is used by  $a$ .*

As [D4](#) relies heavily on the notion of action introduced in this paper, there is no direct counterpart in Gurevich’s requirements. However, [D4](#) can be seen as a part of the *bounded exploration* requirement where ground terms are used to characterize the next-state function of sequential algorithms: the inductive evaluation of ground terms corresponds to the above, inductive definition of “having access to”.

#### 4.5 Actions of Distributed Algorithms Are Bounded

A real-world processor (e.g., a computer, an organization, or a human being) executing an algorithm can obviously perform only a bounded amount of work in each step. For this reason it is quite natural to require the actions of a distributed algorithm to be bounded in size. This idea leads to the fifth and last requirement:

**Requirement D5 (bounded-actions requirement).** *For a distributed algorithm  $\mathcal{D}$  there exists a constant  $c \in \mathbb{N}$  such that for each action  $a$  of  $\mathcal{D}$  holds  $|a^{\text{in}}| \leq c$  (which is equivalent to  $|a^{\text{out}}| \leq c$ ).*

Similar to [D4](#), this requirement can be seen as a part of Gurevich’s *bounded exploration* requirement which demands the next-state function  $\tau$  to be characterized by a *finite* (hence, a bounded) set of ground terms.

We do not demand any further requirements: we call *any* entity satisfying the Requirements [D1](#)–[D5](#) a distributed algorithm.

## 5 Distributed Abstract State Machines

In the previous section we introduced the class of distributed algorithms in a purely semantical and declarative way. But usually algorithms are represented in an explicit and syntactical form, e.g. by program code or by natural language. This gives rise to the question whether a given distributed algorithm can be represented in a syntactical way at all. In this section we answer this questions positively by presenting the computation model of *distributed ASMs*, which is based on sequential ASMs [\[6\]](#).

The version of distributed ASMs we introduce here is a special case of the version presented in the Lipari Guide [\[6\]](#). There, a distributed ASM includes a set of *agents*, each of which is executing an *ASM program*. The agent set may grow and shrink during computation, thus allowing dynamic instantiation and disposing of agents. By contrast, we consider only a fixed set of agents,

where each agent is identified by the program it executes. Furthermore, in [6] Gurevich introduces a highly general notion of distributed run, which allows several agents to access the same location of a state concurrently. As discussed in Sect. 3, concurrent access is not possible in our version of distributed runs.

Despite those restrictions, our version of distributed ASMs is justified by the following fact: *Any* distributed algorithm as discussed in the previous section can be represented by one of our distributed ASMs. We will prove this fact in the upcoming Sect. 6.

## 5.1 Assignment Statements

As usual, a signature  $\Sigma$  yields  $\Sigma$ -terms: each 0-ary symbol from  $\Sigma$  is a  $\Sigma$ -term, and for an  $n$ -ary symbol  $\mathbf{f} \in \Sigma$  and given  $\Sigma$ -terms  $t_1, \dots, t_n$ , the symbol sequence  $\mathbf{f}(t_1, \dots, t_n)$  is a  $\Sigma$ -term, too. Such terms are *evaluated* by a  $\Sigma$ -structure  $S$  in the usual way: for each 0-ary symbol  $\mathbf{a}$ , the element  $\mathbf{a}_S$  denotes the *evaluation* of  $\mathbf{a}$  in  $S$ , and for a term  $\mathbf{f}(t_1, \dots, t_n)$ , its evaluation in  $S$  is defined inductively by  $\mathbf{f}(t_1, \dots, t_n)_S =_{\text{def}} \mathbf{f}_S(t_{1S}, \dots, t_{nS})$ .

Terms are used to form *assignment statements*. A simple example built from signature  $\Sigma_Q$  is the statement  $\text{val}(\mathbf{a}) := \text{inc}(\mathbf{b})$ . Executing this assignment statement in state  $Q$  updates the value of function  $\text{val}_Q$  at argument  $\mathbf{a}_Q = 1$  by the value of  $\text{inc}(\mathbf{b})_Q = 3$ , i.e. replaces the molecule  $(\text{val}, [1], 1)$  by the molecule  $(\text{val}, [1], 3)$ .

The general form of an assignment statement  $\alpha$  is  $t := t'$ , where  $t, t'$  are  $\Sigma$ -terms. Applied to a state  $S$ ,  $\alpha$  updates the location specified by  $t$  by the value of  $t'$ . More precisely, for  $t = \mathbf{f}(t_1, \dots, t_n)$ , the *location of  $t$  in  $S$*  is

$$\text{loc}_S(t) =_{\text{def}} (\mathbf{f}, [t_{1S}, \dots, t_{nS}]).$$

Then  $\alpha$  replaces the molecule set  $\alpha_S^{\text{old}}$  by the molecule set  $\alpha_S^{\text{new}}$ , with

$$\begin{aligned} \alpha_S^{\text{old}} &=_{\text{def}} \{ (\text{loc}_S(t), t_S) \} \\ \alpha_S^{\text{new}} &=_{\text{def}} \{ (\text{loc}_S(t), t'_S) \}. \end{aligned}$$

That is,  $\alpha_S^{\text{old}}$  and  $\alpha_S^{\text{new}}$  contain the modified molecule *before* and *after* executing the assignment statement  $\alpha$ , respectively.

## 5.2 Assignment Statements Generate Actions

Given an assignment statement  $\alpha$  and a corresponding state  $S$ , it is tempting to regard  $(\alpha_S^{\text{old}}, \alpha_S^{\text{new}})$  as the *action* executed by  $\alpha$  in state  $S$ . Unfortunately, this does not work, as this action would only consider the molecules *modified* by  $\alpha$ , but not the molecules *accessed* by  $\alpha$ .

In the distributed case, accessed molecules are significant: two assignment statements obviously cannot be executed concurrently if one of them modifies a location accessed by the other one. As a simple example, consider the two assignment statements  $\mathbf{a} := \mathbf{b}$  and  $\mathbf{b} := \mathbf{a}$ .



Hence, the action generated by  $\alpha$  needs to consider all molecules involved, i.e. all molecules that are modified or accessed. To formalize this idea, for a  $\Sigma$ -term  $t = \mathbf{f}(t_1, \dots, t_n)$  and a  $\Sigma$ -structure  $S$ , we define the *set of involved molecules*  $t_S^{\text{in}}$  inductively as

$$t_S^{\text{in}} =_{\text{def}} t_{1S}^{\text{in}} \cup \dots \cup t_{nS}^{\text{in}} \cup \{ (\text{loc}_S(t), t_S) \}.$$

For an assignment statement  $\alpha : t := t'$ , the set of involved molecules then is defined as

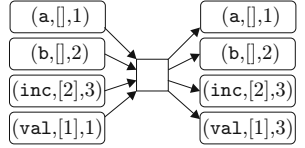
$$\alpha_S^{\text{in}} =_{\text{def}} t_S^{\text{in}} \cup t'_S{}^{\text{in}}.$$

More precisely, the set  $\alpha_S^{\text{in}}$  contains all involved molecules *before* the execution of  $\alpha$ . Correspondingly, the set

$$\alpha_S^{\text{out}} =_{\text{def}} (\alpha_S^{\text{in}} \setminus \alpha_S^{\text{old}}) \cup \alpha_S^{\text{new}}.$$

contains all involved molecules *after* the execution of  $\alpha$ . Notice the different meanings of the superscripts:  $^{\text{in}}$  and  $^{\text{out}}$  denote *all* involved molecules before and after the execution of  $\alpha$ , whereas  $^{\text{old}}$  and  $^{\text{new}}$  denote only the *modified* molecules before and after the execution of  $\alpha$ .

The notion of *action of  $\alpha$*  now is quite obvious: for a given state  $S$ , the action of  $\alpha$  in state  $S$  is defined by  $\alpha_S =_{\text{def}} (\alpha_S^{\text{in}}, \alpha_S^{\text{out}})$ . As an example, executing the assignment statement  $\text{val}(\mathbf{a}) := \text{inc}(\mathbf{b})$  in state  $Q$  yields the action



### 5.3 Guarded Assignment Statements

An assignment statement  $\alpha$  may furthermore be guarded by a *Boolean expression*  $\beta$ . In that way,  $\alpha$  is executed only in states that satisfy  $\beta$ .

In technical terms, a Boolean expression  $\beta$  consists of several *term equations* of the form  $t_1 = t_2$  connected by the usual Boolean connectives  $\neg$ ,  $\wedge$ , and  $\vee$ . For a given state  $S$ , the truth value of  $\beta$  is computed in the obvious way, where  $S \models \beta$  denotes that  $\beta$  holds in  $S$ . For an assignment statement  $\alpha$ , **if  $\beta$  then  $\alpha$**  is a *guarded assignment statement*,  $\gamma$ . For technical convenience, every assignment statement as introduced in the previous subsections is conceived as a guarded assignment statement whose guard holds in every state.

In addition to an ordinary assignment statement, a guarded assignment statement  $\gamma$  involves further molecules to evaluate the truth value of the guard  $\beta$ . Formally, let  $T^\beta$  denote the set of all terms occurring in  $\beta$ . For a state  $S$  satisfying  $\beta$ , the action of  $\gamma$  then is defined by  $\gamma_S =_{\text{def}} (\gamma_S^{\text{in}}, \gamma_S^{\text{out}})$  with

$$\gamma_S^{\text{in}} =_{\text{def}} \alpha_S^{\text{in}} \cup \bigcup_{t \in T^\beta} t_S^{\text{in}} \quad , \quad \gamma_S^{\text{out}} =_{\text{def}} (\gamma_S^{\text{in}} \setminus \alpha_S^{\text{old}}) \cup \alpha_S^{\text{new}}.$$

## 5.4 Sequential ASM Programs

Guarded assignment statements can be executed in parallel. In general, a finite, non-empty set of guarded assignment statements  $\Gamma = \{\gamma_1, \dots, \gamma_n\}$  is a *sequential ASM program*. At a given state  $S$ ,  $\Gamma$  executes simultaneously all assignment statements whose guards hold in  $S$ . More precisely,  $\Gamma$  replaces the molecules

$$\Gamma_S^{\text{old}} =_{\text{def}} \bigcup \{ \alpha_S^{\text{old}} \mid (\text{if } \beta \text{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}$$

by the molecules

$$\Gamma_S^{\text{new}} =_{\text{def}} \bigcup \{ \alpha_S^{\text{new}} \mid (\text{if } \beta \text{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}.$$

Note that  $\Gamma_S^{\text{new}}$  may be *inconsistent*, i.e. may contain two different molecules with the same location. For example, executing the sequential ASM program  $\Gamma = \{f(\mathbf{x}) := \mathbf{u}, f(\mathbf{y}) := \mathbf{v}\}$  in a state  $S$  with  $\mathbf{x}_S = \mathbf{y}_S$  and  $\mathbf{u}_S \neq \mathbf{v}_S$  yields an inconsistent  $\Gamma_S^{\text{new}}$ . As an inconsistent set of molecules cannot update a state,  $\Gamma$  executes no action in that case.

For a state  $S$  such that  $\Gamma_S^{\text{new}}$  is non-empty (i.e. at least one guard is satisfied) and consistent,  $\Gamma$  performs the action  $\Gamma_S =_{\text{def}} (\Gamma_S^{\text{in}}, \Gamma_S^{\text{out}})$  with

$$\Gamma_S^{\text{in}} =_{\text{def}} \bigcup_{\gamma \in \Gamma} \gamma_S^{\text{in}} \quad , \quad \Gamma_S^{\text{out}} =_{\text{def}} (\Gamma_S^{\text{in}} \setminus \Gamma_S^{\text{old}}) \cup \Gamma_S^{\text{new}}.$$

In this case,  $\Gamma_S$  is called an *action of  $\Gamma$* . The set of all actions of  $\Gamma$  is denoted by  $\mathcal{A}_\Gamma$ .

As an important property, no two actions from  $\mathcal{A}_\Gamma$  can be executed concurrently. The reason is that all actions in  $\mathcal{A}_\Gamma$  share the locations of the 0-ary function symbols occurring in  $\Gamma$ . As a consequence, the actions of a sequential ASM program  $\Gamma$  always occur totally ordered in a distributed run.

## 5.5 Distributed Abstract State Machines

In this section we finally introduce *distributed ASMs*. A distributed ASM  $\Delta$  specifies a set of sequential ASM programs, the *components* of  $\Delta$ . These components concurrently change the state of  $\Delta$  by performing actions as introduced above. More precisely, a *distributed ASM*  $\Delta$  consist of

- a signature  $\Sigma_\Delta$ ,
- a non-empty set  $\mathcal{S}_\Delta$  of  $\Sigma_\Delta$ -structures closed under isomorphism (the *states* of  $\Delta$ ),
- a non-empty set  $\mathcal{J}_\Delta \subseteq \mathcal{S}_\Delta$  closed under isomorphism (the *initial states* of  $\Delta$ ),
- a finite, non-empty set  $\mathcal{P}_\Delta$  of sequential ASM programs (the *components* of  $\Delta$ ), all built over signature  $\Sigma_\Delta$ .

Based on the notions introduced so far, the operational semantics of  $\Delta$  is easy to define: the actions of  $\Delta$  are constituted by the actions of the components of  $\Delta$ , i.e. the *set of actions of  $\Delta$*  is defined as

$$\mathcal{A}_\Delta =_{\text{def}} \bigcup_{\Gamma \in \mathcal{P}_\Delta} \mathcal{A}_\Gamma.$$

A *distributed run* of  $\Delta$  then is a distributed run  $R$  such that the initial state of  $R$  is an initial state of  $\Delta$  and every action occurring in  $R$  is an action of  $\Delta$ .

As an example, consider the following two sequential ASM programs built over signature  $\Sigma_Q$ :

$$A = \{ \text{val}(a) := \text{inc}(\text{val}(a)), a := \text{inc}(a) \},$$

$$B = \{ \text{if } (b = \text{val}(b)) \text{ then } b := \text{inc}(b) \}.$$

Intuitively,  $A$  subsequently increases the values of the function  $\text{val}$ , with  $a$  as the argument counter, whereas  $B$  increases the counter  $b$  as long as the values of  $b$  and  $\text{val}(b)$  coincide.

Figure 3 shows a distributed run of the components  $A$  and  $B$  at the initial state  $Q$ . For the sake of clarity, each occurring action is inscribed by the program causing the action.

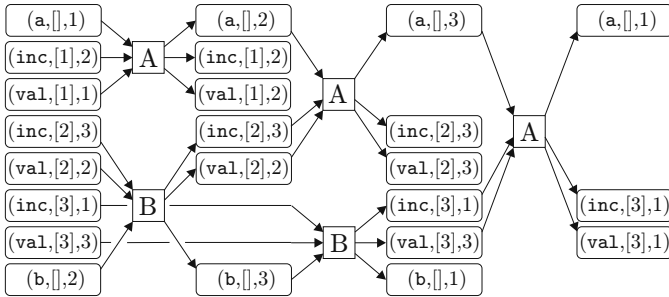


Fig. 3. A distributed run of the components  $A$  and  $B$

## 6 The Distributed ASM Theorem

In this section we present the main result of this paper: every distributed algorithm as introduced in Sect. 4 can be represented by a distributed ASM as introduced in Sect. 5. More precisely, the following theorem holds:

**Theorem 1.** *Let  $\mathcal{D}$  be a distributed algorithm according to [D1-D5](#). Then there exists a distributed ASM  $\Delta$  such that the distributed runs of  $\mathcal{D}$  and  $\Delta$  coincide.*

The reverse of Theorem 1 also holds: it is easy to prove that every distributed ASM constitutes a distributed algorithm by verifying the requirements [D1-D5](#). Hence, the expressive power of distributed algorithms and distributed ASMs coincide.

In the rest of this paper we outline the proof of Theorem 1. We present the most important lemmata and provide for each lemma an idea of the proof. Additionally, we show for each of the requirements [D1-D5](#) its applications in the course of the proof. A much more detailed proof of Theorem 1 (based on slightly different notations) is given in [5](#).

The main idea of the proof is the following: according to [D2](#), a distributed algorithm  $\mathcal{D}$  specifies a set of actions  $\mathcal{A}_{\mathcal{D}}$ . We then decompose  $\mathcal{A}_{\mathcal{D}}$  into finitely many equivalence classes and represent each of these classes by a sequential ASM program.

The equivalence relation that decomposes  $\mathcal{A}_{\mathcal{D}}$  is *action isomorphism*: two actions  $a$  and  $b$  are *isomorphic* if  $b$  can be derived from  $a$  by bijectively replacing the elements of  $a$ . Formally, the *set of elements* of an action  $a$  is defined as

$$E(a) =_{\text{def}} \{ u_1, \dots, u_n, v \mid (\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{in}} \cup a^{\text{out}} \}.$$

For two actions  $a$  and  $b$ , extend every bijective function  $p : E(a) \rightarrow E(b)$  canonically to the molecules of  $a$  and to  $a$  itself. Then  $p(a)$  denotes the action derived from  $a$  by replacing all elements according to  $p$ . The function  $p$  then is an *action isomorphism from  $a$  to  $b$*  iff  $p(a) = b$ . In case there is an action isomorphism from  $a$  to  $b$ ,  $a$  and  $b$  are *isomorphic*, written  $a \cong b$ . We denote by  $[a] =_{\text{def}} \{ b \mid b \cong a \}$  the *isomorphism class of  $a$* .

The relation  $\cong$  is an equivalence relation between actions, i.e.  $\cong$  decomposes  $\mathcal{A}_{\mathcal{D}}$  into disjoint subsets. Furthermore, the following lemma holds:

**Lemma 1.** *Let  $\mathcal{D}$  be a distributed algorithm. Then  $\cong$  decomposes  $\mathcal{A}_{\mathcal{D}}$  into finitely many disjoint subsets.*

*Idea of proof.* This lemma holds due to [D5](#): for each action  $a \in \mathcal{A}_{\mathcal{D}}$ ,  $|a^{\text{in}}|$  and  $|a^{\text{out}}|$  are bounded by a constant  $c \in \mathbb{N}$ . As a consequence, there exists a *finite* set  $M$  such that every action  $a \in \mathcal{A}_{\mathcal{D}}$  is isomorphic to an action  $a_M$  whose molecules contain only elements from  $M$ . But as  $M$  is finite, there can be only finitely many different actions  $a_M$ . As each equivalence class of  $\mathcal{A}_{\mathcal{D}}$  is represented by an action  $a_M$ ,  $\mathcal{A}_{\mathcal{D}}$  contains only finitely many equivalence classes.  $\square$

The next lemma states that  $\mathcal{A}_{\mathcal{D}}$  is closed under action isomorphism: for each action  $a$  of  $\mathcal{D}$ , each action  $b$  isomorphic to  $a$  is also an action of  $\mathcal{D}$ .

**Lemma 2.** *Let  $\mathcal{D}$  be a distributed algorithm and let  $a \in \mathcal{A}_{\mathcal{D}}$ . Then  $[a] \subseteq \mathcal{A}_{\mathcal{D}}$ .*

*Idea of proof.* According to [D2](#), there is a step  $R \xrightarrow{a} R'$  of  $\mathcal{D}$ . Let  $b$  be an action isomorphic to  $a$  with an action isomorphism  $p : a \rightarrow b$ . Based on  $p$ , a state  $S$  and an isomorphism  $i : R \rightarrow S$  can be constructed such that  $p \subseteq i$ . According to [D3](#), there is a step  $S \xrightarrow{i(a)} S'$  of  $\mathcal{D}$ . As  $i(a) = p(a) = b$ ,  $b$  is an action of  $\mathcal{D}$ .  $\square$

The following lemma states that the molecules of an action  $a \in \mathcal{A}_{\mathcal{D}}$  may be characterized by a finite set of terms  $T$ .

**Lemma 3.** *Let  $S \xrightarrow{a} S'$  be a step of  $\mathcal{D}$ . Then there exists a finite set  $T$  of  $\Sigma$ -terms such that*

- $a^{\text{in}} = \bigcup_{t \in T} t_S^{\text{in}}$ ,
- for each molecule  $(l, v) \in a^{\text{out}}$  there are terms  $t^l, t^v \in T$  such that  $\text{loc}_S(t^l) = l$  and  $t_S^v = v$ .

*Idea of proof.* According to [D4], for every molecule  $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{out}}$ ,  $a$  has access to the elements  $u_1, \dots, u_n$  and  $v$ . Then, for each  $x \in \{u_1, \dots, u_n, v\}$ , a term  $t^x$  along the inductive definition of “having access to” is constructed such that  $t_S^x = x$ . The set of all terms constructed in this way then constitutes  $T$ .

Let  $(l, v) \in a^{\text{out}}$  with  $l = (\mathbf{f}, [u_1, \dots, u_n])$ . Then for the term  $t^l =_{\text{def}} \mathbf{f}(t^{u_1}, \dots, t^{u_n})$  holds  $\text{loc}_S(t^l) = l$ , and for the term  $t^v$  holds  $t_S^v = v$ .  $\square$

Based on the set  $T$  of terms obtained in Lemma [B], a sequential ASM program  $\Gamma$  is constructed which performs all actions in the isomorphism class of  $a$ :

**Lemma 4.** *Let  $a$  be an action of  $\mathcal{D}$ . Then there exists a sequential ASM program  $\Gamma$  such that  $\mathcal{A}_\Gamma = [a]$ .*

*Idea of proof.* Let  $S \xrightarrow{a} S'$  be a step of  $\mathcal{D}$  and let  $T$  be as in Lemma [B]. By use of Lemma [B] for each molecule  $(l, v) \in a^{\text{out}}$  construct the assignment statement  $t^l := t^v$ . The set of all assignment statements obtained in this way is an ASM program  $\Gamma^0$  with  $\Gamma_S^0 = a$ . The demanded ASM program  $\Gamma$  is constructed from  $\Gamma^0$  by guarding the assignment statements in  $\Gamma^0$  by the Boolean expression  $\bigwedge_{t, t' \in T, t_S = t'_S} (t = t') \wedge \bigwedge_{t, t' \in T, t_S \neq t'_S} \neg(t = t')$ . Due to this guard,  $\Gamma$  only executes actions isomorphic to  $a$ .  $\square$

The previous lemmata then are composed in the main proof of Theorem [I].

*Idea of proof (of Theorem [I]).* According to Lemma [I], the equivalence relation  $\cong$  decomposes  $\mathcal{A}_\mathcal{D}$  into finitely many disjoint subsets  $C_1, \dots, C_n$ . According to Lemma [J], each  $C_i$  is closed under isomorphism. Then, according to Lemma [A], for each  $C_i$  there exists a sequential ASM program  $\Gamma_i$  such that  $C_i = \mathcal{A}_{\Gamma_i}$ . For the distributed ASM  $\Delta$  with  $\mathcal{S}_\Delta = \mathcal{S}_\mathcal{D}$ ,  $\mathcal{J}_\Delta = \mathcal{J}_\mathcal{D}$ , and  $\mathcal{P}_\Delta = \{\Gamma_1, \dots, \Gamma_n\}$  then holds

$$\mathcal{A}_\mathcal{D} = C_1 \cup \dots \cup C_n = \mathcal{A}_{\Gamma_1} \cup \dots \cup \mathcal{A}_{\Gamma_n} = \mathcal{A}_\Delta.$$

As all states, initial states, and actions of  $\mathcal{D}$  and  $\Delta$  coincide, the distributed runs of  $\mathcal{D}$  and  $\Delta$  are the same.  $\square$

## 7 Conclusion

The theory of ASMs suggests a comprehensive and quite general approach to the notion of “algorithm”. A number of variants of ASMs have been identified, among them sequential, interactive, parallel, and distributed versions. The deeper understanding of all such classes of ASMs requires a characterization of their expressive power. This has been achieved for many of them, including sequential, parallel, and interactive versions.

In this paper we characterized a class of distributed algorithms with bounded actions. As suggested by Gurevich in [G], and in the tradition of Petri and Pratt, we define distributed runs as sets of action occurrences, partially ordered by causal dependencies. We furthermore showed that this class of distributed algorithm is captured by the operational computation model of distributed ASMs, which is based on sequential ASMs [H].

We intend to extend the result of this paper to more general variants of distributed ASMs. As discussed in Sect. 5, in [6] a version of distributed ASMs is introduced which offers advanced features such as concurrent access to locations and dynamic instantiation of new agents. We will examine how the requirements [D1](#) – [D5](#) can be generalized in order to capture those features.

## References

1. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic* 4(4), 578–651 (2003)
2. Blass, A., Gurevich, Y.: Ordinary Interactive Small-Step Algorithms, parts I, II, III. *ACM Trans. Comput. Logic* (2006)
3. Blass, A., Gurevich, Y., Rosenzweig, D., Rossman, B.: General Interactive Small Step Algorithms. Technical Report MSR-TR-2005-113, Microsoft Research (August 2006)
4. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
5. Glausch, A., Reisig, W.: Distributed Abstract State Machines and Their Expressive Power, Humboldt-Universität zu, Berlin. *Informatik-Berichte*, vol. 196 (2006), [http://www.informatik.hu-berlin.de/top/download/publications/GlauschR2006\\_hub\\_tr196.pdf](http://www.informatik.hu-berlin.de/top/download/publications/GlauschR2006_hub_tr196.pdf)
6. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (ed.) *Specification and Validation Methods*, pp. 9–36. Oxford University Press, Oxford (1995)
7. Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic* 1(1), 77–111 (2000)
8. Gurevich, Y., Yavorskaya, T.: On Bounded Exploration and Bounded Nondeterminism. Technical Report MSR-TR-2006-07, Microsoft Research (January 2006)
9. ITU-T. SDL Formal Semantics Definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union (November 2000)
10. Petri, C.A.: Non-Sequential Processes. Interner Bericht ISF-77-5, Gesellschaft für Mathematik und Datenverarbeitung (1977)
11. Reisig, W.: *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York (1985)
12. Reisig, W.: On Gurevich’s Theorem on Sequential Algorithms. *Acta Informatica* 39(5), 273–305 (2003)
13. Stärk, R., Schmid, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, Heidelberg (2001)
14. Pratt, V.: Modeling Concurrency with Partial Orders. *Int. J. of Parallel Programming* 15(1), 33–71 (1986)

# Using Abstract State Machines for the Design of Multi-level Transaction Schedulers

Markus Kirchberg<sup>1</sup>, Klaus-Dieter Schewe<sup>2</sup>, and Jane Zhao<sup>2</sup>

<sup>1</sup> Institute for Infocomm Research (I<sup>2</sup>R), A\*STAR, Singapore  
mkirchberg@i2r.a-star.edu.sg

<sup>2</sup> Information Science Research Centre, Palmerston North, New Zealand  
kdschewe@acm.org, janeqzhao@hotmail.com

**Abstract.** Multi-level transactions have been suggested as an approach to increase transaction throughput in databases. The central idea is to enable some low-level conflicts to be ignored by taking higher-level application semantics into account. In this paper, we approach the formal specification of a multi-level transaction scheduler using Abstract State Machines. We are particularly interested in showing that concrete protocols for multi-level transaction processing arise as refinements of an abstract ground model specification. Furthermore, we are interested in the proof of desirable properties of such schedulers such as the correctness and if possible also completeness with respect to serialisability, and the recoverability of the accepted schedules. For this we investigate a two-phase locking and a hybrid protocol.

## 1 Introduction

Transaction processing is an important component of any database management system, as it enables concurrent access to databases by multiple users. In most applications thousands of transactions have to be processed each second. So, transaction throughput is important for the performance of these systems.

The commonly adopted transaction model only looks at sequences of read and write operations on database “objects” [6]. In most cases these objects are physical pages, but also a finer granularity, e.g. records or even record fields, could be considered. A schedule, i.e. an interleaving of sequences that stem from different transactions, can be accepted, if it is equivalent to a serial one using any order of the involved transactions. The most common notion of equivalence of schedules requires that conflicting operations, i.e. operations on the same object, one of which is a write, from different transactions appear in the same order in equivalent schedules.

A lot of research has been undertaken over decades to enhance this simple model of “flat” transactions to increase transaction throughput, of course without violating the serialisability request. An important improvement is offered by the model of multi-level transactions [1], which is based on the idea of ignoring some low-level conflicts by taking higher-level application semantics into account. In other words, a transaction is defined via operations on various levels, e.g. pages, records and fields, and conflicts are defined on each level. Then

it may happen that low-level operations are in a conflict, whereas their parent operations are not, in which case the low-level conflict can be ignored.

Many concurrency control protocols have been suggested for the multi-level transaction model [10,16] including generalisations of locking protocols and hybrid protocols that combine optimistic with locking strategies [14]. Moreover, the model has been promoted as part of a general architecture for distributed object bases [9]. Nevertheless, it has not yet been widely adopted in practical database management systems. This may reflect some problems with the definition of high-level objects as such, which is rather vague in the original model, and unknown performance issues. The work in [8] is one of the few dealing with an in-depth analysis of the performance of multi-level concurrency control protocols. It supports the vague observation made in [14] that locking protocols eat up the envisioned advantages of the model, whereas for optimistic or hybrid protocols the cases, where the model can be winningly applied, still have to be figured out.

That is, it is necessary to further investigate concurrency control protocols for multi-level transactions. The problem is that protocols are usually developed in an ad-hoc way. Desirable properties such as correctness with respect to serialisability, recoverability or strictness are repeatedly verified, if at all. It would be better, if such properties became part of a formal system specification, so that there is no need to verify them again for refinements of protocols. We therefore think it is a good idea to start from a high-level formal specification of a multi-level transaction scheduler that is then subject to a refinement process leading to the specification of a concrete protocol.

We propose to use Abstract State Machines (ASMs) [4] for this purpose because of their explicit support of a methodology consisting of building an initial “ground model” [2] and applying refinements [3] to it without being forced into detailed verification at all levels. Furthermore, ASMs have already been applied in various areas including database recovery [7], data warehouses [18], and general database transactions [12], i.e. areas that are closely related to the one we are interested in. In [11] it has been argued that for databases it may be advantageous not to rely on the basic ASM method, but to employ a typed ASMs as defined in [5] or [17] instead. This, however, applies to database applications, whereas for internal aspects of database management systems such as the concurrency control component untyped ASMs are sufficient.

In this paper, we present initial results of applying ASMs to the specification of multi-level concurrency control protocols. We start by defining an ASM ground model in Section 2. This amounts to just specifying the multi-level transaction model and the task of the scheduler to accept or reject commit-requests. We enhance this model by defining quality criteria for serialisability and recoverability in Section 3, before we start looking at refinements that lead to concrete concurrency control protocols. Our refinements will be fairly standard in the sense of the work in [4,13]. Basically, we will only exploit  $(1, n)$ -refinements that result from adding new rules, conservative extensions, and pre-post extensions. We first discuss the standard strict two-phase locking protocol str-2PL in



Section 4, before approaching the hybrid FoPL protocol in Section 5. In both cases we use refinements of the ASM ground model. In the case of the hybrid protocol we can further refine the specification by permitting lazy aborts, which leads to the FoPL<sup>+</sup>-protocol. This protocol is indeed not only correct, but also complete. We summarise our conclusions in Section 6 emphasising also further refinements dealing with operations that absorb each other or early aborts.

## 2 ASM Ground Model for Multi-level Transaction Processing

The basic operation of a transaction scheduler is simply to read incoming operations and to schedule them. In general, the read involves checking that the operation is well-defined and belongs to a transaction, thus it also involves maintaining transaction and operation tables. The scheduling involves checking, whether the operation can be executed or has to be aborted. Furthermore, the schedule, i.e. the sequence of the accepted operations, has to be cleaned in case of aborts and transactions successfully leaving the system.

Thus, in terms of ASMs we start with four nullary functions in the signature:

$next(0)$  monitored     $new(0)$  dynamic     $new\_op(0)$  controlled     $tid(0)$  dynamic

If  $new$  is defined, then this indicates that a new operation or transaction has arrived that awaits scheduling. In this case the value of  $next$  will be this new operation. In addition, in case of a top-level transaction, the value of  $tid$  will be set to the transaction identifier. If the operation is well-defined, the flag  $new\_op$  will be set, and scheduling will be started. That is,  $next$  is controlled only by the environment, i.e. the run-time database system, and thus has been modelled by a monitored function. On the other hand,  $new\_op$  is completely controlled by the scheduler, so it is modelled as a controlled function, while  $new$  is controlled by both the scheduler and the environment.

Then the main rule of a transaction scheduler ASM takes the following simple form:

```

main = if   new ≠ ⊥
      then ( add_op(next) || new := ⊥ );
          if   new_op ≠ ⊥
            then schedule(next) ;
              execute(next)
          endif
      endif

```

### 2.1 Multi-level Transactions

According to [14] an  $n$ -level system consists of  $n$  levels  $L_i = (\mathfrak{D}_i, \mathfrak{F}_i)$  ( $i = 0, \dots, n-1$ ), where  $\mathfrak{D}_i$  is a set of *objects* and  $\mathfrak{F}_i$  a set of *operators*. An  $L_i$ -operation is an element of  $\mathcal{O}_i = \mathfrak{F}_i \times \mathfrak{D}_i$ . Therefore, we model operations and objects in our scheduler ASM by two static functions:

*ops*(2) static      *obj*(2) static

Then  $ops(\ell, n) = 1$  means that there is an operator with the name  $n$  on level  $\ell$ . Analogously,  $obj(\ell, o) = 1$  means that there is an object with the name  $o$  on level  $\ell$ .

Let us now look closer at the incoming operations and how they are handled by the scheduler, more precisely by the rule `add_op` used above. For this we adopt the definitions of index tree and  $n$ -level-transaction from [14]. An *index tree* of depth  $n$  is a finite subset  $I \subseteq (\mathbb{N} - \{0\})^*$  such that  $\epsilon \in I$ ,  $\alpha(k+1) \in I \Rightarrow \alpha k \in I$ , and  $\alpha \in I \wedge |\alpha| < n \Leftrightarrow \alpha 1 \in I$  for all  $\alpha \in (\mathbb{N} - \{0\})^*$  and  $k \in \mathbb{N}$  hold. Here, as a syntactic convention, we used small Greek letters  $\alpha, \beta, \mu, \nu, \dots$  for number sequences and small Latin letters  $i, j, k, \ell, \dots$  for the numbers in these sequences.

An  $n$ -level-transaction  $T_j$  consists of an index tree  $I$  of depth  $n$ , a mapping which assigns to each non-empty  $\alpha \in I$  an  $L_{n-|\alpha|}$ -operation, denoted as  $o_{j\alpha}$ , and partial orders  $<_i^{(j)}$  (called  $L_i^{(j)}$ -precedence relation) on each  $\mathfrak{D}_i^{(j)} = \{o_{j\alpha} \mid |\alpha| + i = n\}$ , such that  $o_{j\alpha k} <_i^{(j)} o_{j\alpha \ell} \Rightarrow k < \ell$  holds. Furthermore,  $o_{j\alpha} <_i^{(j)} o_{j\beta}$  holds iff  $o_{j\alpha k} <_{i-1}^{(j)} o_{j\beta \ell}$  holds for all  $k$  and  $\ell$ .

Therefore, in our specification we model operations by triples  $(\alpha, n, o)$ , where  $\alpha$  is a unique identifying number sequence for the operation,  $n$  is the operator name, and  $o$  is the object the operation is applied to. In particular, the value *next* is bound to will always have this form. In case of a top-level transaction we get  $\alpha = \epsilon$ , so we create a transaction number using the controlled function `last_tno(0)`, which contains the last assigned transaction number. For all other operations we assume that we already get the unifying number sequence, which of course could be determined from identifiers of the operation itself and its parent. To simplify our model, we use two unary static functions *parent* and *index*, which will produce the identifying sequence of the parent operation and the index, i.e.  $k$  for the  $k$ 'th child of the parent. Furthermore, we will need the functions

*level*(3) derived      *max\_level*(0) static

Obviously, the latter one defines the number of levels used, while the former one is defined by

$$level(\alpha, n, o) = \begin{cases} max\_level & \text{if } \alpha = \epsilon \\ level(\beta, n', o') - 1 & \text{if } \exists i, n', o'. \alpha = \beta i \wedge ops\_table(\beta, n', o') \neq \perp \end{cases}$$

Then the multi-level transactions that are in the system and the operations involved in them can be modelled by three controlled functions

*ops\_table*(3) controlled    *precedence*(4) controlled    *trans\_table*(2) controlled

Obviously,  $ops\_table(\alpha, n, o)$  is defined iff  $(\alpha, n, o)$  is an operation in the system with components as just defined,  $precedence(i, j, \alpha, \beta)$  is defined iff  $op_\alpha <_i^{(j)} op_\beta$  holds for the operations with the identifying sequences  $\alpha$  and  $\beta$ , respectively,

and  $trans\_table(j, id)$  is defined iff the top-level transaction with number  $j$  has the identifier  $id$ .

With these preliminaries the rule `add_op` used in the main rule is defined as follows:

```

add_op(op) = if   op = (ϵ, ⊥, ⊥) ∧ tid ≠ ⊥
               then last_tno := last_tno + 1 ;
                   ( ops_table(last_tno, ⊥, ⊥) := 1 || tid := ⊥ ||
                     trans_table(tid, last_tno) := 1 || new_op := 1 )
               elsif ∃α, n, o . op = (α, n, o) ∧ α ≠ ⊥ ∧
                   ops(level(op), n) = 1 ∧ obj(level(op), o) = 1 ∧
                   ∃np, op. ops_table(parent(α), np, op) ≠ ⊥ ∧
                   ( index(α) ≠ 1 ⇒ ∃ns, os.
                     ops_table(parent(α)∧(index(α)-1), ns, os) ≠ ⊥ )
               then let α = π1(op) ∧ n = π2(op) ∧ o = π3(op)
                   in begin
                       ( ops_table(α, n, o) := 1 || new_op := 1 ) ;
                       check_prec(α, n, o)
                   end
               else new_op := ⊥
               endif

```

Here the called rule `check_prec` will define the precedence relations leading to a total order on level 0 and partial orders on all other levels. We dispense with the straightforward details of this rule. Note that the addition of rule `add_op` could already be considered a  $(1, n)$ -refinement.

## 2.2 Multi-level Schedules

The execution of concurrent transactions is described by an  $n$ -level-schedule. According to [14] for a set  $\mathfrak{D}_n = \{T_1, \dots, T_k\}$  of  $n$ -level-transactions let  $\mathfrak{D}_i = \bigcup_{j=1}^k \mathfrak{D}_i^{(j)}$  be the set of all  $L_i$ -operations in these transactions ( $0 \leq i < n$ ). Then a (complete)  $n$ -level-schedule on  $\mathfrak{D}_n$  is given by a partial order  $<_0$  on  $\mathfrak{D}_0$  containing all  $L_0^{(j)}$ -precedence relations. Then  $<_0$  induces a partial order  $<_i$  on each level by

$$o_\mu <_{i+1} o_\nu \Leftrightarrow \forall o_{\mu k} \in act(o_\mu). \forall o_{\nu l} \in act(o_\nu). o_{\mu k} <_i o_{\nu l}.$$

As schedules are built-up by incoming operations, we need the notion of a partial schedule. For this, following [14], we define a *prefix* of an  $n$ -level-transaction  $T_j$  to consist of subsets  $\mathfrak{P}_i^{(j)} \subseteq \mathfrak{D}_i^{(j)}$  ( $i = 0, \dots, n$ ) such that

- $o_{j\alpha} <_i^{(j)} o_{j\beta} \wedge o_{j\beta} \in \mathfrak{P}_i^{(j)} \Rightarrow o_{j\alpha} \in \mathfrak{P}_i^{(j)}$  and
- $o_{j\alpha} \in \mathfrak{P}_i^{(j)} \Rightarrow o_{parent(j\alpha)} \in \mathfrak{P}_{i+1}^{(j)}$

hold, whenever the involved operations are defined. As the selection of subsets for a prefix defines an underlying subtree of the index-tree, we may treat

prefixes as if they were (complete) transactions. Thus, we may define schedules on the basis of prefixes, so we obtain *partial n-level-schedules*, for which we write  $(\mathfrak{P}_n, \dots, \mathfrak{P}_0, <_0)$ . Here  $\mathfrak{P}_n = \{P_1, \dots, P_k\}$  is a set of  $n$ -level-prefixes,  $\mathfrak{P}_i = \bigcup_{j=1}^k \mathfrak{P}_i^{(j)}$  and  $<_0$  is a partial order on  $\mathfrak{P}_0$  containing all  $L_0^{(j)}$ -precedence relations restricted to  $\mathfrak{P}_0$ . If all  $P_j$  are transactions, we obtain a *complete* schedule.

On these grounds, the scheduling of incoming operations via the schedule rule used in the main rule is quite straightforward. We only need another controlled function  $g\text{-precedence}(3)$  with  $g\text{-precedence}(i, \alpha, \beta)$  being defined iff  $o_\alpha <_i o_\beta$  holds. So, the schedule rule simply has to create these “global” precedence relations:

```

schedule( $\alpha, n, o$ ) =
  if    level( $\alpha, n, o$ ) = 0
  then forall  $\beta, n', o'$ 
        with ops_table( $\beta, n', o'$ )  $\neq \perp \wedge$  level( $\beta, n', o'$ ) = 0
        do  g-precedence( $0, \beta, \alpha$ ) := 1
        enddo
  elsif  $n = \text{'commit'} \vee n = \text{'abort'}$ 
  then let  $\beta = \text{parent}(\alpha) \wedge i = \text{level}(\alpha, n, o)$ 
        in  begin
            choose  $n', o'$ 
            with ops_table( $\beta, n', o'$ )  $\neq \perp$ 
            do forall  $\gamma, n'', o''$ 
                  with ( ops_table( $\gamma, n'', o''$ )  $\neq \perp \wedge$ 
                        level( $\gamma, n'', o''$ ) =  $i + 1 \wedge$ 
                         $\forall k, \ell. ( \exists n_1, o_1. \text{ops\_table}(\gamma k, n_1, o_1) \neq \perp \wedge$ 
                         $\exists n_2, o_2. \text{ops\_table}(\beta \ell, n_2, o_2) \neq \perp )$ 
                         $\Rightarrow g\text{-precedence}(i + 1, \gamma k, \beta \ell) \neq \perp )$ 
                  do  g-precedence( $i, \gamma, \beta$ ) := 1
                  enddo
            enddo
        end
  endif

```

Again, the definition of rule schedule can be considered to constitute a simple  $(1, n)$ -refinement.

Of course, scheduling of transactions means to permit only schedules that satisfy certain quality criteria. We will discuss these criteria in the next section. Then we will refine the schedule rule, which can be done in a preemptive or an optimistic way. Roughly, the difference is that preemptive scheduling checks conditions *before* the operation is added to the schedule, while optimistic scheduling checks conditions at commit-time, i.e. *after* completion of a transaction. In both cases the violation of the check conditions leads to abortion, which may be relaxed to waiting. We will not discuss details of abortion here.

### 3 Serialisability and Recoverability

A consequence of atomicity of transactions is the request that schedules must be serialisable, i.e. equivalent to a serial schedule, in which transactions would be executed one after another. Different notions of equivalence exist, the most common one – for reasons of decidability and efficiency – being conflict-serialisability, which defines schedules to be equivalent iff conflicting operations appear in the same order in the schedules.

In the model of multi-level transactions conflicts are defined by using level-specific, symmetric *conflict relations*  $CON_i \subseteq \mathcal{O}_i \times \mathcal{O}_i$ . Non-conflicting operations should commute, and conflicts can only occur on the same object, i.e.  $((op_1, x), (op_2, y)) \in CON_i \Rightarrow x = y$ , so conflict relations are de facto defined on operators rather than operations. In terms of ASMs, it thus suffices to provide two functions

$$op-con(3) \text{ static} \quad con(5) \text{ derived}$$

with  $op-con(i, n, n')$  being defined iff the  $L_i$ -operators  $n, n' \in \mathfrak{F}_i$  are conflicting, and  $con(i, n, o, n', o')$  being defined iff  $op-con(i, n, n')$  and  $o = o' \in \mathfrak{D}_i$  hold.

Furthermore, these conflict relations must satisfy a natural *conformity* condition, which simply requests that whenever operations  $o_\alpha$  and  $o_\beta$  conflict, then there must exist children  $o_{\alpha k}$  and  $o_{\beta \ell}$  on the next lower level that also conflict each other. On these grounds, we can define

$$o_{j\mu} \rightarrow_i o_{j'\nu} \Leftrightarrow j \neq j' \wedge (o_{j\mu}, o_{j'\nu}) \in CON_i \wedge o_{j\mu} <_i o_{j'\nu}$$

for  $o_\mu, o_\nu \in \mathfrak{D}_i$ .

Then according to [14] two  $n$ -level-schedules are *conflict-equivalent* iff their associated relations  $\rightarrow_i$  coincide for all  $i = 0, \dots, n-1$ . An  $n$ -level schedule which is conflict-equivalent to a serial one, is called *serialisable*. A partial schedule is serialisable iff it can be extended to a complete serialisable schedule.

Serialisability determines which schedules should not be accepted. It does, however, not yet help to decide, whether a transaction can be *committed*, i.e. successfully completed. Even if the partial schedule at hand is serialisable, a completed transaction cannot be simply taken away from it, because we simply have no control over the completion, which depends on the incoming operations. This could be mended by requiring that the completed transaction is the first in the equivalent serial schedule, but this would rule out too many schedules. What we require instead is *recoverability*, i.e. a transaction can only commit, if no later abort of another transaction would require the transaction at hand to be aborted as well. In other words, a committed transaction is considered to be final forever, and no failure should be possible.

For multi-level transactions recoverability has been formalised in [14] by means of marked schedules. A partial schedule  $S = (\mathfrak{P}_n, \dots, \mathfrak{P}_0, <_0)$  is turned into a *marked schedule* by a partial mapping  $m : \bigcup_{i=0}^n \mathfrak{P}_i \rightarrow \{c, a\}$  such that the following hold:

- If  $m(o')$  is defined for all successors  $o'$  of  $o \in \mathfrak{P}_i$ , then  $m(o)$  must also be defined.
- Whenever  $m(o) = c$  and  $o'$  is a successor of  $o$ , then we also have  $m(o') = c$ . Whenever  $m(o) = a$  holds, there must exist some successor  $o'$  with  $m(o') = a$ .
- Whenever  $o' <_i o$  holds, then  $m(o') = c$  must hold.

Furthermore, we need the notion of strong dependence of operations:  $o_{j'\nu}(x)$  *strongly depends* on  $o_{j\mu}(x)$  (notation:  $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$ ) iff  $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$  holds and the effect of the sequence  $o_{j\mu}(x); o_{j'\nu}(x)$  (on the database) differs from the effect of  $o_{j'\nu}(x)$ .

Then, according to [14] a schedule  $(\mathfrak{D}_n, \dots, \mathfrak{D}_0, <_0)$  is *recoverable on level  $L_i$*  iff for all prefixes  $S$ , all markings  $m$  of  $S$  and all  $j \neq j'$

$$o_{j\mu k} \rightarrow_{i-1} o_{j'\nu \ell} \wedge m(o_{j'\nu}) = c \Rightarrow m(o_{j\mu}) = c$$

holds. Recoverability does not exclude cascading aborts, so recovery may still be a complex task. We dispense, however, with discussing the stronger notions of cascade-freeness or even strictness.

## 4 The str-2PL-Refinement

Each concurrency control protocol should result as a refinement of the scheduling rule defined in Section 2. Let us start with discussing the common two-phase locking protocol (2PL) in its strict version (str-2PL). Locking protocols work preemptively, i.e. before an operation is scheduled, it must acquire a lock on the object it wants to access. Thus, following [14] for each  $L_i$ -operator  $op \in \mathfrak{F}_i$  we define a specific lock  $lock_{op}$ . Then, each  $L_i$ -operation  $op_{\mu k}(x)$  may only be executed after setting a lock, namely  $lock_{op}$ , on the object  $x$ . In addition, we associate with this lock the identifying number sequence  $\mu$  of its parent  $o_\mu$ . After its commit,  $o_\mu$  must release all its locks. An  $L_i$ -object  $x$  may hold several locks at a time, provided the associated operations do not conflict with each other, i.e. if  $lock_{op_1}$  and  $lock_{op_2}$  are locks on object  $x \in \mathfrak{D}_i$  issued by the  $L_i$ -operations  $o_{\mu k}$  and  $o_{\nu \ell}$ , respectively, then these locks are *incompatible* iff  $o_{\mu k} \rightarrow_i o_{\nu \ell}$  or  $o_{\nu \ell} \rightarrow_i o_{\mu k}$  holds.

Thus, an operation may only set a lock on  $x$ , if this is not incompatible with any existing lock on  $x$ . Otherwise, the operation has to be aborted or must wait until all incompatible locks on  $x$  are released. In 2PL we have a *growing phase*, in which all locks are acquired, but none can be released, followed by a *shrinking phase* in which existing locks will be released, but no new lock can be acquired. In str-2PL no lock will be released before commit or abort. So, the str-2PL-refinement of the main rule will be

```

main = if    new  $\neq \perp$ 
      then ( add_op(next) || new :=  $\perp$  );
      if    new_op  $\neq \perp$ 
      then schedule(next) ;

```

```

    acquire_lock(next) ;
    if    lock_table(next) ≠ ⊥
    then execute(next)
    else  wait_or_abort(next)
    endif
  endif
endif

```

In doing so, we actually combine two  $(1, n)$ -refinements. First, we apply a *wrapping* refinement – in general, this means to replace a rule  $\langle \text{old} \rangle$  by an extended rule of the form  $\langle \text{pre} \rangle; \langle \text{old} \rangle; \langle \text{post} \rangle$  – to the execute rule. In fact, we only add a preprocessing rule `acquire_lock`. In a second step we apply a *conservative extension* – in general, this means to replace a rule  $\langle \text{old} \rangle$  by a conditional rule **if**  $\langle \text{condition} \rangle$  **then**  $\langle \text{old} \rangle$  **else**  $\langle \text{new} \rangle$  **endif** – to the execute rule. Such refinements were discussed intensively in [43, 13].

We dispense with discussing details of the `wait_or_abort` rule. If waiting is possible, the operation will arrive as a new operation again later, while abortion leads to a set of complex actions. So, we can concentrate on the `acquire_lock` rule, for which we need the ternary controlled function `lock_table`. Then the rule takes the following form:

```

acquire_lock( $\alpha, n, o$ ) =
  let     $i = \text{level}(\alpha, n, o)$ 
  in begin
    if     $\forall \beta, n'. (\text{lock\_table}(\beta, n', o) \neq \perp \Rightarrow (\text{first}(\alpha) = \text{first}(\beta) \vee \text{con}(i, n', o, n, o) \neq \perp \vee \text{g-precedence}(i, \beta, \alpha) \neq \perp))$ 
    then  $\text{lock\_table}(\alpha, n, o) := 1$ 
    endif
  end

```

Furthermore, we have to refine the schedule rule in a way that a commit-operation releases all locks. As there is nothing more to do in case of commit, this is straightforward. As shown in [14, Thm.1 & Prop.1] schedules accepted by str-2PL are always recoverable and serialisable. We dispense with repeating the proofs here.

## 5 The Basic FoPL-Refinement

The hybrid FoPL-protocol introduced in [14] is an alternative to str-2PL that does not require locks. In the tradition of optimistic concurrency control, FoPL consists of three phases: propagation, validation and execution. In the *propagation phase* the operations at the various levels  $L_i$  are executed. In addition, *flaglists* for the objects are built up that will later be used to decide, whether an operation commits or aborts. The task of the *validation phase* is to perform this decision. The flaglists are used to detect, whether the interleaved execution of the operations has lead to a situation that forces an abort or not. Finally, in the

*execution phase* the commit or abort is executed. The commit-case is the easier one, as it suffices to remove flags from flaglists, while the abort-case requires additional efforts for rollback.

So, the refinement of the main rule originating from propagation is simply to set flags before scheduling, i.e. the main rule becomes:

```

main = if    new  $\neq \perp$ 
      then ( add_op(next) || new :=  $\perp$  ) ;
      if    new_op  $\neq \perp$ 
      then set_flag(next) ;
          schedule(next) ;
          if    n  $\neq$  'end'
          then execute(next)
          else choose  $\beta, n_p, o_p$ 
                with     $\beta = \text{parent}(\pi_1(\text{next})) \wedge$ 
                        ops_table( $\beta, n_p, o_p$ )  $\neq \perp$ 
                do    validate( $\beta, n_p, o_p$ ) ;
                        if    successful( $\beta$ )  $\neq \perp$ 
                        then commit( $\beta$ )
                        else wait_or_abort( $\beta, n_p, o_p$ )
                endif
          enddo
      endif
endif

```

The kind of refinement we applied in this case is again a combination of a simple wrapping with a conservatibve extension. In fact, we added `set_flag` as preprocessing rule to schedule, then added a condition to the execute rule with an extensive rule for the else-case.

As an  $L_i$ -operation  $o_\mu$  is implemented by means of its children on level  $L_{i-1}$ , we mark the objects in  $\mathfrak{D}_{i-1}$  that are accessed by  $o_\mu$ . For a successor  $o_{\mu k}$  of  $o_\mu$  defined by the operation  $(op, o)$  we use the *flag*  $(op, \mu)$  on  $o$ . Flags are organised in flaglists for all objects  $o$ , which are built dynamically extending  $<_{i-1}$ .

In terms of ASMs, we simply need a 4-ary controlled function *flag* to specify the `set_flag` rule:

```

set_flag( $\alpha, n, o$ ) = choose k
      with  $k \geq 0 \wedge \forall \ell, n', \beta. (\text{flag}(o, \ell, n', \beta) \neq \perp \Rightarrow \ell < k) \wedge$ 
          ( $\exists n', \beta. \text{flag}(o, k-1, n', \beta) \neq \perp \vee k = 0$ )
      do    choose  $\beta$ 
          with     $\beta = \text{parent}(\alpha)$ 
          do    flag( $o, k, n, \beta$ ) := 1
          enddo
      enddo

```

Validation of an  $L_i$ -operation  $o_\mu$  first has to check, whether all flags of successor operations that implement  $o_\mu$  are still set – flags may have been removed from a



flaglist by another operation. For the flaglists of all objects  $o \in \mathfrak{D}_{i-1}$  that were accessed by one of these operations during the propagation phase, exclusive locks will be requested and kept until the end of the commit-phase. If at least one flag is missing, the operation  $o_\mu$  must be aborted. Otherwise the protocol checks, whether  $o_\mu$  was *successful*. This is the case, if none of the objects accessed by  $o_\mu$  was accessed earlier by some other conflicting operation  $o_\nu$ . This can be detected from the flaglists.

According to [14] this can be formalised as follows: An  $L_i$ -operation  $o_\mu$  is *blocked* on an object  $o \in \mathfrak{D}_{i-1}$  iff there are flags  $(op_1, \nu)$  and  $(op_2, \mu)$  in its flaglist with  $\nu \neq \mu$  such that  $(op_1, \nu)$  precedes  $(op_2, \mu)$  and  $((op_1, o), (op_2, o)) \in CON_{i-1}$  holds. An  $L_i$ -operation  $o_\mu$  is *successful* iff it is not blocked on any object  $o$  that it accesses.

In terms of ASMs, we have to specify the validate rule used in the refined main rule above. This rule uses the unary controlled function *successful*:

```

validate( $\alpha, n, o$ ) =
  successful( $\alpha$ ) := 1 ;
  forall  $\beta, n_c, o_c$ 
  with  $\exists k. \beta = \alpha k \wedge ops\_table(\beta, n_c, o_c) \neq \perp$ 
  do set_lock( $o_c, \alpha$ ) ;
    if  $\neg \exists \ell. flag(o_c, \ell, n_c, \alpha) \neq \perp$ 
    then successful( $\alpha$ ) :=  $\perp$ 
    else choose  $k$ 
      with  $flag(o_c, k, n_c, \alpha) \neq \perp$ 
      do if  $\exists \gamma, \ell, n'_c. (\ell < k \wedge \gamma \neq \alpha \wedge$ 
           $flag(o_c, \ell, n'_c, \gamma) \neq \perp \wedge$ 
           $con(level(\beta, n_c, o_c), n'_c, o_c, n_c, o_c) \neq \perp)$ 
          then successful( $\alpha$ ) :=  $\perp$ 
          endif
      enddo
    endif
  enddo
enddo

```

As the actual commit in the commit-phase mainly amounts to removing flags, we can dispense with the straightforward details. On the other hand, wait and abort rules will require some sophisticated refinements.

## 6 Conclusion

In this paper, we approached the specification of multi-level concurrency control protocols using Abstract State Machines (ASMs). First of all, this is of course an exercise in the application of ASMs in an important area of database systems, and it is no surprise that this could be achieved without any difficulty. It is also not surprising that the ground model specification and its refinements towards a locking and a hybrid protocol can be used to prove important characteristics of

protocols such as recoverability and serialisability of the accepted schedules. In this sense we simply translated the work done in [14] to the framework of ASMs.

What is more important is that the development of the specifications follow a uniform pattern. Both protocol specifications arise as refinements of a very general specification, which does not specify more than the very basics of transaction scheduling. This suggests the refinement-based ASM-method as a promising approach to develop various other protocols, where each refinement step is motivated by a desirable property of the schedules. In particular, the basic refinement steps that we outlined in this paper were motivated by the requirement to accept only serialisable schedules, in which all transactions are recoverable. This goal has been achieved in a systematic way.

The refinement-based specification of the protocols is, however, not yet completed. We left out the important aspect of abortion, which in itself leads to complex specifications. We will complement our work elsewhere. Furthermore, we left out other possible refinements of the protocols, e.g. waiting strategies in case immediate aborts are not desired. In case of locking protocols this involves the whole area of deadlock detection, while for the hybrid FoPL protocol even completeness can be verified, i.e. all serialisable schedules will be accepted, if the basic protocol is refined by a waiting strategy.

In [14] further refinements of the FoPL protocol were discussed. These comprise early aborts in case a flag is removed, which requires communication among transactions. It further comprises partial abortion, which amounts to support restarting transactions, and the support of absorbing operations. All these protocol extensions lead to further refinement of our ASMs, thus remain within the outlined framework. They further lead directly to verifiable properties of the accepted schedules. In view of the work in [15] it is further possible to extend the work towards distributed concurrency control.

All this together demonstrates the strength of the ASM method. The specifications are easy to obtain, and the refinements are achieved systematically. While our study has been placed in an application area, where the theoretical results are known, the easiness and flexibility of the approach suggests to apply the method also to other problem areas.

In continuation of the work reported in this paper, we intend to complete this study adding details of the indicated further refinements. We further intend to complete the verification work also with respect to properties that have not yet been covered by previous work, e.g. characterising the benefits of the subtle protocol refinements.

## References

1. Beeri, C., Bernstein, P.A., Goodman, N.: A model for concurrency in nested transactions systems. *Journal of the ACM* 36(2), 230–269 (1989)
2. Börger, E.: The ASM ground model method as a foundation for requirements engineering. In: *Verification: Theory and Practice*, pp. 145–160 (2003)
3. Börger, E.: The ASM refinement method. *Formal Aspects of Computing* 15, 237–257 (2003)

4. Börger, E., Stärk, R.: Abstract State Machines. Springer, Heidelberg (2003)
5. Del Castillo, G., Gurevich, Y., Stroetmann, K.: Typed abstract state machines (1998), <http://research.microsoft.com/~gurevich/Opera/137.pdf>
6. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishing, San Francisco (1993)
7. Gurevich, J., Sopokar, N., Wallace, C.: Formalizing database recovery. *Journal of Universal Computer Science* 3(4), 320–340 (1997)
8. Kirchberg, M., Schewe, K.-D.: A comparison of multi-level concurrency control protocols. In: Orlowska, M.E., Roddick, J.F. (eds.) *Database Technologies: Proceedings of the 12th Australasian Database Conference (ADC 2001)*. Australian Computer Science Communications, vol. 23(2), pp. 153–160. IEEE Computer Society, Los Alamitos (2001)
9. Kirchberg, M., Schewe, K.-D., Tretiakov, A., Wang, R.: A multi-level architecture for distributed object bases. *Data and Knowledge Engineering* 60(1), 150–184 (2007)
10. Lewis, P.M., Bernstein, A.J., Kifer, M.: *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, Reading (2001)
11. Link, S., Schewe, K.-D., Zhao, J.: Refinements in typed abstract state machines. In: Virbitskaite, I., Voronkov, A. (eds.) *PSI 2006*. LNCS, vol. 4378, pp. 310–321. Springer, Heidelberg (2007)
12. Prinz, A., Thalheim, B.: Operational semantics of transactions. In: Schewe, K.-D., Zhou, X. (eds.) *Database Technologies 2003: Fourteenth Australasian Database Conference. Conferences in Research and Practice of Information Technology*, vol. 17, pp. 169–179 (2003)
13. Schellhorn, G.: ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theoretical Computer Science* 336(2-3), 403–435 (2005)
14. Schewe, K.-D., Ripke, T., Drechsler, S.: Hybrid concurrency control and recovery for multi-level transactions. *Acta Cybernetica* 14(3), 419–453 (2000)
15. Speer, J., Kirchberg, M.: D-ARIES: A distributed version of the ARIES recovery algorithm. In: Eder, J., Haav, H.-M., Kalja, A., Penjam, J. (eds.) *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, pp. 13–30. Tallinn University of Technology Press (2005)
16. Weikum, G.: Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems* 16(1), 132–180 (1991)
17. Zamulin, A.V.: Typed Gurevich machines revisited. *Joint Bulletin of NCC and IIS on Computer Science* 5, 1–26 (1997)
18. Zhao, J., Ma, H.: ASM-based design of data warehouses and on-line analytical processing systems. *Journal of Systems and Software* 79, 613–629 (2006)

# Validating and Animating Higher-Order Recursive Functions in B

Michael Leuschel<sup>1</sup>, Dominique Cansell<sup>2</sup>, and Michael Butler<sup>3</sup>

<sup>1</sup> Institut für Informatik, Universität Düsseldorf  
leuschel@cs.uni-duesseldorf.de


<sup>2</sup> LORIA, Nancy  
cansell@loria.fr

<sup>3</sup> School of Electronics and Computer Science, University of Southampton  
mjb@ecs.soton.ac.uk

**Abstract.** PROB is an animation and model checking tool for the B Method, which can deal with many interesting specifications. Some specifications, however, contain complicated functions which cannot be represented explicitly by a tool. We present a scheme with which higher-order recursive functions can be encoded in B, and establish soundness of this scheme. We then describe a symbolic representation for such functions. This representation enables PROB to successfully animate and model check a new class of relevant specifications, where animation is especially important due to the involved nature of the specification.

**Keywords:** B-Method, Tool Support, Model Checking, Animation, Logic Programming, Constraints 

## 1 Introduction

The B-method, originally devised by J.-R. Abrial , is a theory and methodology for formal development of computer systems. It is used by industries in a range of critical domains. B specifications are structured into *abstract machines*. The state of an abstract machines is represented by variables and correctness conditions can be expressed in the *invariant*, which is specified in predicate logic augmented with set theory and arithmetic. In addition to variables, B machines may also contain constants, which must satisfy conditions expressed in the “properties” clause. Operations of a machine are specified as *generalised substitutions*, which allow deterministic and non-deterministic state transitions to be specified.

There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by tools, such as Atelier-B, B4Free, and the B-toolkit.

In addition to the proof activities it is increasingly being realised that validation of the initial specification is important to avoid deriving a “correct”

---

<sup>1</sup> This research is being carried out as part of the EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

implementation of an incorrect specification. This validation can come in the form of *animation*, e.g., to check that certain functionality is present in the specification. Another useful tool is *model checking* [6], whereby the specification can be systematically checked for certain temporal properties. In previous work [9], we have presented the PROB animator and model checker to support those activities. The tool can also be used to complement proof activities, as it supports automated consistency checking of B machines and has been recently extended for automated refinement checking [10].

**Motivation.** The PROB tool has been successfully applied to various academic and industrial examples (e.g., a Volvo vehicle function [9]). PROB can deal with B's data structures, such as relations, functions and sequences as well as set comprehensions and lambda abstractions. PROB can also handle constants and the first step of animating or model checking a B model then consists of finding values for the constants which satisfy the PROPERTIES clause. To avoid naïve enumeration of possible values for the constants, PROB uses various mechanisms to propagate (partial) information about the possible values of the constants. Still, in the end, the constants will be represented explicitly inside PROB. This is not a problem for some models: for example, the railway model in [4] based on a requirements document from Siemens Transportation Systems, can be animated and model checked: the constants represent, amongst others, the underlying rail network topology. Some specifications, however, contain complicated functions or sets which cannot be represented explicitly. Take the following recursive function over sequences of sequences, coming from an industrial case study [12]:

```

removeDuplicates = {ss,rs | ss: seq(seq(PLACE)) & rs:seq(seq(PLACE))
  & (ss=<> => rs=<>) &
  (card(ss)=1 => rs=ss) &
  (card(ss)>1 => ( #(s1,s2).(s1:seq(PLACE) & s1=first(ss)
    & s2:seq(PLACE) & s2=ss(2) &
    (last(s1)= first(s2) =>
      rs = front(s1) -> removeDuplicates(tail(ss)) )
    & (last(s1)/=first(s2) =>
      rs = s1 -> removeDuplicates(tail(ss)))
    )))
  & removeDuplicates: seq(seq(PLACE)) --> seq(seq(PLACE))
}

```

Intuitively, the above function takes a sequence of sequences and removes duplicates (i.e., the last element of a sequence is identical to the first element of the next sequence). This is represented as a recursive function, and is part of a larger algorithm.

Validating such specifications is especially important, since they are particularly error prone and may contain crucial computational aspects of a software system. Such specifications also pose a major challenge to animation and model checking. Indeed, completely expanding these functions or sets is prohibitively expensive or even impossible. Supposing that  $card(PLACE) = p$  and supposing that we set a maximum length  $m$  for sequences, the above function contains

$\frac{q^{m+1}-1}{q-1}$  maplets, where  $q = \frac{p^{m+1}-1}{p-1}$ . With  $p = 3$  and  $m = 4$  this gives rise to 216, 145, 205 pairs which need to be pre-computed. Assuming that every maplet requires 25 bytes of storage on average, we obtain a memory requirement of over one Gigabyte; and with  $p = 4$ ,  $m = 4$  we exceed 315 Gigabytes.

In this paper we try to overcome this problem, and enable PROB to animate and model check such specifications by using a symbolic representation. The main contributions of the paper are as follows:

- A method and implementation to symbolically store set comprehensions and lambda abstractions within an animator and model checker, without having to expand them into an explicit form.
- A sound scheme whereby recursive higher-order functions can be encoded in B, along with proof obligations to ensure well-definedness.
- A method and implementation to animate and model check specifications containing such functions. The central idea is to extend the above symbolic form with a way to encode the recursion. Only when a recursive function is actually applied to some arguments, is the symbolic form evaluated (for that particular argument).

We also provide some empirical results as well as some possible applications of our technique.

## 2 Symbolic Representation of Sets of Values

We will first study how to represent non-recursive functions (and more generally set comprehensions) symbolically.

**ProB and Explicit Value Representation.** PROB uses an explicit representation for the values of machine variables, constants, etc. The following table shows how the basic B’s data structures are encoded by the PROB Kernel:

B Type	B value	Prolog encoding
INTEGER	5	<code>int(5)</code>
BOOL	TRUE	<code>term(bool(1))</code>
element of a SET $S$	$C$	<code>fd(3, 'S')</code>
pair (*)	$2 \mapsto 5$	<code>(int(2), int(5))</code>
set (POW)	$\{2, 5\}$	<code>[int(2), int(5)]</code>

For example, the set comprehension  $\{x \mid x \in 1..3\}$  gets expanded into  $\{1, 2, 3\}$  (or in the Prolog representation: `[int(1), int(2), int(3)]`). The lambda abstraction  $\lambda x.(x \in 1..3 \mid x * x)$  gets expanded into the set of pairs  $\{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 9\}$ . Note that internally, a lambda abstraction is simply translated into a set comprehension and then expanded. E.g.,  $\lambda x.(x \in 1..3 \mid x * x)$  gets translated into  $\{x, res \mid x \in 1..3 \wedge res = x * x\}$ .

**Symbolic Values.** In this paper we introduce an additional symbolic representation for set comprehensions (and thus also lambda abstractions). The idea is that under certain circumstances a set comprehension is *not* expanded out but kept symbolically. Some of the B operators inside the PROB kernel are then

extended to deal with those symbolic representations. Suppose for example that the properties of a B machine contains the predicate  $s = \{x \mid x * x \in 1..99\}$  and that at some point we wish to evaluate the predicate  $9 \in s$ . This can simply be done by replacing  $x$  inside the set comprehension by the value 9 and then checking the predicate inside the set comprehension, i.e., in this case  $9 * 9 \in 1..99$ . Similarly, if the properties (or an initialisation or an operation body) contain the predicate  $sqr = \lambda x.(x : INTEGER \mid x * x)$  and if at some point we need to evaluate  $sqr(8)$ , we need to replace the parameter  $x$  with the actual value 8 and compute the body  $8 * 8$  of the lambda abstraction. Other operators, however, may require the complete set of values and hence require the expansion of the set comprehension, e.g., if we encounter  $s \cap 8..10 \neq \emptyset$  we need to expand  $s$  to check if it has a value in common with  $8..10$ .

**Closures.** Which form should the symbolic representation take? Naively, one may think that to represent a set like  $\{y \mid P\}$  with  $y$  being of type INTEGER, we could just store the arguments, the types of the arguments, and a representation 'P' of  $P$  inside a Prolog term such as: `symbolic_set([y],[INTEGER],'P')`. A similar scheme could be used for lambda expressions, by translating them into a set comprehension first.

There is, however, one problem with this approach, which the following example illustrates:

```
MACHINE FunPlus
VARIABLES x, fun
INVARIANT x:INTEGER & fun: INTEGER --> INTEGER
INITIALISATION x:=0 || fun:= %y.(y:INTEGER|0)
OPERATIONS
  Inc = x := x+1;
  Set_fun_plus_x = fun := %y.(y:INTEGER| x+y);
  cc <-- ApplyFun(y) = PRE y:INTEGER THEN cc := fun(y) END
END
```

Assume that we start out by executing the operation `Set_fun_plus_x`. Here the variable `fun` would be given the value `symbolic_set([y,res],[INTEGER,INTEGER], 'res=x+y')`. After applying the `Inc` operation, the variable `x` is updated, which would implicitly change the function `fun` represented by our symbolic set! This is of course incorrect, as in the above B machine `Inc` does *not* change the function `fun`.

The solution lies with the *closure* concept, familiar from programming language implementation (see, e.g., [15]) in general and functional programming in particular. It is used when procedures or functions can be used as values. A closure of a function combines the source code of the function together with the current values of the global variables it refers to. This is also called *environment capture*, as the environment is packaged up together with the function.

In PROB this is achieved by compiling a set comprehension or lambda abstraction into a closure; where all references to machine variables are compiled into the code. This is achieved by replacing a reference to a variable `x` by a

special construct `value(V)` where  $V$  is the value of  $x$  at the point of construction of the lambda abstraction or set comprehension. Thus, the symbolic representation of `%y.(y:INTEGER|x+y)`, supposing that  $x$  has the value 2, would be: `closure([y,res],[INTEGER,INTEGER], 'res=value(int(2))+y')`. The value indicates to the PROB kernel that this is not an expression that needs to be interpreted (it is already in the internal representation)<sup>2</sup>

**Implementation.** We have implemented the symbolic representation inside PROB's kernel. Several operators were extended to directly work on this symbolic representation: equality ( $=$ ), set membership ( $\in$ ), and function application ( $f(\cdot)$ ). If a symbolic representation is used with any other B operator the symbolic closure is expanded into an explicit form (where the expansion is delayed until all the free variables of the set comprehension have been given a value). We plan to extend the kernel for further operators; but for the case studies so far, extending the above operators was sufficient. The user can set a boolean preference value to indicate whether set comprehensions and lambda abstractions should be stored symbolically if possible. In future we plan to add a more fine-grained control, whereby each individual set comprehension can be treated differently (indeed, in some cases it is more efficient to expand a symbolic representation once and for all).

We have also applied the same scheme for certain other expressions which are likely to yield big sets: Cartesian products ( $A \times B$ ), powerset constructions ( $\mathbb{P}(A)$ ), sets of relations ( $A \leftrightarrow B$ ) and sets of functions ( $A \leftrightarrow B, A \rightarrow B, \dots$ ). Those expressions are usually used for typing and rarely need to be expanded out. For example, given a predicate  $r \in \mathbb{P}(NAT \leftrightarrow NAT)$  and a value of 3 for `MAXINT`<sup>3</sup> the set  $\mathbb{P}(NAT \leftrightarrow NAT)$  has a cardinality of  $2^{2^{3+4}} = 2^{65536}$ , and even with `MAXINT` of just 2 we have a cardinality of  $2^{5^{12}} \approx 1.34 * 10^{154}$ . Thus,  $\mathbb{P}(NAT \leftrightarrow NAT)$  cannot possibly be stored explicitly, while it is relatively straightforward to check if a given relation  $r$  is a member of the set. A separate user preference indicates whether the symbolic representation should be applied for those type expressions (but there should be little need to switch off the symbolic representation for those expressions).

**Correctness.** By storing a set comprehension  $\{x_1, \dots, x_n \mid P\}$  symbolically inside an animator or model checker for B, we implicitly assume two things:

1. that the set comprehension exists,
2. that only a single value for the set comprehension exists

In this section we do not yet study recursion, i.e., we assume that the predicate  $P$  makes no reference to the set itself (i.e., we do *not* yet consider definitions of the form  $s = \{x \mid x \in s\}$  or  $s = \{x \mid x \notin s\}$ ). Without recursion, the

<sup>2</sup> Also, if the value of  $x$  is not yet known by the kernel then the `value` constructor will contain a Prolog variable, which will be instantiated as soon as  $x$  becomes known (e.g., through enumeration or through evaluation of some other predicate).

<sup>3</sup> `NAT` represents the implementable natural numbers from 0 to `MAXINT`.



set comprehension must exist, but it could of course be empty. For example,  $\{x \mid x \in INT \wedge 1 = 2\}$  is equal to the empty set. Also, there can only be a single solution, otherwise we would have found values for the comprehension parameters for which  $P$  is both true and false; this cannot be.

Hence, in the non-recursive case, the correctness of our approach is unproblematic. However, some interesting real-life specifications require recursive definitions, and we tackle those in the following sections.

### 3 Defining Recursive Functions in B

#### 3.1 The Problems with Recursive Set Comprehensions

In the previous section we have assumed that set comprehensions are not recursive, i.e., that the truth value of the predicate  $P$  inside  $\{x_1, \dots, x_k \mid P\}$  does not depend on the set itself. Let us examine what happens if we drop this restriction. Take for example the two cases:  $st = \{x \mid x \in \mathbb{N} \wedge x \in st\}$  and  $sf = \{x \mid x \in \mathbb{N} \wedge x \notin sf\}$ . The predicates  $x \in \mathbb{N} \wedge x \in st$  and  $x \in \mathbb{N} \wedge x \in sf$  both depend on the values represented by the set comprehensions themselves. In the first case this means that there are multiple solutions for the equation ( $st$  can be any set of natural numbers) while in the second case there is no solution for  $sf$ . Assigning a single symbolic representation to  $st$  would hide non-determinism in the animator and model checker. More seriously, however, assigning a symbolic representation to  $sf$ , even though there is no solution for it, would lead to unsoundness of the animator and model checker.

Note that the set comprehensions  $\{x \mid x \in \mathbb{N} \wedge x \in st\}$  and  $\{x \mid x \in \mathbb{N} \wedge x \notin sf\}$  on their own are not a problem: these expressions have a single value (for any given value of the free variables  $st$  and  $sf$ ). The problem only arises when treating the equations  $st = \{x \mid x \in \mathbb{N} \wedge x \in st\}$  and  $sf = \{x \mid x \in \mathbb{N} \wedge x \notin sf\}$ : we can no longer simply assign the symbolic closure computed for the set comprehension to  $st$  and  $sf$  respectively. Our solution to this problem is as follows:

- identify cases where we can guarantee that a recursive set comprehension has a single solution,
- in those cases, provide a special treatment for recursive set comprehensions and unroll them on demand.

#### 3.2 How to Define Recursive Functions in B

The most common use of recursive definitions is to define constant functions (i.e., the function is a B constant defined in the PROPERTIES clause) which perform computations required by the specification. We will restrict ourselves to such cases in this paper. We will present a way to formally write down such recursive functions in B such that they are well defined in B *and* can be animated and validated by PROB. We will illustrate this using the well-known Factorial

function (see, e.g., [5]), trying to define a constant `factorial` which can be used to compute the factorial of a natural number.

Let us first attempt to use lambda abstractions. Unfortunately, in B, these are not well suited to define recursive functions. Indeed—unlike Z—B has no if-then-else expression<sup>4</sup> and hence no way to provide a base case for the recursion. Hence, one can use a lambda abstraction only as a *part* of the function definition:

PROPERTIES

```
factorial: NATURAL --> NATURAL1 &
factorial = {0|->1} \ / %x.(NATURAL1 | x* factorial(x-1))
```

While this is a possible way to define the factorial function, it is not particularly elegant and well suited for proof. Furthermore, from an implementation point of view, this style would require us to extend set union to be able to combine (recursive) symbolic closures. We have chosen not to pursue this approach. An alternative specification style is to use universal quantification to express the various cases of the function:

PROPERTIES

```
factorial: NATURAL --> NATURAL1 & factorial(0) = 1 &
!x.(NATURAL1 => factorial(x) = x* factorial(x-1))
```

This is already more elegant, and better suited for proof using the B prover tools. However, the definition of the factorial function is “scattered” among different conjuncts of the PROPERTIES clause. It is not obvious how one could translate that into a symbolic closure representation.

Fortunately, it turns out that a set comprehension allows for an elegant specification of the function:

PROPERTIES

```
factorial: NATURAL --> NATURAL1 &
factorial = {x,y| x:NATURAL & y:NATURAL &
  (x=0 => y=1) &
  (x>0 => (y=x* factorial(x-1))) }
```

The advantage over the previous scheme is that the definition of the factorial function now resides within a *single* set comprehension.<sup>5</sup> This will enable us to provide techniques to detect such recursive function definitions and then provide a special symbolic representation for them.

We will of course need to make sure that the recursion is well-founded and progresses towards the base case. Otherwise, we can get a specification like  $f \in \mathbb{N} \rightarrow \mathbb{N} \wedge f = \{x, y \mid x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge y = f(x + 1)\}$  which has multiple solutions (something which we do not want; see the end of Sect. 2). In the above definition of `factorial` we can find a variant ( $x$ ) which ensures that the recursion must terminate.

<sup>4</sup> It only has an if-then-else generalised substitution which cannot be used inside a lambda expressions.

<sup>5</sup> The form is less convenient for proof; we return to this issue later.

There is, however, still one problematic issue with the above solution: the soundness of the recursive definition of the function relies on the preceding declaration  $fact \in \mathbb{N} \rightarrow \mathbb{N}1$ . Without it, we have in principle no guarantee that, for the recursive call  $fact(x - 1)$ , the function is actually defined for  $x - 1$  and that it actually is a function (and not just a relation; see also [3]).

Thus, a more rigorous definition of the function is as follows:

```

MODEL Factorial
CONSTANTS factorial
PROPERTIES
    factorial : NATURAL <-> NATURAL &
    factorial =
        { x,y | x: NATURAL & y: NATURAL &
            (x=0 => y=1) &
            (x>0 & x-1:dom(factorial) => #z.(x-1|->z:factorial& y=x*z))}
END
    
```

It turns out that from this model, we can completely prove within B4Free, the following theorems:

```

THEOREMS
    !P.(P <: NATURAL & 0 : P & succ[P] <: P => NATURAL <: P);
    dom(factorial)=NATURAL;
    factorial : NATURAL --> NATURAL1;
    factorial(0)=1 & !x.(x: NATURAL1 => factorial(x)=x*factorial(x-1));
    factorial = { x,y | x: NATURAL & y: NATURAL &
        (x=0 => y=1) &
        (x>0 => y=x*factorial(x-1))}
    
```

The first theorem establishes the induction principle over natural numbers; it can be proven by contradiction. We then proceed to prove that the domain of the factorial is the set of natural numbers and that we have defined a total function. (There is thus no need to declare factorial to be a total function; this follows mathematically from the way the function is defined.) From that we can further deduce two alternate formalisations of the factorial function: one well suited for proof with the B prover tools and one close to a functional programming style and well suited for animation. We have thus established full correctness of our initial description as a set comprehension.

The crucial proof is for the theorem `factorial: NATURAL --> NATURAL1` (i.e., factorial is a total function). B4free splits this into two subsidiary predicates `dom(factorial) = NATURAL` (factorial is total) and `factorial:NATURAL +--> NATURAL1` (factorial is a partial function). Both predicates are proved by induction as in [5]. To prove the totality of factorial we have instantiated the inductive theorem with `dom(factorial)` and to prove the partial functionality of factorial we have instantiated the inductive theorem with the set

```
{n | n : NATURAL & 0..n <| factorial : 0..n --> NATURAL1}.
```

Remark that we prove also the totality between `0..n` for convenience reasons, otherwise we need to consider (for the inductive step) two cases: `n:dom(factorial)` and `n/:dom(factorial)`.

### 3.3 A General Scheme

Inspired by the factorial example, we now proceed to present a general scheme for defining recursive functions in B. To ensure well-foundedness, we will require a variant which for simplicity we will assume to be of type natural. In our general scheme, a recursive function with  $n$  arguments and  $N$  cases is defined as follows:

$$f = \{x_1, \dots, x_n, out \mid x_1 \in T_1 \wedge \dots \wedge x_n \in T_n \wedge P(x_1, \dots, x_n) \wedge out \in T_{n+1} \wedge \\ (Cond_1 \Rightarrow out = Exp_1) \wedge \\ \dots \wedge \\ (Cond_N \Rightarrow out = Exp_N)\}$$

In this scheme each  $Cond_i$  has free variables included in  $\{x_1, \dots, x_n\}$  and where  $Exp_i$  can also make reference to  $f$ . The scheme must include a variant function  $V \in T_1 \times \dots \times T_n \rightarrow \mathbb{N}$ .

We introduce the following abbreviation:

$$ArgType \equiv x_1 \in T_1 \wedge \dots \wedge x_n \in T_n \wedge P(x_1, \dots, x_n)$$

Below we will formalise the conditions under which this scheme ensures that  $f$  defines a total function over  $T_1 \times \dots \times T_n$ .

We first ensure that for any input value in  $T_1 \times \dots \times T_n$  we have one and exactly one condition  $Cond_i$  that is true:

1.  $\forall(x_1, \dots, x_n).(ArgType \Rightarrow (Cond_1 \vee \dots \vee Cond_N))$
2. for any  $i \neq j$ :  $\forall(x_1, \dots, x_n).(ArgType \Rightarrow \neg(Cond_i \wedge Cond_j))$

The variant must always be a natural number and must always be decreased by each recursive reference:

3.  $\forall(x_1, \dots, x_n).(ArgType \Rightarrow V(x_1, \dots, x_n) \geq 0)$
4. for any  $i \in 1..N$  and for any recursive call  $f(e_1, \dots, e_n)$  inside  $Exp_i$ :  
 $\forall(x_1, \dots, x_n).(ArgType \wedge Cond_i \Rightarrow 0 \leq V(e_1, \dots, e_n) < V(x_1, \dots, x_n)).$

More precisely the conditions will ensure that

- A. Each  $(x_1, \dots, x_n)$  is in the domain of  $f$ :

$$\forall(x_1, \dots, x_n).(ArgType \Rightarrow (x_1, \dots, x_n) \in dom(f))$$

- B. Each  $(x_1, \dots, x_n)$  is mapped to at most one range value:

$$\forall(x_1, \dots, x_n, y_1, y_2).(ArgType \wedge (x_1, \dots, x_n) \mapsto y_1 \in f \wedge (x_1, \dots, x_n) \mapsto \\ y_2 \in f \Rightarrow y_1 = y_2)$$

The proof of A and B is by general induction over the range of variant function  $V$ , that is, assuming that the property holds for all  $(x'_1, \dots, x'_n)$  where  $V(x'_1, \dots, x'_n) < V(x_1, \dots, x_n)$  show that it holds for  $(x_1, \dots, x_n)$ .

## 4 Implementation: Recursive Closures

We now show how the implementation scheme of Sect. 2 can be adapted to deal with recursive set comprehensions. For this we introduce a second symbolic representation, namely for recursive closures.

The first part of our implementation is the extension in the PROB kernel of the equality Prolog predicate. This Prolog predicate gets called whenever an equality between two B objects needs to be checked. If none of the objects is a symbolic closure then the check proceeds as usual [9]. If both objects are closures, then they are both expanded and checked for equality. Let us now assume that one of the objects is a closure  $C$ , as explained in Sect. 2, while the other (say  $f$ ) is not. (This situation would arise for the predicate  $f = \{x_1, \dots, x_n \mid P\}$ , unless  $f$  was already given a symbolic closure by some preceding predicate.) There are now three cases for  $f$ :

- $f$  has some value associated with it (or is partially instantiated); in this case the closure for the set comprehension gets expanded and the equality is checked as usual.
- $f$  is unconstrained (i.e., apart from the type, nothing is known about  $f$ ) but does not appear in  $P$ : in this case  $f$  is now set to be equal to the symbolic closure  $C$ .
- $f$  is unconstrained and does appear in  $P$ : in this case we have a recursive set comprehension. There two sub-cases:
  - The equality  $f = \{x_1, \dots, x_n \mid P\}$  does not conform to the scheme outlined in Sect. 3.3. In this case the Prolog predicate needs to delay until  $f$  is completely known, at which point the set comprehension can be expanded and the equality checked.
  - Otherwise we generate a new identifier ID to identify the recursive function and replace all occurrences of  $f$  inside the  $C$  with  $\text{rec}(\text{ID})$ , yielding  $C'$ . The variable or constant  $f$  now gets as value the symbolic representation  $\text{recursive\_closure}(\text{ID}, C')$ .

The equality predicate is also the only place which can lead to the generation of a new recursive closure. For our `factorial` constant above we would thus get as symbolic representation:

```
recursive_closure(1, closure([x,y], [INTEGER, INTEGER],
  'x:NATURAL & y:NATURAL & (x=0 => y=1) &
  (x>0 => y=x*value(rec(1))(x-1))' ))
```

These recursive closures are then unrolled on demand. More precisely, when a recursive closure  $\tau = \text{recursive\_closure}(i, C)$  is examined (e.g., by the function application Prolog predicate) it is converted into the closure  $C' = C[\tau/\text{rec}(i)]$ , i.e.  $C$  where all  $\text{rec}(i)$  have been replaced by  $\tau$ .

For example, unrolling the above recursive closure yields:

```
closure([x,y],[INTEGER,INTEGER],
  'x:NATURAL & y:NATURAL & (x=0 => y=1) &
  (x>0 => y=x*value(recursive_closure(1,...))(x-1))' )
```

If the first argument is 0 then no further unrolling is required, but if  $x > 0$  the inner recursive closure will be unrolled, etc., until we reach the base case of the recursion.

Note that this way to handle recursion is related to the *fix* operator sometimes used in process algebras (see, e.g., [11]).

## New Syntax

The introduction of a new syntax for recursive functions can provide both an effective way to animate recursive functions as well as a convenient way to prove properties with and about them. This would require extending the PROB parser and then either convert the syntax into a form suitable for proving or suitable for PROB for animating and model checking. A possible syntax for the factorial function could be:

```
FUNCTIONS
  y <-- factorial(x) = WHERE x:NAT & y:NAT THEN
                        CASE x=0 THEN y=1
                        CASE x>0 THEN y=fact(x-1)
                        VARIANT x
  END
```

We have not yet finalised the new syntax, and in the remainder of the paper we carry out our experimentation with the set comprehension style suitable for animation.

## 5 Higher-Order Functional Programming Examples in B

So far we have shown that first-order recursive functions can be encoded within B, and animated and validated by PROB. As it turns out, our scheme is actually powerful enough to animate higher-order recursive functions. In other words, we have actually developed a scheme to enable higher-order functional “programming” inside B specifications. In this section we provide a few examples to illustrate this point.

**Mutual Recursion.** Before examining higher-order functions, let us first discuss the issue of mutual recursion. The examples so far contained a single recursive function. Does our scheme also work for several mutually recursive functions? The answer to this question is affirmative: the proof obligations need to be adapted to handle multiple functions (in a straightforward fashion), but the implementation scheme is already suited to handle such functions.

Take the following artificial example, splitting the factorial function into two mutually recursive functions:

```

CONSTANTS fact1,fact2
PROPERTIES
  fact1: INT --> INT &
  fact1 = {x,y | x:NAT & y:NAT &
            (x=0 => y=1) & (x>0 => (y=x*fact2(x-1))) } &
  fact2: INT --> INT &
  fact2 = {x,y | x:NAT & y:NAT &
            (x=0 => y=1) & (x>0 => (y=x*fact1(x-1))) }

```

In this case, `fact1` will be stored as a standard closure (calling `fact2`) and `fact2` will be a recursive closure with no reference to `fact1`. Note that we can also deal with the problematic example discussed in [8].

**Higher-Order Functional Programming.** Some higher-order programming is actually already built into B: to *map* a function  $f$  over a sequence  $s$  we simply need to use the relational composition  $(s; f)$ , as  $s$  is a function from  $1..size(s)$  to  $ran(s)$ . In general, however, this is not so easy. Below we show how we can specify the well-known “`foldr`” higher-order function, which takes a base value and a function  $f$  and maps it over a sequence to compute a single value. In the `FoldMul` operation we use this higher-order function to compute the product of the elements of a sequence.

```

MACHINE SeqFoldr
CONSTANTS mul, foldr
PROPERTIES
  mul: (NATURAL*NATURAL)<->NATURAL &
  mul = {i,j,res | i:NATURAL & j:NATURAL & res:NATURAL & res=i*j} &
  foldr:(((NATURAL*NATURAL)<->NATURAL)*NATURAL*seq(NATURAL))<->NATURAL &
  foldr =
    { f,base,i,res | i:seq(NATURAL) & base:NATURAL &
                    res: NATURAL & f:(NATURAL*NATURAL)-->NATURAL &
                    (i<> => res=base) &
                    (size(i)>0 => res = f(first(i),foldr(f,base,tail(i))) )
    }
VARIABLES ss
INVARIANT ss: seq(NATURAL)
INITIALISATION ss := <>
OPERATIONS
  Add(nn) = PRE nn:NATURAL THEN ss := ss <- nn END;
  FoldMul = BEGIN ss := foldr(mul,1,ss) -> ss END
END

```

We can easily show that `foldr` satisfies our proof obligations, if we choose  $size(i)$  as the variant:

1.  $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow (i = \langle \rangle \vee size(i) > 0))$
2.  $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow \neg(i = \langle \rangle \wedge size(i) > 0))$
3.  $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \Rightarrow size(i) \geq 0)$
4.  $\forall(f, i).(i \in seq(\mathbb{N}) \wedge f \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \wedge size(i) > 0 \Rightarrow size(tail(i)) < size(i))$

One difference with functional programming still persists though: `foldr` is usually defined to be a polymorphic function, i.e., the type of the elements of the list is not hardcoded like in our B machine. To overcome this, one has to define `foldr` within a separate machine, whose argument is the type of the elements of the list. Unfortunately, the machine parameters are not visible inside the `PROPERTIES` clause; hence we actually need to make `foldr` a variable (which is not very convenient):

```
MODEL Foldr(TYP)
VARIABLES foldr
INVARIANT foldr : (((TYP* TYP)--> TYP)*TYP*seq(TYP)) --> TYP
INITIALISATION
  foldr : (foldr = { f,b,i,res |
              i:seq(TYP) & res:TYP & f:(TYP* TYP)--> TYP & b:TYP &
              (i=<> => res=b) &
              (size(i)>0 => res = f(first(i), foldr(f,b,tail(i))) ) } )
END
```

## 6 Empirical Results

The experiments were all run on a multiprocessor system with 4 AMD Opteron 870 Dual Core 2 GHz processors, running SUSE Linux 10.1, SICStus Prolog 3.12.7 (x86\_64-linux-glibc2.3) and PROB version 1.2.4<sup>6</sup>

The first experiment consisted in running the previously discussed `factorial` function. The results are presented in the upper half of Table 1. Note that PROB does check the function arguments (to see if they are a natural number) at every function application. Also, note that without our new symbolic approach, the present formalisation cannot be animated (even for small values of  $n$ ). However, using the axiomatic formalisation from Sect. 3.2 (using universal quantification) it is possible, provided we limit the domain of the function. It then takes 0.85 sec to compute factorial for 0..100 using classical PROB. (For functions such as `SeqFoldr` as seen earlier, it is of course impossible to precompute the function.)

In order to measure a specification requiring a large number of recursive calls we have used the naïve recursive definition for computing the Fibonacci numbers: `fib = {x,z| x:NATURAL & z:NATURAL & (x=0 => z=1) & (x=1 => z=1) & (x>1 => (z=fib(x-1)+fib(x-2))) }`. The results are summarised in the same Table 1. For `Fib(20)` we have 21891 ( $2 \times fib(20) - 1$ , see, e.g., [13]) calls to the Fibonacci function. This corresponds to 3357 calls per second. For a programming language this would of course be very slow (even though PROB works with big integers); but for animation purposes this is actually quite reasonable (also given the fact that the typing predicates are repeatedly evaluated). However, there is definitely scope for improvement. Possibly with the use of partial evaluation [7] and more sophisticated implementation techniques, a big improvement in speed should be possible. Still, in its current form the tool can be used to animate a wide range of specifications with recursive functions. In particular, PROB has been

<sup>6</sup> Note that neither SICStus Prolog nor PROB take advantage of multiple processors.



**Table 1.** Empirical Results

n	factorial(n)	Time	Function calls per sec
5	120	0.00 sec	-
10	3,628,800	0.00 sec	-
20	2,432,902,008,176,640,000	0.02 sec	1000
100	see footnote <sup>7</sup>	0.06 sec	1667

n	fib(n)	Time	Function calls per sec
5	8	0.01 sec	1500
10	89	0.10 sec	1770
15	987	0.67 sec	1999
20	10946	6.52 sec	3357

successfully applied to an industrial case study (cf. Section [11](#)) which was hitherto impossible to animate or model check, and was able to detect an error in the original specification [\[12\]](#).

## 7 Related Work and Conclusion

While there are various other animators for B and Z, to our knowledge, none of them can handle recursive functions.

In [\[2\]](#) authors explain how we can specify higher order expression and theorems using B and how we can prove such theorems using B tools. The second work [\[5\]](#) is more related to our work. The `factorial` function is also defined first like the smallest relation which satisfies `factorial`'s properties and then the proof of the functionality of `factorial` is done using B4free as in our work. This definition is not suited for animation but some algorithms to compute the `factorial` function are given. These algorithms are developed using B and the refinement. These are correct by construction. The first abstract model computes `factorial(n)` in one shot. The first refinement computes a finite subset of the `factorial` function like in dynamic programming. The last refinement computes `factorial(n)` using the well known loop from 1 to n. Another related work is [\[14\]](#), which presents a framework to reconcile axiomatic and model-based specifications. As such it is related to our desire at the end of Sect. [4](#) to present two different views of a specification: one suitable for proving and one suitable for animation. In the context of higher-order logic and Isabelle/HOL, [\[8\]](#) presents a method to reason about recursive functions.

In summary, we have presented a scheme to define higher-order recursive functions in B and have shown how we can animate and model check them using extensions to the ProB toolset. We have carried out various experiments, showing the practicality of the approach.

<sup>7</sup> For `factorial(100)` the result computed by ProB is:

933262154439441526816992388562667004907159682643816214685929638952175999932299156089  
414639761565182862536979208272237582511852109168640000000000000000000000000.

**Acknowledgements.** We would like to thank Daniel Plagge and Jens Bendisposto for very useful comments and discussions. We would also like to thank the anonymous referees for their insightful suggestions and pointers to related work.

## References

1. Abrial, J.-R.: *The B-Book*. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Cansell, D., Laffitte, G.: Higher-order mathematics in B. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) *B 2002 and ZB 2002*. LNCS, vol. 2272, pp. 370–393. Springer, Heidelberg (2002)
3. Abrial, J.-R., Mussat, L.: On using conditional definitions in formal theories. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) *B 2002 and ZB 2002*. LNCS, vol. 2272, pp. 242–269. Springer, Heidelberg (2002)
4. Butler, M.: A system-based approach to the formal development of embedded controllers for a railway. *Design Automation for Embedded Systems* 6(4), 355–366 (2002)
5. Cansell, D., Méry, D.: Foundations of the B method. *Computing and informatics* 22(3–4), 221–256 (2003)
6. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. MIT Press, Cambridge (1999)
7. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs (1993)
8. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006*. LNCS (LNAI), vol. 4130, pp. 589–603. Springer, Heidelberg (2006)
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. Massart, T., Devillers, R.: Equality of agent expressions is preserved under an extension of the universe of actions. *Formal Aspects of Computing* 5(1), 79–88 (1993)
12. 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)
13. Robertson, J.S.: How many recursive calls does a recursive function make? *SIGCSE Bull.* 31(2), 60–61 (1999)
14. Robinson, K.: Reconciling axiomatic and model-based specifications using the B method. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) *B 2000, ZUM 2000, and ZB 2000*. LNCS, vol. 1878, pp. 95–106. Springer, Heidelberg (2000)
15. Roy, P.V., Haridi, S.: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge (2004)

# A Systematic Verification Approach for Mondex Electronic Purses Using ASMs

Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius,  
and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,  
Universität Augsburg, D-86135 Augsburg, Germany  
{schellhorn,grandy,haneberg,moebius,reif}@informatik.uni-augsburg.de

**Abstract.** In previous work we solved the challenge to mechanically verify the Mondex challenge about the specification and refinement of an electronic purse, using the given data refinement framework. In this paper we show that using ASM refinement and generalized forward simulations instead of the original approach allows to find a more systematic proof. Our technique of past and future invariants and simulations avoids the need to define a lot of properties for intermediate states during protocol runs. The new proof can be better automated in KIV. The systematic development of a generalized forward simulation uncovered a weakness of the protocol that could be exploited in a denial of service attack. We show a modification of the protocol that avoids this weakness, and that is even slightly easier to verify.

## 1 Introduction

Mondex smart cards implement an electronic purse. They have become famous for having been the target of the first ITSEC evaluation of the highest level E6, which requires formal specification and verification.

Such formal specifications were given in [SCW00] using the Z specification language. Two models of electronic purses were defined: an abstract one which models the transfer of money between purses as elementary transactions, and a concrete level that implements money transfer using a communication protocol that can cope with lost messages using a suitable logging of failed transfers. To mechanize the security and refinement proofs in [SCW00] has been recently proposed as a challenge for theorem provers (see [Woo06] for more information on the challenge and its relation to 'Grand Challenge 6').

In [SGHR06a] we have solved the challenge: we have tried to repeat the case study as faithful as possible by formalizing the underlying data refinement theory given in [CSW02] and by using the original backward simulation and invariant. A detailed description of this verification (including the extra protocol for archiving exception logs) is contained in [HSGR07].

When we solved the original challenge, we found that the backward simulation and in particular the invariant needed for the concrete level<sup>1</sup> looked rather ad hoc and specific to Mondex. All corrections that were necessary for the backward simulation proof were due to this invariant: two properties had to be added, one was redundant (for details see [SGHR06a] and the technical report [SGHR06b]). Much more work than the 4 weeks we needed to do the mechanical verification surely was necessary to develop this invariant by incrementally adding properties.

Therefore, in this paper we show how to develop a simulation relation and an invariant systematically. We do this using Abstract State Machines (ASM, [BS03]) as our specification language. In [SGHR06a] we have already given ASMs for Mondex, and shown that the proof for the main backward simulation condition for this ASM is the same as the one for data refinement, but with a lot of technical overhead removed. Therefore we feel justified to use the simplified version here.

Using ASMs naturally leads to the use of ASM refinement ([BR95], [Sch01], [Bör03], [Sch05]) and (generalized) forward simulations.

This paper is organized as follows: Section 2 introduces the ASMs used in the case study, and gives an informal idea why the refinement is correct. Section 3 develops a forward simulation for Mondex systematically using two core ideas of ASM refinement: focussing the simulation relation on *states of interests*, which in this case naturally are those *future* states where all protocols have completed, and *localizing invariants* to individual purses. We show that the main proof obligation, a commuting diagram for a local invariant can be verified fully automatically in KIV.

Development of a systematic invariant for the concrete level turned out to be much harder than the development of a simulation relation. The main reason is that the protocol is vulnerable to a certain kind of denial of service attack described in Section 4. Although the attack does not violate the security properties defined in [SCW00] (no money is lost), it came into sharp focus when we applied ASM techniques to develop an invariant in Section 5. We use *lazy development* for the invariant and show that the main proof only requires a few KIV interactions.

Finally, Section 6 gives related work and Section 7 concludes. Unfortunately, the space limitations of this paper prohibit to give full formal definitions of the simulation relation and the invariant. They can be found in the technical report [SGH<sup>+</sup>06] and the Web presentation [KIV].

## 2 The ASM Specifications of Mondex

In this section we describe the specifications of the smart cards involved in the Mondex case study. To have a self-contained paper, we repeat here a slightly

<sup>1</sup> Formally, the verification of an invariant is encoded in [SCW00] as a second refinement of a between level, that assumes an invariant (the properties of `BetweenWorld`, section 5.3 and the auxiliary properties of purses, section 4.6) to a concrete level, that does not.

modified version of the description given in [SGHR06a]. The changes are purely cosmetic to have shorter formulas in the proof obligations. Detailed information on how the ASMs given here were derived from the original Z specification of [SCW00] are given in [HSGR07].

The specification is given on two levels: an abstract level, which defines an atomic transaction for transferring money, and a concrete level, which defines the protocol. Both levels are defined using abstract state machines (ASMs, [Gur95], [BS03]) and algebraic specifications as used in KIV [RSSB98].

## 2.1 The Abstract Level

The abstract specification of a purse consists of a function `abalance` from purse names to their current balance. Since the transfer of money from one purse to another may fail (i.e. the card being pulled abruptly from the card reader, or for internal reasons like lack of memory), the state of an abstract purse also must log the amount of money that has been lost in such failed transfers.

In the formalism of ASMs this means that the abstract state `astate` consists of two dynamic functions `abalance : name → IN` and `lost : name → IN`.

Purses may be faked, so we have a finite number of names which satisfy a predicate `authentic`. How authenticity is checked (using secret keys, pins etc.) is left open on both levels of the specification, so the predicate is left unspecified.

Transfer of money between purses is done with the ASM rule `ASTEP#2`:

```

ASTEP#
choose from, to, value, fail?
with authentic(from) ∧ authentic(to) ∧ from ≠ to ∧ value ≤ abalance(from)
in if ¬ fail? then TRANSFEROK#
                else TRANSFERFAIL#
ifnone skip //do nothing, if there is no authentic pair of purses

TRANSFEROK#
abalance(from) := abalance(from) – value
abalance(to) := abalance(to) + value

TRANSFERFAIL#
abalance(from) := abalance(from) – value
lost(from) := lost(from) + value

```

The rule chooses two authentic, different names `from` and `to`, and an amount `value` which should be transferred from the `from` purse to the `to` purse. The `from` card must have enough money left for the transfer (`value ≤ abalance(from)`). Additionally, a boolean value `fail?` indicates whether the actual transaction will complete regularly or will nondeterministically fail for internal reasons. If the step completes normally, the rule `TRANSFEROK#` subtracts `value` from the `from` purse and adds it to the `to` purse in one step. Otherwise, the rule `TRANSFERFAIL#`

<sup>2</sup> By convention, rule names end with a # sign to distinguish them from predicates.

subtracts the money from the from purse and logs it in the `lost(from)` state function instead.

### 2.2 The Concrete Level

On the concrete level transferring money is done using a protocol with 5 steps. To execute the protocol, each purse needs a status that indicates how far it has progressed executing the protocol. The possible states a purse may be in are given by the enumeration `status = idle | epr | epv | epa`. Purses not participating in any transfer are in the `idle` state. To avoid replay attacks each purse stores a sequence number `nextSeqNo` that is used in the next transaction. This number is incremented at the start of every protocol run. During the run of the protocol each purse stores the current payment details in a dynamic function `pdAuth` of

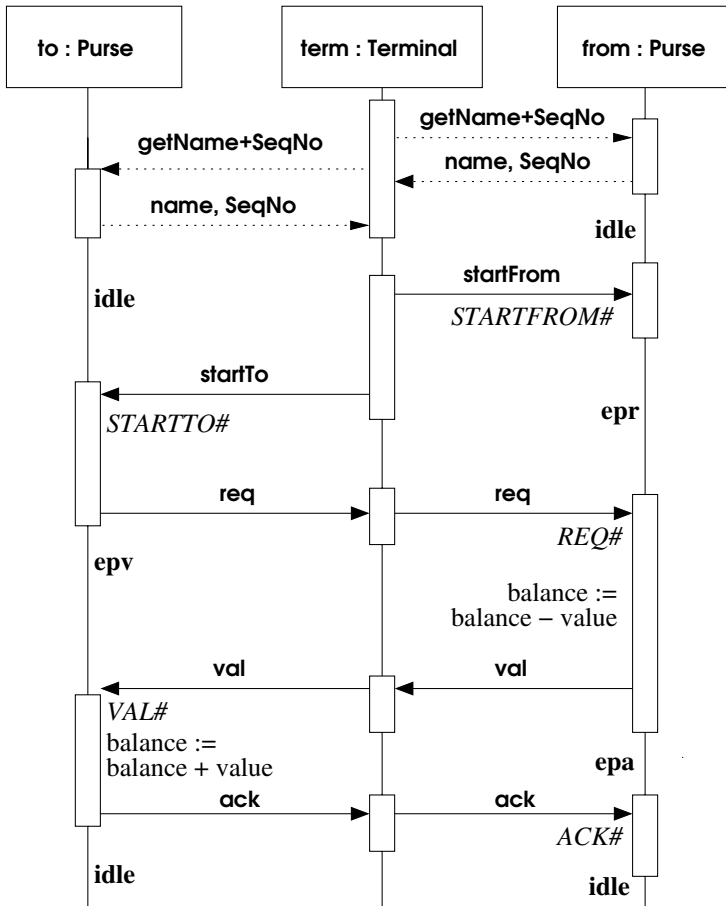


Fig. 1. An overview of the Mondex protocol

type `PayDetails`. These are tuples consisting of the names of the from and to purse, the sequence numbers they use and the amount of money that is transferred. In KIV we define a free data type

```
PayDetails = mkpd(.from:name; .fromno:nat; .to:name; .tono:nat; .value:nat)
```

with postfix selectors, so that `pd.from` is the name of the from purse stored in payment details `pd`. The state of a purse finally contains a log `exLog` of failed transfers represented by their payment details. The protocol is executed by sending messages between the purses. The `ether` collects all messages that are currently available. A purse receives a message by selecting a message from the `ether`. Since the environment of the card is assumed to be hostile the message received may be any message that has already been sent, not just one that is directed to the card. The state of the concrete ASM, abbreviated `cstate` below, is

```
balance : name → ℕ           exLog : name → set(PayDetails)
state   : name → status      ether  : set(message)
pdAuth  : name → PayDetails
```

An overview of the protocol on the concrete level is shown as an UML sequence chart in Fig. 1. The message `getName+SeqNo` (shown by dashed lines) is needed in a real implementation of the Mondex scenario, since the terminal must be able to get the information about card names and their sequence numbers. This information is used in the following protocol steps. For the specification of Mondex, those messages are not modelled, even the terminal itself is not modelled explicitly. Instead, all messages needed to start a protocol run are assumed to be initially contained in a set of messages available to the purses, called the `ether`.

The protocol is started with `startFrom(msgna, value, msgno)` and `startTo(msgna, value, msgno)` messages which are sent to the from and to purses respectively, by the interface device. These two messages are assumed to be always available, so the initial `ether` already contains every such message. The arguments `msgna` and `msgno` of `startFrom(msgna, value, msgno)` are assumed to be the name and `nextSeqNo` of the to purse, where `value` is the amount of value transferred. Similarly, for `startTo(msgna, value, msgno)`, `msgna` and `msgno` are the corresponding data of the from purse. The messages `req(pdAuth(name))`, `val(pdAuth(name))` and `ack(pdAuth(name))` are responsible for the actual money transfer. After receiving a `req`, the from purse withdraws money from its internal balance. After receiving the corresponding `val`, the to purse deposits money on its internal balance. The `ack` message is used to acknowledge a successful transfer between from and to. We now describe the ASM rule `CSTEP#`, which executes all the protocol steps, and the individual protocol steps in detail:

```
CSTEP#
choose msg, receiver fail? with msg ∈ ether ∧ authentic(receiver)
in LCSTEP#
```

```

LCSTEP#
  if isOKstartFrom(msg)  $\wedge$   $\neg$  fail? then STARTFROM#
  else if isOKstartTo(msg)  $\wedge$   $\neg$  fail? then STARTTO#
  else if msg = req(pdAuth(receiver))  $\wedge$  state(receiver) = epr  $\wedge$   $\neg$  fail?
    then REQ#
  else if msg = val(pdAuth(receiver))  $\wedge$  state(receiver) = epv  $\wedge$   $\neg$  fail?
    then VAL#
  else if msg = ack(pdAuth(receiver))  $\wedge$  state(receiver) = epa  $\wedge$   $\neg$  fail?
    then ACK#
  else ABORT#

```

where

```

isOKstartFrom(msg)
: $\leftrightarrow$  isStartFrom(msg)  $\wedge$  state(receiver) = idle  $\wedge$  authentic(msg.msgna)
     $\wedge$  receiver  $\neq$  msg.msgna  $\wedge$  msg.value  $\leq$  balance(receiver)

isOKstartTo(msg)
: $\leftrightarrow$  isStartTo(msg)  $\wedge$  state(receiver) = idle
     $\wedge$  authentic(msg.msgna)  $\wedge$  receiver  $\neq$  msg.msgna

```

The ASM rule CSTEP# chooses an authentic receiver *receiver* for some message *msg* from *ether*. Additionally the purse may fail (e.g. for internal reasons), denoted by the flag *fail?*. The rule LCSTEP# executes the different protocol steps. It checks whether the incoming message is wellformed regarding to the current internal state. For example, when the purse is in state *epr* (“expecting request”), it will only accept messages of the form *req(pdAuth(receiver))*. Every other message received in *epr* will result in an *ABORT#* operation, which resets the purse state and logs the current transaction as faulty if necessary (see below).

On receiving a *startFrom* message *msg* from *ether*, the purse *receiver*<sup>3</sup> first checks whether it is in the *idle* state, and if the message is syntactically correct. This means it must be of the right message type (*isStartTo(msg)*), and the contained card name satisfies *authentic*. Additionally, the transmitted *msg.msgna* (the name of the other purse) must be different from *receiver*. Finally *receiver* must have enough money stored (*msg.value*  $\leq$  *balance(receiver)*) for the transaction to take place. Then, *receiver* executes the following step:

```

STARTFROM#
choose n with nextSeqNo(receiver) < n in
in pdAuth(receiver) := mkpd(receiver, nextSeqNo(receiver),
    msg.msgna, msg.msgno, msg.value)
state(receiver) := epr
nextSeqNo(receiver) := n seq
SENDMSG#( $\perp$ )

```

The purse stores the requested transfer in its *pdAuth* component, using its current *nextSeqNo* number as one component and proceeds to the *epr* state. Thereby

<sup>3</sup> Receiver is always a purse receiving a message. This can be a from purse sending money as well as a to purse receiving money and should not be confused with the latter.



it becomes the *from* purse of the current transaction.  $\text{nextSeqNo}(\text{receiver})$  is incremented, which makes the old value unavailable for further transactions. An empty output message  $\perp$  is generated that will be added to the ether (see  $\text{SENDMSG\#}$  below).

The action of a purse receiving a  $\text{startTo}$  message in *idle* state is similar except that it enters *epv* state (“expecting value”) and becomes the *to* purse of the transaction. Additionally it sends a request message to the *from* purse.

```

STARTTO#
choose n with nextSeqNo(receiver) < n
in pdAuth(receiver) := mkpd(msgna, msgno, receiver,
                           nextSeqNo(receiver), value)
    state(receiver) := epv
    nextSeqNo(receiver) := n seq
    SENDMSG#(req(pdAuth(receiver)))

```

The request  $\text{req}(\text{pdAuth}(\text{receiver}))$  contains the payment details of the current transaction. Although this is not modeled, the message is assumed to be securely encrypted. Since an attacker can therefore never guess this message before it is sent, it is assumed that the initial ether does not contain any request messages. When the *from* purse receives the request in state *epr*, it executes  $\text{REQ\#}$ .

```

REQ#
balance(receiver) := balance(receiver) - pdAuth(receiver).value
state(receiver) := epa
SENDMSG#(val(pdAuth(receiver)))

```

The message is checked to be consistent with the current transaction stored in  $\text{pdAuth}$  and if this is the case the money is sent with an encrypted value message  $\text{val}(\text{pdAuth}(\text{receiver}))$ . The state changes to *epa* (“expecting acknowledge”). On receiving the value the *to* purse executes  $\text{VAL\#}$ .

```

VAL#
balance(receiver) := balance(receiver) + pdAuth(receiver).value
state(receiver) := idle
SENDMSG#(ack(pdAuth(receiver)))

```

This rule adds the money to its balance, sends an encrypted acknowledge message back and finishes the transaction by going back to state *idle*. When this acknowledge message is received, the *from* purse finishes similarly.

```

ACK#
state(receiver) := idle
SENDMSG#( $\perp$ )

```

Finally a rule for adding the sent messages to the ether is needed. Additionally the ether is assumed to lose messages randomly (due to an attacker or technical reasons like power failure). Both is now done in the rule  $\text{SENDMSG\#}$  used in all the rules above.

```

SENDMSG#(outmsg)
choose ether' with ether'  $\subseteq$  ether  $\cup$  {outmsg} in ether := ether'

```

If a purse is sent an illegal message  $\perp$  or a message for which it is not in the correct state, the current transaction is aborted by

```

ABORT#
choose n with nextSeqNo(receiver) ≤ n
in LOGIFNEEDED#
  state(receiver) := idle
  nextSeqNo(receiver) := n
  SENDMSG#(⊥)

LOGIFNEEDED#
if state(receiver) = epa ∨ state(receiver) = epv
then exLog(receiver) := exLog(receiver) ∪ {pdAuth(receiver)}

```

This action logs if money is lost due to aborting a transaction. The idea is that the lost money of the abstract level can be recovered if both the *from* and *to* purses have a log of the failed transaction. Logging takes place if either the purse is a *to* purse in the critical state *epv*, or a *from* purse in the critical state *epa*. Note that aborting in states *idle* and *epr* requires no exception log. Logging achieves, that in states where all purses are currently *idle*, balances and the lost money are related by

$$\begin{aligned} \text{abalance}(\text{na}) &= \text{balance}(\text{na}) \\ \wedge \text{lost}(\text{na}) &= \Sigma (\text{fromLogged}(\text{na}) \cap \text{toLogged}) \end{aligned} \quad (1)$$

where

```

fromLogged(na) := {pd : pd ∈ exlog(na) ∧ pd.from = na},
toLogged(na)   := {pd : pd ∈ exlog(na) ∧ pd.to = na} and
toLogged      := ∪na toLogged(na)

```

and where  $\Sigma$  takes the sum of all values of a set of payment details. For future use,  $\text{fromLogged} := \bigcup_{\text{na}} \text{fromLogged}(\text{na})$ .

### 3 Systematic Development of a Forward Simulation

One of the key ideas of ASM refinement (and also verification of ASM invariants) is *not* to consider all intermediate states of runs of the ASM but to focus on *states of interest* and to define properties  $\varphi$  only for these. In general, there is a choice to use future or past states. Either we can say:

- Future: From every state a state of interest that will satisfy  $\varphi$  is reachable via some ASM rule applications
- Past: Every state is reachable from some state of interest that satisfies  $\varphi$

Applied to Mondex the states of interest are states where a purse with name *na* does not participate in a protocol, i.e. where  $\text{state}(\text{na}) = \text{idle}$ . For the development of the simulation relation, we will consider future states of interest, in section 5 we will develop the invariant based on past states of interest.

There are two problems we have to solve. First, what is the simulation relation for states of interest? This question was already answered by formula (II) at the end of the previous section. Second, we have to show how a state of interest is reachable from any state. For the concrete level this is easy: simply call **ABORT#** for all purses. To have an equivalent state on the abstract level we must execute failing transactions for all those purses where money will be lost on the concrete level by executing **ABORT#**. This money, which is currently in transit, can be characterized by the set of relevant payment details. This set was already central to the correctness consideration of the original Mondex case study [SCW00]. It is called *maybelost*, and is defined as

$$\text{maybelost} := (\text{fromInEpa} \cap \text{toInEpv}) \cup (\text{fromInEpa} \cap \text{toLogged}) \\ \cup (\text{fromLogged} \cap \text{toInEpv})$$

where

$$\text{fromInEpa} = \{\text{pdAuth}(\text{na}) : \text{authentic}(\text{na}) \wedge \text{state}(\text{na}) = \text{epa}\} \\ \text{toInEpv} = \{\text{pdAuth}(\text{na}) : \text{authentic}(\text{na}) \wedge \text{state}(\text{na}) = \text{epv}\}$$

The definition is based on the idea that money is lost in **ABORT#** iff a new pair of matching exception logs is created, which happens if either both purses log, or one logs and the other has already logged.

Putting everything together we get the following formula of Dynamic Logic [HKT00]:

$$\langle \text{forall authentic}(\text{na}) \text{ do } \text{ABORT}\#(\text{na}; \text{cstate}); \\ \text{forall } \text{pd} \in \text{maybelost} \text{ do} \\ \text{TRANSFERFAIL}\#(\text{pd.from}, \text{pd.to}, \text{pd.value}; \text{abalance}, \text{lost}) \rangle \\ (\text{abalance} = \text{balance} \\ \wedge \text{lost} = \lambda \text{na}. \Sigma(\text{fromLogged}(\text{na}) \cap \text{toLogged}))$$

Since  $\langle \alpha \rangle \varphi$  means that there is a terminating execution of  $\alpha$  after which  $\varphi$  holds, this says informally: After executing the necessary **ABORT#** steps to get to an idle state, and the necessary **TRANSFERFAIL#** steps to get to a corresponding abstract state we have the simple correspondence of balances and *lost* vs. *exLog*'s as stated in Section 2.

This is already all that is required to define the simulation relation. To efficiently prove it, we simplify it: the only relevant state modification of **ABORT#** is that of the exception log, and **TRANSFERFAIL#** only modifies *abalance* and *lost* of the *from* purse. We also apply the idea to *localize* the simulation relation to one individual purse with name *na*. We get the following definition of the simulation relation **ACINV** and its localized version **LACINV**:

$$\text{ACINV}(\text{astate}, \text{cstate}) \\ := \leftrightarrow \forall \text{authentic}(\text{na}). \text{LACINV}(\text{na}, \text{astate}, \text{cstate}) \\ \text{LACINV}(\text{na}, \text{astate}, \text{cstate}) \\ := \leftrightarrow \langle \text{exLog}'(\text{na}) := \text{exLog}(\text{na}) \cup \text{if } \text{state}(\text{na}) = \text{epv} \vee \text{state}(\text{na}) = \text{epa} \\ \text{then } \{\text{pdAuth}(\text{na})\} \text{ else } \emptyset \rangle \\ (\text{abalance}(\text{na}) = \text{balance}(\text{na}) + \text{intransit}(\text{na}, \text{exLog}'(\text{na}), \text{state}, \text{pdAuth}) \\ \wedge \text{lost}(\text{na}) = \text{lostafterabort}(\text{na}, \text{exLog}'(\text{na}), \text{exLog}))$$

where `intransit` (money that is in transit) and `lostafterabort` (money that will be added to lost when all purses abort) are defined recursively over the set of payment details:

$$\begin{aligned}
 & \text{intransit}(\text{na}, \emptyset, \text{state}, \text{pdAuth}) = \text{lostafterabort}(\text{na}, \emptyset, \text{exLog}) := 0 \\
 & \text{intransit}(\text{na}, \text{pds} \dot{\cup} \{\text{pd}\}, \text{state}, \text{pdAuth}) \\
 := & \text{intransit}(\text{na}, \text{pds}, \text{state}, \text{pdAuth}) \\
 & + \text{if } \text{pd.from} = \text{na} \wedge \text{state}(\text{pd.to}) = \text{epv} \wedge \text{pdAuth}(\text{pd.to}) = \text{pd} \\
 & \text{then } \text{pd.value} \text{ else } 0 \\
 & \text{lostafterabort}(\text{na}, \text{pds} \dot{\cup} \{\text{pd}\}, \text{exLog}) \\
 := & \text{lostafterabort}(\text{na}, \text{pds}, \text{exLog}) \\
 & + \text{if } \text{pd.from} = \text{na} \wedge \text{pd} \in \text{exLog}(\text{pd.to}) \text{ then } \text{pd.value} \text{ else } 0
 \end{aligned}$$

This definition is simpler to use than the original (backward) simulation of [SCW00](#), which uses and has to expand `maybelost`: the definitions of `intransit` and `lostafterabort` can be directly used as rewrite rules.

The main proof obligation of ASM refinement is expressed in Dynamic Logic and assumes an additional (yet unknown) invariant  $\text{CINV}(\text{cstate})$  of the concrete level:

$$\begin{aligned}
 & \text{CINV}(\text{cstate}) \wedge \text{ACINV}(\text{astate}, \text{cstate}) \\
 \rightarrow & \langle\langle \text{CSTEP}\#(\text{cstate}) \rangle\rangle \langle \text{ASTEP}\#(\text{astate}) \vee \text{skip} \rangle \text{ACINV}(\text{astate}, \text{cstate}) \quad (2)
 \end{aligned}$$

$\langle\langle \alpha \rangle\rangle \varphi$  means “all applications of ASM rule  $\alpha$  terminate and yield a state for which  $\varphi$  holds”, so the formula reads: given two states `astate` and `cstate` for which the simulation relation  $\text{ACINV}$  and the invariant  $\text{CINV}$  hold, every execution of the concrete step must terminate, and there must be either a terminating abstract step or a stutter step (which step is necessary, may depend on the final state of the concrete step), such that the two states reached satisfy  $\text{ACINV}$  again. Informally the proof obligation characterizes a commuting diagram with one concrete step and zero or one abstract steps.

The proof obligation can be localized by replacing  $\text{ACINV}$  with  $\text{LACINV}$  for some purse `na`, and the generic  $\text{CSTEP}\#$  with  $\text{LCSTEP}\#$  for some purse receiver (where receiver and `na` may be different or the same). For this localized setting we can replace the nondeterministic choice between  $\text{TRANSFEROK}\#$ ,  $\text{TRANSFERFAIL}\#$  (within  $\text{ASTEP}\#$ ) and `skip` with a deterministic statement by the following reasoning: our simulation relation says, that “after executing  $\text{ABORT}\#$ ’s we reach a concrete state of interest related to some abstract state”. Therefore, only when executing  $\text{ABORT}\#$ ’s before and after  $\text{CSTEP}\#$  leads to a *different* state of interest, an abstract transition must be executed. There are three cases where this is true:

- If receiver is a to purse that successfully receives a value in  $\text{VAL}\#$ , an  $\text{ABORT}\#$  before the operation will cause the money to be lost, but has no effect after the operation. Therefore a  $\text{TRANSFEROK}\#$  is necessary.
- If receiver is a to purse that aborts in `epv`, when the `from` purse already has sent money (i.e. is in state `epa` or has logged the payment details) then

the money will be lost by this action (the `ABORT#` of the to purse before the step has the effect to lose money, executing another `ABORT#` after the `ABORT#` has no effect).

- If receiver is a from purse that accepts a request in `REQ#`, sending a value to a to purse which has already aborted, then the `ABORT#` before the step will not create an exception log, while executing `ABORT#` afterwards will create the second exception log needed for money to be lost. Therefore this case requires a `TRANSFERFAIL#` to be executed too.

Altogether, we get a proof obligation (the full formula is in [SGH<sup>+</sup>06]) that has no quantified formulas (and no nondeterministic programs, which would also lead to instantiating quantifiers), so the proof is fully automatic by symbolic execution of the involved programs. Compared to the 197 interactions needed for the original backward simulation proof in [SGHR06a] this is a vast improvement (especially when considering that the proof of the main lemma has to be iterated on corrections). The proof was developed in about a week, once the right approach had been found (see the remarks at the end of Section 5).

The proof ends in premises of the form  $CINV \rightarrow \varphi$  which give requirements for the invariant of the concrete level that we will develop in Section 5. We close these premises automatically by defining around 40 rewrite rules. Some of them are inter-derivable leaving a basic set of 26 rewrite rules. Our development of the invariant of `CINV` is *lazy* in the sense that it is done after the main simulation proof, guided by the requirement that the rewrite rules must be provable from the definition.

## 4 A Denial of Service Attack

In the next section we will derive an invariant systematically that satisfies the requirements from the end of the last section. Like for the simulation relation, we tried to localize the invariant to individual purses. Several attempts to do this did not work, basically because we did not understand that the protocol allows a particular kind of denial of service attack.

The attack does not violate the original security requirements “no money lost” and “all money accounted” of [SCW00], our proofs show that they are correct and preserved by the refinement. But the attack shows that the protocol violates the property that an attacker should not be able to *systematically* create exception logs. This section describes the relevant scenario and the global property it makes necessary for the invariant.

The scenario is as follows: we assume an attacker that has a faked card and knows the freely available (via `getName+SeqNo`) name and current sequence number of some purse called `from` that is used in `startTo`.

Now, without the `from` purse involved in any way, on the next connection to a to purse the attacker can pose as `from` purse: he will answer the `getName+SeqNo` request of the terminal with `from` and a *future* sequence number `n` of the `from` purse. The terminal will then start a protocol with a `startTo(from, n, value)` message to the

to purse, which will send a request message back. Since to does not receive a value message as response, it has no choice but to abort the protocol.

Repeating the attack several times, the to purse will create an exception log each time. This will fill up the limited amount of space reserved for exception logs quickly: in reality, only a very small number of exception logs is allowed. Since exception logs can also be created by accidents (power failures or an impatient card holder pulling his card too early out of the card reader) the original Mondex specification in [SCW00] has an additional protocol, where the customer shows his card at the bank and gets his exception logs moved from the card to a central archive of the bank<sup>4</sup>. Therefore the fact, that an attacker can systematically (and not accidentally) create exception logs on the to purse can be seen as a mere inconvenience, since the to purse does not lose money.

But the scenario can be taken one step further: if the attacker connects to several to purses as described above, each time posing as the from purse he can collect the request messages he receives (although he can not decrypt them!).

In a second stage he then connects to the from purse, this time posing as one of the various to purses: he can send `startFroms` and all the collected request messages from the to purses to the from purse (provided he has used suitable sequence numbers in the first stage). This will cause the from purse to lose an arbitrary amount of money *immediately* and to write exception logs. Although money is then recoverable at the bank<sup>5</sup>, and all security properties are kept intact by the attack (the attacker just damages the from purse, he does not benefit) we think this behavior is undesirable. The owner of the from purse must go to the bank and force *every* to purse to do the same: the bank will only have evidence to give the money back when it detects matching pairs of exception logs. The motivation for the owners of to purses will be low to do that, since they have not lost any money. They will not notice that the exception log their purse carries does some damage to a from purse, they did not even communicate with.

In our verification the scenario showed up as as the following property: any purse `na` in `idle` state must expect `req(pd)` messages with `na = pd.from` and future sequence numbers `pd.fromno` in the ether. All those messages may be used in future protocol runs, that may fail since `pd` has already been logged by the to purse `pd.to`. Note that future *request* messages are the only ones that may be relevant for a purse in `idle` state (i.e. in a state of interest): all other encrypted messages contain past sequence numbers. The property of future sequence numbers in requests is recorded in the invariant `CINV` of the protocol that we will develop in the next section.

A proposal to remedy the situation is to send an encrypted `startTo` message only as a response to `startFrom`. This would not allow an attacker to create exception logs without having both purses available at the same time (or by

---

<sup>4</sup> Archiving of exception logs at a bank is considered in our contribution to [JW07]. Proofs are available in the Web presentation [KIV]. Since the protocol is small and independent of the main protocol for money transfer it is not considered in the formalization here.

<sup>5</sup> Michael Butlers case study in [JW07] explicitly considers such a recovery step.

pulling out one card in front of his owner). Another solution would be to force purses to respond to a challenge from the terminal to prove, that they are indeed the authentic purses with the correct name. The second solution depends on the authenticity of terminals and requires to include them explicitly in the formal model. Therefore we prefer the first solution, which requires three small modifications for the ASM:

- The last line of the `STARFROM#` rule (see Section 2) now sets `outmsg` to `startTo(pdAuth(receiver))` instead of  $\perp$ .
- This `startTo` message is assumed to be encrypted, so the initial ether no longer contains `startTo` messages.
- Since `STARTTO#` now receives a `startTo` message (in `msg`) with full payment details, it simply sets `pdAuth(receiver) := msg.pd`.

The modifications required to replay those parts of the proofs of the previous section, where `STARFROM#` or `STARTTO#` are considered. Replay worked without any problem and required about an hour of work.

## 5 Systematic Development of an Invariant

Like for the simulation relation, the basic idea for the development is again to focus on states of interest, i.e. idle states, and to use local invariants for purses. Our local invariant `LCINV` will use the last past idle state to say

“the current state `cstate` of the purse named `na` is the result of executing some steps of the protocol starting from an idle state `oldcstate`”

Which steps have been executed can be determined from `state(na)`: for idle state no steps have been done, so `oldcstate = cstate`. If the state is `epr` then the purse has successfully executed (i.e. `okstartFrom`  $\wedge$   $\neg$  `fail?` holds) a `STARFROM#`. Similar clauses result for `state(na) = epv, epa`. Formally `LCINV` is:

```

LCINV(na, oldcstate, cstate)
: $\leftrightarrow$  case state(na) of
  idle : oldcstate = cstate
  epr :   isOKstartFrom(msg)  $\wedge$   $\neg$  fail?
          $\wedge$   $\langle$ STARFROM#(msg, na; oldcstate)# $\rangle$  (oldcstate = cstate)
  epv :   isOKstartTo(msg)  $\wedge$   $\neg$  fail?  $\wedge$ 
          $\wedge$   $\langle$ STARTTO#(msg, na; oldcstate)# $\rangle$  (oldcstate = cstate)
  epa :   isOKstartFrom  $\wedge$   $\neg$  fail?
          $\wedge$   $\langle$ STARFROM#(msg, na; oldcstate);
         REQ#(msg, na; oldcstate) $\rangle$  (oldcstate = cstate)

```

Using this local approach has the advantage, that `LCINV` is trivially invariant for all steps into the protocol (`STARFROM#`, `STARTTO#` and `REQ#`). In these steps `oldcstate` stays the same before and after the step. For the steps finishing a protocol run `oldcstate` after the step is chosen to be the final state of the

step. Therefore, for these steps we will prove properties of *full protocol runs of one purse*: e.g. executing an `ACK#` in state `epa` yields a proof obligation that considers a full execution `STARTFROM#; REQ#; ACK#` of a `from` purse. In essence we will have to verify a “big” diagram consisting of one abstract step and 3 concrete steps.

Compared to the original invariant effort can be concentrated to get the invariant right for full protocol runs. There is no need to explicitly define properties of intermediate protocol states.

The approach works for the *local* state of purses. It is not sufficient for the (global) ether. Two things are necessary for the ether. First, the global invariant is necessary, that we already derived in the last section. Second, for each intermediate state of a purse we need to characterize, whether its communication partner already has sent a response. This can be done abstracting from Mondex protocol to any protocol that sends messages forth and back. Finally, we have to characterize states of interest. All we need for them is that `state(na) = idle` (of course) and that exception logs have sequence numbers in the past. Again, due to the scenario of the previous section, exception logs in `exLog(na)` with `pd.from = na` may have future `to`. Formal definitions of the resulting properties and the resulting full definition of the invariant `CINV` are given in [SGH<sup>+</sup>06].

To verify that `CINV` is a global invariant for `CSTEP#` we again reduce this property to lemmas about the local invariant. In this case we need two lemmas: one for the case where the purse `na` of the local invariant is the same as the receiver `receiver` of the message, and one where it is different.

The proofs are nearly fully automatic: 5 interactions are required. Compared to the original proof, which had 71 interactions, this is again a significant improvement. Working out the proof, once the right approach was found, took around 2 weeks. The main effort was to find the right approach to solve the problem. Several weeks were spent trying to find out, *why* purely local invariants always failed to work, to figure out the worst-case scenario of the previous section, and to find the global property of requests with future sequence numbers.

Summarizing, to work out the full case study using ASM refinement required about 2 person months of work. Changing the ASM to avoid the attack of the previous section makes the proof easier, since the global property of future requests becomes obsolete. Since no other significant change is required, it took only some hours to redo the proofs for the modified protocol.

## 6 Related Work

Prior to our work [B.JPS06] showed that it is possible to define a forward simulation for the Mondex scenario restricted to exactly one `from` and one `to` purse. This work also suggested using generalized forward simulations. Although the complexity of interleaving protocol runs is absent in this scenario, and some of the reasoning of the paper is informal, this work was rather influential for ours. It is interesting to see that, when restricted to the scenario with two purses our forward simulation differs slightly from theirs: While in ours a transition from



`ep` to `epa` implements `TRANSFERFAIL#`, when the `to` purse has already logged, in their refinement it implements `skip` and the failed transfer only happens for the subsequent `ABORT#`. An analysis of the different possibilities to define simulations has been done in [BS07]. This paper also gives a formal account of future and past invariants within an algebraic setting.

Parallel to our work in [SGHR06a], several other groups have successfully formalized and verified the Mondex case study. Their results shall all be published in [JW07]: [R.J06] demonstrates that Alloy and bounded model checking can find all the problems we found, and one more in the proof structure we did not use. The RAISE development in [HGS06] shows an interesting alternative to develop the protocol: it starts with a send and receive instead of a transfer operation (our `ASTEP#`). The RAISE case study develops the communication protocol with two refinements using a variant of forward simulation. This development has a rather well structured invariant, and many of the formulas used in this development are quite close to the ones we use. There are also differences: the formalism used is purely algebraic, and many properties are defined for intermediate states. Nevertheless, proving the Mondex refinement in two steps, one that splits money transfer into send and receive, and another that develops the protocol, could be an improvement for our development too.

The idea of splitting the refinement into smaller steps is taken to its extreme by Michael Butler's group: their development splits the original development into 9 very small refinements, that could be verified in very short time and with very good automation using the B4free tool. Although the final protocol still differs from the Mondex protocol (a `startTrans` action is required, that prohibits some protocol interleavings), the idea of deriving a protocol by introducing a generic transaction concept is remarkable.

A large part of the proof of the Mondex refinement has also been done by David Crocker using the resolution based prover Perfect Developer.

[WF06] use the original Z specification and the original proofs within the Z-Eves tool. By avoiding any translation into another formalism, their approach found some small problems, that no one else could find.

Our idea of using local invariants is a common idea in ASM refinement, it is used e.g. as the core idea in [BM96], which verifies a refinements from sequential to pipelined execution instruction of instructions of the DLX processor using localized invariants for each pipeline stage. The idea is also not specific to ASM refinement, it can be found in other refinement notions, e.g. in work that relates promotion in Z specifications and data refinement.

Our use of states of interest on the other hand seems rather particular to ASM refinement. It was used informally in [BR95] for the compilation of Prolog to WAM, in our formal proofs to verify them [SA98], and was a key notion in the formalization of ASM refinement [Sch01]. The term *states of interest* itself was coined in [Bör03]. The only related refinement notion outside of ASM refinement we are aware of is coupled refinement [DW03], which uses past states of interest (as shown in [Sch05]).

Recently, states of interest were also used in [HGRS07] to analyze security protocols. The idea is to focus on states after all possible attacks have been tried.

## 7 Conclusion

In this paper we have shown how techniques of ASM refinement, namely focusing on *states of interest* and defining *local* invariants, can be used to systematically define a simulation and an invariant for the Mondex refinement.

Our technique has resulted in a simple forward simulation for the Mondex case study that can be verified with a very high degree of automation.

The systematic definition of an invariant has led us to discover a weakness of the protocol with respect to a denial of service attack. Verifying a modified version of the protocol that avoids the attack could be done in a few hours, and it should be debated, whether the weakness is serious enough to change the protocol as suggested.

Although this largely remains future work, we hope that the techniques we used are also applicable for a wider range of security protocols. A first result in this direction is that they can be used in security proofs on an abstract cryptography level [HGRS06].

Our work is part of the more ambitious goal to develop verified JavaCard code for Mondex: a refinement of the communication protocol to a protocol using abstract cryptography has been verified, and is described in [HSGR07]. We are currently working on a refinement to Java Code.

**Acknowledgement.** Gerhard Schellhorn would like to thank Jim Woodcock for his kind invitations to the Mondex workshops.

## References

- [BJPS06] Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the purse: The balance enquiry quandary, and generalised and (1,1) forward refinements. *Fundamenta Informaticae* 77 (2006)
- [BM96] Börger, E., Mazzanti, S.: A Practical Method for Rigorously Controllable Hardware Design. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 151–187. Springer, Heidelberg (1997)
- [Bör03] Börger, E.: The ASM Refinement Method. *Formal Aspects of Computing* 15(1–2), 237–257 (2003)
- [BR95] Börger, E., Rosenzweig, D.: The WAM—definition and compiler correctness. In: Beierle, C., Plümer, L. (eds.) *Logic Programming: Formal Methods and Practical Applications. Studies in Computer Science and Artificial Intelligence*, vol. 11, pp. 20–90. North-Holland, Amsterdam (1995)
- [BS03] Börger, E., Stärk, R.F.: *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003)
- [BS07] Banach, R., Schellhorn, G.: On the refinement of atomic actions. In: *Proceedings of the REFINE Workshop at IFM 2007, Oxford (2007)* (to appear in ENTCS)

- [CSW02] Cooper, D., Stepney, S., Woodcock, J.: Derivation of Z Refinement Proof Rules: forwards and backwards rules incorporating input/output refinement. Technical Report YCS-2002-347, University of York (2002), <http://www-users.cs.york.ac.uk/~susan/bib/ss/z/zrules.htm>
- [DW03] Derrick, J., Wehrheim, H.: Using Coupled Simulations in Non-atomic Refinement. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 127–147. Springer, Heidelberg (2003)
- [Gur95] Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
- [HGRS06] Haneberg, D., Grandy, H., Reif, W., Schellhorn, G.: Verifying Smart Card Applications: An ASM Approach. Technical Report 2006-08, Universität Augsburg (2006)
- [HGRS07] Haneberg, D., Grandy, H., Reif, W., Schellhorn, G.: Verifying Smart Card Applications: An ASM Approach. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 313–332. Springer, Heidelberg (2007)
- [HGS06] Haxthausen, A.E., George, C., Schütz, M.: Specification and Proof of the Mondex Electronic Purse. In: Reed, M., Xin, C., Liu, Z. (eds.) Proceedings of 1st Asian Working Conference on Verified Software, AWCVS 2006, UNU-IIST Reports 348, Macau (November 2006)
- [HKT00] Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
- [HSGR07] Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. Formal Aspects of Computing (2007) (to appear, older version available as Techn. Report 2006-32 at [KIV])
- [JW07] Jones, C., Woodcock, J. (eds.): Formal Aspects of Computing. Springer, Heidelberg (2007) (to appear)
- [KIV] Web presentation of the mondex case study in KIV, <http://www.informatik.uni-augsburg.de/swt/projects/mondex.html>
- [RJ06] Ramananadro, T., Jackson, D.: Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method (2006), <http://www.eleves.ens.fr/home/ramanana/work/mondex/>
- [RSSB98] Reif, W., Schellhorn, G., Stenzel, K., Balsler, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, volume II: Systems and Implementation Techniques: Interactive Theorem Proving, ch. 1, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
- [SA98] Schellhorn, G., Ahrendt, W.: The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, pp. 165–194. Kluwer Academic Publishers, Dordrecht (1998)
- [Sch01] Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. Journal of Universal Computer Science (J.UCS) 7(11), 952–979 (2001), <http://www.jucs.org>
- [Sch05] Schellhorn, G.: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. Journal of Theoretical Computer Science 336(2-3), 403–435 (2005)

- [SCW00] Stepney, S., Cooper, D., Woodcock, J.: AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory (July 2000), <http://www-users.cs.york.ac.uk/~susan/bib/ss/z/monog.htm>
- [SGH<sup>+</sup>06] Schellhorn, G., Grandy, H., Haneberg, D., Moebius, N., Reif, W.: A systematic verification Approach for Mondex Electronic Purses using ASMs. Technical Report 2006-27, Universität Augsburg (2006), <http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/>
- [SGHR06a] Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 16–31. Springer, Heidelberg (2006)
- [SGHR06b] Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. Technical Report 2006-2, Universität Augsburg (2006)
- [WF06] Woodcock, J., Freitas, L.: Z/aves and the mondex electronic purse. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 14–34. Springer, Heidelberg (2006)
- [Woo06] Woodcock, J.: First steps in the verified software grand challenge. IEEE Computer 39(10), 57–64 (2006)

# Management of UML Clusters

Peggy Schmidt and Bernhard Thalheim

Christian-Albrechts-University Kiel, Computer Science Institute, 24098 Kiel,  
Germany

`pescthalheim@is.informatik.uni-kiel.de`

**Abstract.** Software engineering uses UML diagrams as a standard technique for specification and development of software. Various UML diagrams are used for specification of different aspects of the application. Their interpretation, extension, revision and integration becomes awful difficult if developers use the full freedom of UML, apply their own semantics and do not agree on common parts. We propose an approach that limits this freedom to the necessary extent. Developers have the full freedom on parts of the specification that is independent from others and are committed to fulfill contracts on parts of the specification that is also used by other developers.

Due to a lack of semantics the integration of various UML diagrams is often left to the intuition of software engineers, which bears the risk of UML-based software development becoming error-prone. In this paper we propose the use of Abstract State Machines (ASMs) as a means to support the integration of UML diagrams by means of invertible translations of *UML clusters*, i.e. sets of UML diagrams together with constraints defined on them, into easily understandable ASM specifications. In doing so, the rigorous semantics of ASMs induces an unambiguous semantics for the UML clusters. These translations themselves can be formalised by ASM specifications thereby automating the translation process. Furthermore, the evolution of UML clusters is guarded by *contracts*, which can again be specified by ASMs.

## 1 Introduction

Research has been conducted in software engineering for more than forty years. Ideas which have been studied include support for different levels of abstraction, information hiding, and reasoning with local computations. The goal of software engineering is to create high-quality software. Enterprises are becoming increasingly complex in the information age. To realize the building of complex information systems it is essential to resolve such problems as high level specification and target planning at the concept level. Software engineering has produced an enormous amount of notation and methodology that aims to handle the software process.

### 1.1 Software Engineering and Software Specification

Software engineering is thus the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation

required to develop, operate, and maintain them [Boe76]. UML diagrams are ubiquitous in software engineering, forming the cornerstones of modelling techniques. UML intentionally leaves open semantics and formalisations of UML diagrams. This variety of understanding makes the management of UML a mental challenge for large projects and in the context of team development. The pUML (precise UML) initiative [pUM07] tries to add some mathematical rigor to UML but is not yet supported by tools.

*Problem 1. Adding mathematical rigor to representations:* Each work product in software engineering should have an operational semantics. This semantics should allow the development of a (small) logical theory on the basis of which properties of specifications can be proven, validated or verified.

Software products are typically developed in large team whose members are locally developing products. The completion of the development task and the quality [ISO01] is treated in very different ways. The final goal of the software development process is however an integrated and well-behaved product.

*Problem 2. Integrated development of different representations:* UML diagrams are used to specify different views of the same software. They must be used consistently in an integrated form. Their integration must be made explicit.

Software engineering has focused in the past mainly on development processes such as requirements engineering, conceptual system specification, implementation, maintenance, testing, introduction, and deployment. Whereas the latter processes are interwoven the first three processes are more or less sequential. Each development step changes one or two of the development products and leaves the others out.

*Problem 3. Flexible change management:* Any change to one of the development products must take into consideration the changes required by the current changes in other software products.

Software engineering products are documents such as UML diagrams. They are revised and changed during the development process. We need a mechanism that supports the evolution of software specification products and reasoning their properties quality. UML provides a way of communicating between developer and user, and is well accepted in research as well as in industry. UML collects a federation of different models with different views and scope for the product. UML issues a rich set of pictorial and graphical notations. Currently UML is comprised of miscellaneous notations with no formal meaning. The problems of UML include a large number of diagram types and a consequent underspecification of their semantics. There are, however, main differences in the semantics of these diagrams and in the way these diagrams are used.

*Problem 4. Evolution of different representations:* Changes within any UML document, must either be refinements of previous diagrams or explicit revisions of such diagrams. These changes must be enforced for other diagrams as well whenever those are concerned too.

We claim that these four problems can be solved on the basis of abstract state machines (ASM). ASM [BS03] were introduced as a general mathematical framework for systems specification and implementation. This framework supports rough, sketchy specification as well as detailed, fine-grained specification. Both types of specifications can be seen as a machine. The behaviour of these machines can be simulated.

The different specifications vary in their level of detail. We want, however, that one specification can be considered to be a refinement [Bör03] or a revision of the other one. A refinement is generally defined (1) by a structural scope of interest for both machines, (2) by an equivalence relation on states of both structural scopes of interest, (3) by a behavioural scope of interest called a computational segment, and (4) by a partial equivalence relation between the behavioural scope of interest. We require that these equivalence relations allow any behaviour of one machine to be seen or understood as a behaviour of the other machine.

## 1.2 Requirements for Management of UML Specifications

UML diagrams are typically developed either by hand using drawing tools or with support of UML tools such as Rational Rose. The interpretation of the diagrams is often fuzzy and may be hidden within the assumptions of the tools. The UML standard has left this interpretation to the developers and implementers. Therefore we must provide a very flexible framework that allows interpretations to be injected at a later stage. These diagrams are intertwined and depend on each other based on agreement. We thus must cope with the *coexistence* of these diagrams. They allow a specific view on the system that is under development.

The development process has typically four dimensions. The *integration dimension* of a set of diagrams is concerned with the incorporation of several diagrams depending on the overlap and viewpoints supported by the diagram. The *abstraction layer dimension* considers relations among different abstract views on the product, e.g., the realisation of the analysis diagrams by the system specification. The *evolution dimension* adds complexity to the development process by modifying existing diagrams to new diagrams or initiating the development of new diagrams. For example, first a use case diagram is specified and second the state-chart diagram under consideration of the use case diagram. The *collaboration dimension* must take care on the steps that developers may apply while working in a development team. It is not surprising that only the first dimension is partially supported by tools. UML tools are harmed by the lack of a general formal framework of each of the UML diagrams and by the resulting lacking support for integration of diagrams. The integration and the abstraction layer dimensions have not got a satisfying treatment inside UML. The survey [EB04] covers around 4 dozen papers. Constraint examples considered are typically rather trivial. The main kind of constraints are constraints on well-formedness and on existence. Moreover, very few research has been carried out on the evolution dimension and the collaboration dimension.

At the same time a large body of knowledge is available for formal foundation of each of the UML diagrams. We prefer the ASM research that has

resulted in a formal basis of the most important UML diagrams such as class diagrams [Obe01], statecharts [BCR00b], use case diagrams [BGS<sup>+</sup>03], and activity diagrams [BCR00a]. We thus may assume that each of the diagrams can be embedded into their ASM. The consistency of these ASM interpretations has not yet got a satisfying solution.

### 1.3 Organisation of the Paper

Section 2 introduces UML diagram clusters and contracts that allow to manage UML diagram clusters. The explicit formulation of clusters and of contracts is novel and has not been considered in the past. Section 3 describes the support of UML clusters by an *ASM* specifications. We give examples for the transformation UML-Diagrams into *ASM* specifications and for the check of consistency between UML clusters in Section 3 and 4. Section 4 demonstrates the potential of our method by applying it to collaborative development of software within a team of developers. Section 5 summarizes the paper.

## 2 UML Diagram Clusters

The software-development process yields a partially ordered set of UML models that should be consistent. Their semantics should be precisely defined. At the same time it must however defined in a very flexible way and is going to be refined through UML diagram set transformation. The problem thus arises of how to understand and how to check consistency among diagrams and between different versions.

### 2.1 Algebraic UML Systems for Kinds of UML Diagrams

Signatures  $\mathbb{S}$  define the kind of UML diagrams. Typically, signatures describe the base canon that is used within the abstract syntax. Most signatures are given by the specification of the sorts or domains, of the (dynamic) predicates and functions and by formal grammars for inductive construction of complex elements. These grammatical rules may lead to syntactically different but semantically equivalent expressions. Therefore, we use an equivalence relation for expressing the same meaning. Typically, equivalences can be defined on the basis of an equality system.

A UML diagram may be defined as a labelled graph  $\mathcal{G}_{\mathbb{S}}$  of certain signature  $\mathbb{S}$  with a set of integrity constraints or well-formedness rules. These rules are expressed either in the OCL well-formedness rules which extend the syntactic rules given in the meta-language or in first-order predicate logic. The nodes correspond to main elements of signature  $\mathbb{S}$  and edges correspond to their relations. Labels are used for providing additional information on the nodes and edges.

For instance, a statechart diagram is given by a set of states and their transitions. A initial/ final state has at least one outgoing / incoming transition and no incoming / outgoing transition. Attribute and operation identifiers in class



diagrams may required to be unique. We may also use general constraints such as the *unique name assumption* that requires names to be unique within a diagram.

Such conditions can be considered to be constraints that specify well-formedness  $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$  of diagrams. This set also includes any equivalence of interest for the given kind of diagrams.

In general, we may weaken requirements to well-formedness for the intermediate steps in a development process. For instance, the assignment of *public*, *private* and the specification of operations can be delayed. Additionally, UML intentionally leaves open semantics of diagrams. For instance, a number the options for semantical interpretation of statecharts is discussed in [BCR00b]. At least four different decisions for semantics of UML statecharts must be made before a statechart diagram interpretation can be generated: (a) event handling (whether an event is considered to be a releasable and dispatched event); (b) concurrency treatment for transitions in compound states; (c) environment for generation of state completion; (d) selection and handling of transitions.

A similar variety of interpretations can be observed for class diagrams, activity diagrams, and interaction diagrams. The clarification of class diagram semantics may either be based on semantics of object-oriented databases [ST93, Obe01] or on proposals made by the pUML (precise UML) initiative [pUM07]. The interpretation of use diagrams we use is based on the SiteLang language [DT01] that has been developed for specification of user behaviour within information-intensive web systems.

Since interpretations do not have a Church-Rosser property and one choice for interpretation of one construct may restrict the application of an interpretation of another construct we restrict our consideration to the **semantics integration assumption**: *Given a kind of UML diagrams defined over a signature  $\mathbb{S}$ . Restrictions and equivalences applied to this kind of diagrams are not contradicting.* If this assumption is applicable then we may assume that all diagrams of the given kind may have a strongest and a weakest interpretation (final and free interpretation) defined by the conjunction or the disjunction of all restrictions, respectively. The interpretations form a lattice of hierarchically ordered equationally partial heterogeneous structures or algebras [Mal70, Rei84].

This approach may be combined with the theory of institutions and the CASL approaches [BST02] to algebraic UML systems. Institutions provide a set of signatures, a set of formulas defined on these signatures and a set of model classes for the signatures. Model classes are related by a satisfaction relation that define whether a model  $M$  of some signature  $\mathcal{S}$  satisfies a formula  $\alpha$  from  $\mathcal{L}_{\mathcal{S}}$ .

We assume now that the signature is fixed for one kind of UML diagrams. The different interpretations possible form a lattice  $\mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$  of subsets of  $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$  with the set  $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$  as the largest element and a subset of  $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$  as the smallest element that is considered the lowest standard for well-formedness of diagrams. Additionally, modification operations must be specified for UML diagrams. These operations may be considered as specialisations of typical graph and schema editing operations known for database schemata

[Tha00]. Some of these operations can contingent others, i.e. can be used to enhance the other operations in such a way that constraints are becoming fulfilled if these operations are applied. Some of these operations can also be compensating operation that allow to undo other operations.

An algebraic UML system  $(\{\mathcal{G}_{\mathbb{S}}\}, \mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}}), \mathcal{O}_{\mathbb{S}})$  of signature  $\mathbb{S}$  is given by all labelled graphs  $\{\mathcal{G}_{\mathbb{S}}\}$  on signature  $\mathbb{S}$ , a lattice  $\mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$  of sets of constraints, and by modification and development operations  $\mathcal{O}_{\mathbb{S}}$  that can be used for the development of diagrams of signature  $\mathbb{S}$ .

## 2.2 Contracted Development of UML Diagrams

Consistency of sets can be specified through a set of logical formulas that are specified in an appropriate language. For instance, database applications use integrity constraints which specify which states of the database are desirable and which states are forbidden. UML diagrams can be considered as a set of abstract objects. The meaning of these objects is the illustration of different viewpoints and properties of a program. The UML meta-language provides a framework for description of objects that are represented by a UML diagram. These objects typically have complex structures and are well-formed according to the requirements of the UML standard [HKKR05]. A set of UML diagrams describe an application. Therefore, UML diagrams must also obey consistency constraints. Typically, these consistency constraints are not explicitly specified. The explicit specification and the sophisticated treatment of these constraints contributes to the solution of the four problems discussed in Section 1.1. We base our approach on a four-layer treatment in contracted development:

1. Declaration of constraints that are applied to a singleton diagram or to sets of coexisting diagrams;
2. Description of enforcement mechanisms (when must the constraint checked, how the constraint is checked, what to do if the constraint is violated, what mechanism can be used to trigger the constraint) that support constraint validity during development, change, and evolution of UML diagrams;
3. Description of change and evolution steps that can be applied for refinement or modification of sets of UML diagrams based on scopes of constraints and operational use of constraints;
4. Support by tools or workbenches that maintain validity of constraints.

The third layer may also consider the development of UML diagrams within development teams. In this case, team members are supported by approaches to collaboration, e.g. explicit services and exchange frames [ST07].

A contract  $\zeta^{\text{Contract}}$  consists of a declaration of constraints, of a description of the enforcement mechanism and of a prescription of modification steps that transform a consistent set of diagrams into a consistent set of diagrams.

A contract may include obligations, permissions and sanctions. Therefore,

- contracts declare correctness of a set of diagrams, separate exceptional states from normal states for these sets, and forbid meaningless sets of diagrams,

- contracts enable the direct manipulation of the set of diagrams as transparently as possible and offer the required feedback in the case of invalidation of constraints based on echo back, visualisation of implications, on deferred validation, instant projection and hypothetical compilation, and
- contracts consider mechanisms that address the long term integrity of diagram sets by forecasting confirmation, by anticipating changes made in a team, by providing a mechanism for adjusting and confirming correctness, and by specifying diagnostic queries for inspection of diagram sets.

A typical constraint on a set of diagrams is the existence constraint

$$EC_1^{states(SC,CT)} : \text{StateChart}(\text{States}) \subseteq \text{ClassDiagram}(\pi_X(\text{RulingClass}))$$

where *RulingClass* is a class defined in the class diagram and  $\pi_X(Y)$  is the projection function. The constraint requires that states in statecharts must be defined by attributes of the ruling class in a class diagram. Changes to states are thus restricted. We declare

$$\begin{aligned} \text{State}' \notin \text{ClassDiagram}(\pi_X(\text{RulingClass})) &\longrightarrow \\ &F \text{ modify}(\text{StateChart}(\text{State}, \text{State}')), \end{aligned}$$

$$O \text{ cascade}(\text{modify}(\text{ClassDiagram}(\text{RulingClass}, X)), \text{modify}(\text{StateChart}(\text{State})))$$

where `modify` denotes any diagram modification operation applicable to the signature of the diagram, `cascade` denotes an obligation to apply a second action if the first action has been completed and *F*, *P*, *O* are the deontic operators forbid, permit, oblige. Furthermore, we assume

$$\begin{aligned} \text{do}(A_1, \text{modify}(\text{ClassDiagram}(\text{RulingClass}, X))) &\longrightarrow \\ &\text{do}(\text{notify}(A_2, \text{modify}(\text{ClassDiagram}(\text{RulingClass}, X)))) \end{aligned}$$

for any two agents  $A_1$  and  $A_2$  where  $A_1$  the right to modify class diagrams and  $A_2$  has the right to read the statechart.

For instance, in the running example we require that the states *IsBorrowed*, *IsReturned* are definable for the class *Book*, i.e.

$$\text{StateChart}(\{IsBorrowed, IsReturned\}) \subseteq \text{ClassDiagram}(\pi_{LendingState}(\text{Book})) .$$

Contracts typically follow a number of norms that are given by laws, regulations or agreements among the parties involved. Our treatment generalise this understanding. Parties involved into a contract are either singleton diagrams or team members involved in a development project. A contract may be extended by the following information: roles of the parties that are involved; relationships between contracting parties; begin and end of contract; the status of contracts; a contract monitoring facility that performs checking of the fulfillment of obligations and compliance monitoring; a contract notification component that sends various contract notifications to the roles involved in contract management; other components and facilities to support contract negotiations, enforcement and also dynamic configurations of the system to reflect new business rules and structures.

### 2.3 Coexistence of UML Diagrams and UML Clusters

During UML-based system specification a number of UML diagrams is used for description of different viewpoints in different levels of detail on the application. UML does not provide mechanisms for support of coexistence of diagrams beyond

the UML meta-model, the Object Constraints Language OCL [HKKR05] and rules developed within the precise UML initiative. Coexistence of diagrams must be handled within all four dimensions: integration, abstraction layer, evolution and collaboration.

The development of UML diagrams is performed by agent (or actors) that obtain rights to apply (do) modification and development operations and can be obliged to apply these operations during their work. Rights are ordered. For instance, an agent that can modify diagrams have typically the right to read these diagrams.

Beside well-formedness we might also consider any kind of constraint over diagrams of signature  $\mathbb{S}$ . For instance, hierarchies in class diagrams must be acyclic. States in statecharts must uniquely determine the node in the statechart. Therefore, we define a **diagram type**  $\mathcal{T}_{\mathbb{S}} = (\mathbb{S}, \Sigma_{\mathbb{S}})$  by the signature of the diagram and by constraints  $\Sigma_{\mathbb{S}} \in \mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$  applicable to all diagrams defined over this type .

Constraints in UML clusters can be categorised into

- existence constraints**  $\mathbb{S}_1[E] \subseteq \text{Exp}(\mathbb{S}_2)$  that bind the utilisation of one element set  $E$  in a diagram of signature  $\mathbb{S}_1$  to the existence of this element or of an expression that declares this element within a diagram of signature  $\mathbb{S}_2$ ,
- visibility constraints**  $E \subseteq \text{Public}(\text{Exp}(\mathbb{S}_2))$  for  $E \subseteq \text{Exp}(\mathbb{S}_1)$  that require that expressions used in one diagrams must be visible on the other diagram,
- cardinality constraints**  $\text{card}_{\mathbb{S}_1} = (m_1, n_1) \longrightarrow \text{card}_{\mathbb{S}_2} = (m_2, n_2)$  that restrict the multiplicity of elements in one diagram is bind by the multiplicity of elements in another diagram,
- refinement constraints**  $e = E$  for elements  $e$  in a diagram of signature  $\mathbb{S}_1$  and expressions  $E \in \text{Exp}(\mathbb{S}_2)$  that restrict the refinement of diagrams of signature  $\mathbb{S}_1$  by diagrams of signature  $\mathbb{S}_2$  and
- evolution constraints** that specify the consistency of different versions of the same application.

For instance, a link between a sender and receiver in a sequence diagram must be based on the existence of a corresponding association in the class diagram. All elements in a sequence diagram require that the corresponding classes, attributes, operations and references are visible in the class diagram. Since messages in sequence diagrams can initiate creation or deletion of object the cardinality of these object sets must be consistent with the cardinality constraints specified for the class diagram. Use cases in a use case diagram can be refined by an entire activity diagram.

A UML cluster type  $\mathcal{CT} = (\mathcal{T}_{\mathbb{S}_1}, \dots, \mathcal{T}_{\mathbb{S}_n}, \Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n})$  is given by UML types  $\mathcal{T}_{\mathbb{S}_i}$  defined on a set  $\mathbb{S}_1, \dots, \mathbb{S}_n$  of signatures and a set  $\Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$  of constraints on these signatures. A UML cluster  $\mathcal{C}$  on a cluster type  $\mathcal{CT}$  consists of UML diagrams  $(\mathcal{D}_1, \dots, \mathcal{D}_n)$  of type  $\mathcal{T}_{\mathbb{S}_i}$  that obey  $\Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$ . The contract on  $\mathcal{CT}$  thus consists of the constraints  $\Sigma_{\mathbb{S}_1} \cup \dots \cup \Sigma_{\mathbb{S}_n} \cup \Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$ , a description of the enforcement mechanisms for any operation that can be used for modification of one UML diagram, and a set of consistent evolution transformations.

The evolution steps may be very complex. We may either use a transaction approach that accept only those modification step sequences which are correct or may develop modification operations which are correct. We prefer the second approach and aim in development of atomic steps. These steps must obey the contract. Therefore we also need compensation and contingent operations that compensate the operation or that continue the operation in the case that the step has led to a cluster which is not consistent. The final result of evolution steps is typically a UML cluster. Currently, all approaches prioritise one diagram signature and consider the corresponding diagram to be the final product supported by the other diagrams.

### 3 ASM-Based UML Clusters

#### 3.1 ASM Basis for UML Clusters

The coexistence of UML diagrams can be either supported on the basis of algorithms that check contracts, e.g., [Tsi00], or map the consistency to certain logics or check consistency on the basis of operational semantics. Most approaches develop algorithms for certain classes of constraints. Any new class of constraints must thus be supported by new algorithms. Some few approaches map constraints to certain logics such as description logics, e.g. [SMSJ03]. Unfortunately [Tha00], description logics are already insufficient for handling of cardinality constraints. We prefer operational semantics. We map UML diagrams to an ASM based on the mappings that have already been given in [BGS<sup>+</sup>03, BCR00b, BCR00a, Obe01]. This mapping achieves an integration of all diagrams. We therefore use an operational consistency paradigm based on the requirement that a UML cluster has a common semantic interpretation.

For instance, the transformation of class diagrams to ASM is based on the theory of object-oriented databases [ST93, ST99] and [Obe01]. We use the layering in Section 2.3. Let us assume that a number of static domains such as *Name*, *AttrName*, *OpName*, *AssocName*, ... are given. The signature of class diagrams is given by

$\mathcal{ASM}_{\text{ClassTypeSystem}}$

$$\begin{aligned} \mathcal{T}^{\text{CT}} &= \{\tau_1, \dots, \tau_{n_T}\}, & \mathcal{A}^{\text{CT}} &= \{A_1, \dots, A_{n_A}\}, & \mathcal{O}^{\text{CT}} &= \{o_1, \dots, o_{n_O}\} & \mathcal{E}^{\text{CT}} &= \{e_1, \dots, e_{n_E}\} \\ \text{name} &: \mathcal{T}^{\text{CT}} \rightarrow \text{Name} \\ \text{attr}^T &: \mathcal{T}^{\text{CT}} \times \mathcal{A}^{\text{CT}} \rightarrow \text{AttrName} \times \text{Visibility} \times \text{AttrDataType} \\ \text{operation}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{O}^{\text{CT}} \rightarrow \text{OpName} \times \text{Visibility} \times \text{ParamList} \times \text{MethDataType} \\ \text{association}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{AssocName} \times \text{AssocKind} \\ \text{association\_attr\_source}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{Role} \times \text{Card} \times \text{Visibility} \\ \text{association\_attr\_source}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{Role} \times \text{Card} \times \text{Visibility} \\ &\dots \end{aligned}$$

The entire translation process results in an  $\mathcal{ASM}_C^{\text{Spec}}$  specification of the cluster. This specification also reflects all constraints in the cluster. For instance, we use constraints expressing well-formedness of diagrams. Let us consider a constraint which requires that names of associations uniquely determine the kind of the association (normal, generalisation, ...)

**constraint**  $\mathbf{WF}_{FD1}^{CT} : \forall \tau_1 \forall \tau_2 \in \mathcal{T}^{CT} \forall \zeta_1 \forall \zeta_2 \in \mathcal{E}^{CT}$   
 $(\text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1) \neq \mathbf{undef} \wedge \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2) \neq \mathbf{undef}$   
 $\text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1) = (an_1, ak_1) \wedge \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2) = (an_2, ak_2)$   
 $\wedge an_1 = an_2) \longrightarrow ak_1 = ak_2$  .

We notice that a number of different transformation styles can be applied. We used for the example above an object-preserving transformation that concentrates the translation to nodes and edges of the graph. Instead we might use full unnesting that generates a function for each component of the graph. Another style uses objects such as attributes, operations, etc. on their own and associates them with the corresponding class through a function. We use the unique name assumption and assign a unique namespace to each diagram.

Constraints can be transformed to ASM rules, e.g. the constraint above to

$\mathbf{WF}_{FD1}^{CT} =$   
**foreach**  $\tau_1, \tau_2 \in \mathcal{T}^{CT}, \zeta_1, \zeta_2 \in \mathcal{E}^{CT}$  **do**  
**let**  $(an_1, ak_1) = \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1)$  **in**  
**let**  $(an_2, ak_2) = \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2)$  **in**  
**if**  $an_1 = an_2 \wedge an_1 \neq \mathbf{undef} \wedge ak_1 \neq \mathbf{undef} \wedge ak_2 \neq \mathbf{undef} \wedge ak_1 \neq ak_2$  **then**  
 $\text{reaction}(\zeta_1, \zeta_2) := \text{contract\_reaction}(\mathbf{WF}_{FD1})$

where we assume that `contract_reaction` is a function set on initialisation.

### 3.2 ASM-Based Contract Management

An  $\mathcal{ASM}_C^{\text{Spec}}$  specification is enhanced by assumptions  $\mathcal{ASM}^{\text{Assum}}$  and guarantees  $\mathcal{ASM}^{\text{Guaran}}$ , i.e.,  $\mathcal{ASM}^{\text{Contract}} = (\mathcal{ASM}^{\text{Assum}}, \mathcal{ASM}^{\text{Guaran}})$ . Guarantees can be seen as the smallest common divisor. They are going to be supported by the environment  $\mathcal{ASM}_{\text{Environment}}^{\text{Spec}}$  if it behaves according to  $\mathcal{ASM}_A$ . Assumptions may also represent requirements of  $\mathcal{ASM}_C^{\text{Spec}}$  to the environment. We assume that the assumptions imply the guarantees, i.e., any consistent with  $\mathcal{ASM}^{\text{Assum}}$  specification obeys the guarantees.

The abstract state machines  $\mathcal{ASM}^{\text{Contract}}$  handles the fulfillment of contracts. The generation process of these machines is demonstrated in Section 4.

The management of contracts is based on three steps [KN02]: registration, contract negotiation and contract execution.

**Registration** Two agents are involved into the registration phase and use the role of acting addressee and the role of reacting counter-party. They identify their need to be engaged in a change of  $\mathcal{ASM}^{\text{Spec}}$  under the supervision of the contract manager ( $\mathcal{ASM}^{\text{Contract}}$ ). Within the next step they may agree in principle on issues or open a negotiation. These agreements will determine the type of service required from the contract manager. The purpose of the contract will be negotiated in the following phase. The type of service is expressed as a **Contract Template** and put forward by the authority to the two contracting agents.

**Contract negotiation:** This step manages the domain-specific content of the contract following the template agreed upon in the registration phase. Issues determined to be important in the registration phase can be negotiated, for

example, for a decomposition step applied to a class and resulting in several classes within a UML class diagram. In general, a contract specifies the collaboration of agents whenever changes applied to  $\mathcal{ASM}^{\text{Spec}}$  specifications (e.g.,  $\mathcal{ASM}^{\text{ClstrT}}$ ,  $\mathcal{ASM}^{\text{Clstr}}$ , or  $\mathcal{ASM}^{\text{Prgrm}}$ ). We distinguish between *obligations* for the acting and the reacting agent, *permissions* given by the reacting agent to the acting agent, and *sanctions* applied to the  $\mathcal{ASM}^{\text{Contract}}$  to the acting agent or to the reacting agent.

**Contract execution:** The fully negotiated contract is executed by the three agents under the supervision of the  $\mathcal{ASM}^{\text{Contract}}$ . The bound contract contains declarations of obligations, permissions and sanctions of each party following the template used. These declarations will lead to the execution of the contract.

Contracts describe constraints that the  $\mathcal{ASM}^{\text{Spec}}$  must satisfy before using the service as well as the constraints that are guaranteed by the service when used. The **Activation** defines preconditions for obligations, permissions or sanctions. **Finalization** assumes that the activation is true. If the activation and finalization conditions are met, then the service permissions must be preserved.

A development contract may contain additional constraints. For example, the following  $\mathcal{ASM}^{\text{Contract}}$  specifies that all methods in a statechart of a cluster have to exist in the class diagram of this cluster, i.e. the existence constraint

$$\text{EC}_1^{\text{meth}(SC,CT)}: \text{StateChart}(\text{Methods}) \subseteq \text{ClassDiagram}(\text{Methods})$$

is transformed to the generalised rule

$$\text{INTEGRITYCONSTRAINTTEST} \Leftrightarrow \mathcal{O}^{\text{SC}} \subseteq \mathcal{O}^{\text{CT}} \wedge \{\delta^{\tau_{\text{class diagram}}}, \delta^{\tau_{\text{statechart}}}\} \subseteq \mathcal{C}$$

and to the rule

$$\begin{aligned} \text{EC}_1^{\text{meth}(SC,CT)} = & \\ & \text{foreach } o \in \mathcal{O}^{\text{SC}} \text{ do} \\ & \text{if } o \notin \mathcal{O}^{\text{CT}} \text{ then} \\ & \quad \text{reaction} := \text{contract\_reaction}(\text{EC}_1^{\text{meth}(SC,CT)}) \end{aligned}$$

### 3.3 Supporting Management by Contract Templates

The contract specifies the coherence within and between UML diagram clusters and supports propagation of modifications throughout the cluster. The contract states which constraints remain to be valid after a modification. Contracts may include specific styles and pattern for the treatment of modifications. For instance, contracts may be based on the ACID paradigm, i.e. any modification set is either completely applied to the cluster or rolled back, can be applied in isolation from other modification sets and is then persistent for the cluster. If contracts can generically be defined then contract templates can be defined. A contract template is a contract with parameters for the UML diagrams. The template is instantiated to the machine  $\mathcal{ASM}^{\text{Contract}}$  that manages contracts.



## 4 Development of Clusters and Contracts

We may easily extend the approach to cope with several clusters. In this case we have to solve two problems at the same time:

- coexistence of diagrams within one cluster based on the cluster constraints and
- collaboration of developers that develop diagrams within different clusters.

The result of a collaboration should be a cluster that is commonly agreed between the agents. Diagrams should be mapped to an  $ASM^{Prgrm}$  specification. Any modification can only be applied to a diagram in the cluster if the contract is preserved by the  $ASM^{Prgrm}$ .

We use an  $ASM^{Contract}$  for management of deployment of the  $ASM^{Clstr}$  and  $ASM^{Prgrm}$  specifications based on three services: generating a  $ASM^{Prgrm}$  specification from a  $ASM^{Clstr}$ , decomposition of classes inside an  $ASM^{Prgrm}$ , and collaboration of two UML diagram clusters.

### 4.1 Example: Library Support System

We illustrate the approach on the basis of a small example for a library support system. We shall use this example to demonstrate the power of our approach.

In Figure 1 two UML diagram clusters  $C_1$  and  $C_2$  are depicted. The first one uses a use case and a class diagram. The second one considers a start chart and a view on the class diagram of the first cluster. The use cases in Figure 1 are *borrow* and *return* a book from the library. The class diagram in the first cluster  $C_1$  uses the classes *Book*, the *Person*, *Log4BorrowingABook*. The *statechart* diagram models the different states. In our example one can borrow a book only if the book is available (*isReturned*).

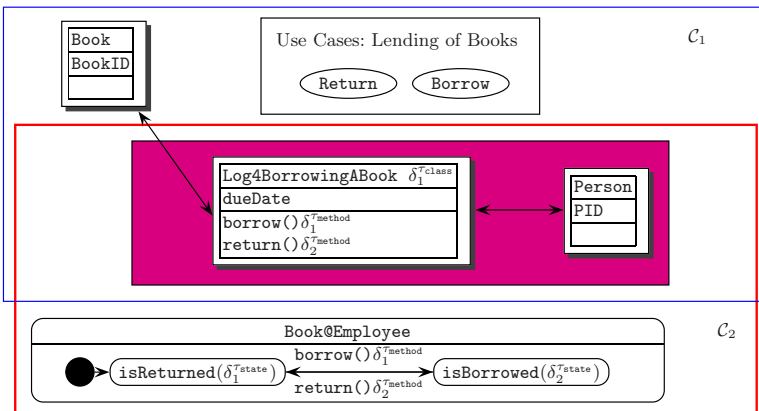


Fig. 1. Two diagram clusters developed for the *Library* application



The UML clusters developed so far can be transformed into an  $ASM_{C_1}$  and  $ASM_{C_2}$ . The transformation depends on the transformation style. For instance, one transformation style requires that each method of a class is transformed into a dynamic function. The translation style we used above transforms methods into domains.

The transformation of UML diagrams to programs or  $ASM$  can be based on translation profiles or styles similar to those developed by diagramming techniques for databases [Cha00]. The transformation result of cluster  $C_1$  may be based on style A:

SIGNATURE OF  $ASM^{Program, styleA}$

```
Log4BorrowingABookID = {1, ..., n}
...
Method = {borrow, return}
LendingState = {IsBorrowed, IsReturned}
book : Log4BorrowingABookID → BookID
person : Log4BorrowingABookID → PID
dueDate : Log4BorrowingABookID → Date
method_m : Log4BorrowingABookID → String
```

MAINRULE

```
for each o ∈ BorrowedBook do
  if method(o) == borrowBook then
    INITIALIZE()
  ...
```

We may also apply another style style B. It results in the following:

SIGNATURE OF  $ASM^{Program, styleB}$

```
...
log4BorrowingABook: Log4BorrowingABookID × BookID × PID
                    × DueDate → Bool
```

MAINRULE

```
for each o ∈ Log4BorrowingABook do
  ...
```

In the current example style A is used for both clusters since the contract requires that the style for transformation must be fixed. We get the ASM in Figure 2.

## 4.2 Collaborative Development and Collaboration Contracts

The collaboration contract  $\zeta_{C_1, C_2}^{Contract, .co11}$  is based on an equality system  $\mathfrak{E}_{C_1, C_2}$  that relates expressions in  $C_1$  to expressions in  $C_2$ . A simple case of such equality systems all expressions are basic elements of the cluster. In this case, we may introduce directed equalities. These equalities propose a preference which of the notions is going to be used for the integrated cluster  $C_{12}$ . At the same time, we can use inequalities for those basic elements that cannot be related to each other. The practicality of equality logics for schema integration has already been shown for database schemata in [Ves05]. It can be enhanced by cooperating views [FRT05, Cha00] which are used for database collaboration.

```

SIGNATURE OF  $ASM_{aM}^{Program, styleA}$ 
student : Log4BorrowBookByStudentID  $\rightarrow int$ 
book : Log4BorrowBookByStudentID  $\rightarrow int$ 
dueDate : Log4BorrowBookByStudentID  $\rightarrow Date$ 

MAINRULE
foreach  $o \in Log4BorrBookStud$  do
  if method( $o$ ) == borrow() then
    INITIALIZE()
  if method( $o$ ) == borrow() then
    // Code is generated from the state-chart
1   if Actor( $o$ ) == Student then
    // precondition
2   if ctl_state( $o$ ) == undef  $\vee$  ctl_state( $o$ ) == isReturned then
3     let  $o = \text{new} (Log4BorrBookStud())$ 
4     book( $o$ ) = param_book
5     Student( $o$ ) = param_Student
6     Duedate( $o$ ) = param_Duedate
    // postcondition
7     ctl_state( $o$ ) := isBorrowed
8     ctl_stateIsBorrowed( $o$ ) := isNotOD
  if method( $o$ ) == return then
    if Actor == Student then
      // precondition
9      if ctl_state( $o$ ) == isBorrowed then
10     return() // postcondition
11     ctl_state( $o$ ) := isReturned
  if Date > DueDate( $o$ ) then
    // precondition
12    if ctl_state( $o$ ) == isNotOD then
13    ctl_stateIsBorrowed( $o$ ) := isOD

```

**Fig. 2.** The ASM specification after generation

The equality system  $\mathfrak{E}_{\mathcal{C}_1, \mathcal{C}_2}$  is now the basis for an extended contract. The contracts of the clusters  $\mathcal{C}_1, \mathcal{C}_2$  can be enhanced by the equality system, the agreements on enforcement for equalities provided by  $\mathfrak{E}_{\mathcal{C}_1, \mathcal{C}_2}$  and the modifications that can be applied to  $\mathcal{C}_1$  or  $\mathcal{C}_2$ . This contract part is called **collaboration contract**  $\zeta_{\mathcal{C}_1, \mathcal{C}_2}^{\text{Contract}}$ . It defines what each agent in the development process of their clusters  $\mathcal{C}_1, \mathcal{C}_2$  should do.

The collaboration contract can also be represented by a contract of the union of the two clusters. UML clusters may contain several UML diagrams of the same kind. Therefore, we can use the same approach for cluster development and for collaborative development of several clusters. Contracts can directly be transformed to corresponding rules of the cluster management  $ASM^{\text{Cluster}}$ .

For instance, the condition  $c \in \zeta_{C_1, C_2}^{\text{Contract} \cdot \text{Coll}}$   
 An agent is permitted to modify a cluster as long as the agent changes only those shared parts in a UML diagram cluster that do not change the input-output-behavior of any associated  $\text{ASM}^{\text{Prgrm}}$ .

is supported by the following rule for activities of agents  $a$  to cluster  $C_i$ :

```

PERMISSIONFORMODIFICATION ( $a, C_{i,old}, C_{i,new}$ )
if  $\text{typeOfEvent}(\text{event}(a)) == \text{Modification} \wedge$ 
     $\text{owner}(C_{i,old}) == a \wedge \text{owner}(C_{i,new}) == a$ 
  then // Activation
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{tryToCommitModification} \wedge$ 
         $\text{inputState}(\text{ASM}(C_{i,old})) == \text{inputState}(\text{ASM}(C_{i,new})) \wedge$ 
         $\text{outputState}(\text{ASM}(C_{i,old})) == \text{outputState}(\text{ASM}(C_{i,new}))$ 
      then  $\text{stateOfEvent}(\text{event}(a)) := \text{permitted}$  // Execution
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{permitted}$  // Finalisation
      then  $\text{CommitModification}(C_{i,old}, C_{i,new})$ 
         $\text{stateOfEvent}(\text{event}(a)) := \text{completed}$ 

```

We use basic functions  $\text{stateOfEvent}(e)$ ,  $\text{typeOfEvent}(e)$  for events,  $\text{event}(a)$  for agents,  $\text{owner}(C)$  for clusters, a number of derived functions such as the functions  $\text{inputState}(\text{ASM})$ ,  $\text{outputState}(\text{ASM})$ , and the transition rules

$\text{CommitModification}(C_{old}, C_{new})$  and  $\text{Negotiate}(a, C_{old}, C_{new})$ .

The transition rule  $\text{Change}(C_{old}, C_{new})$  can only be applied if the permission for this modification is given to an agent.

At the same time if a shared part is affected by the modification then a negotiation process must start.

```

OBLIGATIONSIMPOSEDBYMODIFICATION ( $a, C_{i,old}, C_{i,new}$ )
if  $\text{typeOfEvent}(\text{event}(a)) == \text{Modification} \wedge$ 
     $\text{owner}(C_{i,old}) == a \wedge \text{owner}(C_{i,new}) == a$ 
  then // Activation
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{tryToCommitModification} \wedge$ 
         $\text{inputState}(\text{ASM}(C_{i,old})) == \text{inputState}(\text{ASM}(C_{i,new})) \wedge$ 
         $\text{outputState}(\text{ASM}(C_{i,old})) \neq \text{outputState}(\text{ASM}(C_{i,new}))$ 
      then  $\text{Negotiate}(a, C_{i,old}, C_{i,new})$  // Execution
         $\text{stateOfEvent}(\text{event}(a)) := \text{negotiate}$ 
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{negotiate} \wedge$ 
         $\text{stateOfNegotiation}(a) = \text{completed}$ 
      then  $\text{CommitModification}(C_{i,old}, C_{i,new})$  // Finalisation
         $\text{stateOfEvent}(\text{event}(a)) := \text{completed}$ 

```

In a similar form we develop sanctions for the case of the violation of a contract. The most liberal sanction is the delivery of a copy to the agent that has been inconsistently modifying the cluster. The collaboration with other agents is then interrupted until the cluster copy is again associated with others by a contract.

We may generalise this approach to contract managers as discussed in Section

**3.2** The rules above may be generalised to

```

PERMISSIONFORMODIFICATION ( $a, C_{i,old}, C_{i,new}, \zeta^{\text{Contract}}$ ) and
OBLIGATIONSIMPOSEDBYMODIFICATION ( $a, C_{i,old}, C_{i,new}, \zeta^{\text{Contract}}$ ).

```

### 4.3 Support for Change Management

The constraint  $EC_1^{states(SC,CT)}$  requires that states used in a statechart must be states of the ruling class. Let us now assume that one agent decides to decompose a ruling class. For instance, it becomes known that the lending process for a student is different from that lending process of an employee. Employees can borrow a book for an unlimited period. The student must return the book at least by the the deadline. We thus are going to modify cluster  $C_2$ . People considered are either students or employees.

Since the class *Person* is split into two classes the statechart diagram in  $C_2$  is going to be modified. The new classes inherit the attributes of the old class *Log4BorrowingABook*. This change is performed according to the rules for collaboration discussed in Section 4.2. The result is given in Figure 3. Finally, the statechart diagram is harmonised with the *Log4BorrBookEmp* and *Log4BorrBookStud* classes. The integrated  $ASM_{styleAaM}^{Prgrm}$  is obtained as a result of these operations.

This example shows now that our framework prohibits from developing inconsistent clusters.

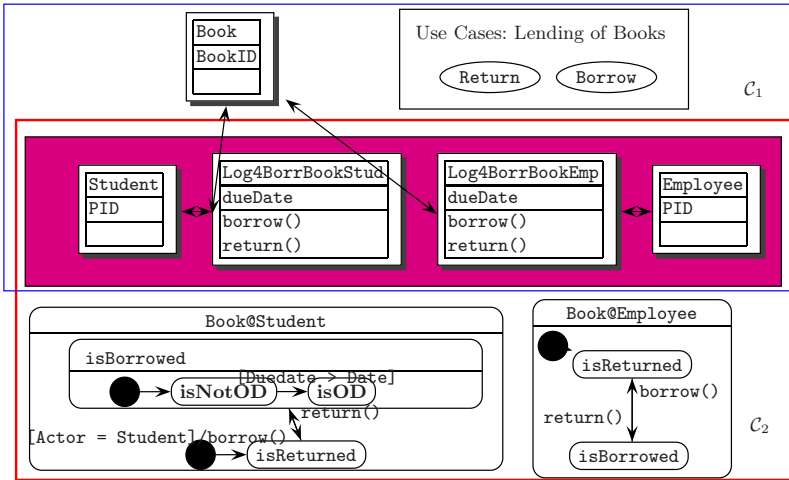


Fig. 3. Diagram clusters after modification (separation of class *BorrowingABook*)

## 5 Conclusion

Software is typically a complex product that has been developed by a team. The development process is often similar to the work of an artisan. Intermediate products are typically rather informal and may be contradictory. The final product is a program that has a well-defined semantics. Large software systems can be simplified tremendously if techniques of modular modelling such as design by

UML diagram clusters are used. Modular modelling is an abstraction technique based on principles of hiding and encapsulation. Design by UML diagram clusters and its corresponding  $\mathcal{ASM}$  allows to consider parts of the software systems in a separate fashion. Software reuse has been considered but never reached the maturity for application engineering.

It is important to provide a consistent and unambiguous semantics for software specification within the team context, so that all team members have the same interpretation of the specification. The fact that UML lacks a precise semantics is a serious disadvantage of UML-based methodologies.

UML might use semantic variation points [CJ05] for support of intentional degrees of freedom for interpretation of the metamodel semantics. Semantic variation points are used for a family of languages sharing commonalities and some variabilities that one can customize for a given application domain. This UML approach does, however, neither solve the consistency problem nor the problems 1 to 4.

UML suffers typically from the non-integrated development of different UML diagram types. Currently some of the diagrams such as use cases and state-chart diagrams are associated by mappings. These mappings are however not refinements [BS03] in the sense of the  $\mathcal{ASM}$  approach, for example, changes in state-chart diagrams do not result in changes of use case diagrams. Often diagrams are not associated at all, see, for instance, the lacking association between class diagrams and use case diagrams in Figure II.

This paper contributes to consistency management of clusters of UML diagrams. Our solution has a number of advantages:

**Faithful coexistence of UML diagrams:** As long as we have been choosing faithful representations of the  $\mathcal{ASM}^{\text{Prgm}}$  by a number of UML diagrams we may bind any change of one of the UML diagrams by  $\mathcal{ASM}^{\text{Contract}}$ . The main aim of these contracts is to support consistency among these diagrams.

**Contract-based refinement of specifications:** Whenever a refinement is applied to the specification then the refinement is only committed if all contract conditions are satisfied. Otherwise we apply an enforcement method such as cascading refinement or default refinement/modification to other parts of the specification, such as rejection of the current refinement or such as the derivation of obligations for later refinement steps.

**Co-evolution of UML diagram clusters:** UML development methodologies often use a number of diagrams at the same time. Their integrated evolution has not been satisfactorily solved so far. A solution cannot be envisioned due to open semantics of UML diagrams. We use  $\mathcal{ASM}$  specifications as the backing specification and demonstrate how UML diagrams can co-evolved and coexist.

The main achievement of our approach is consistency management during the development process. Developers can use the large variety of UML diagrams. Utilization of such varieties of UML diagrams is often required by industrial partners at the moment and is thus a convincing argument in favor for UML.

*Acknowledgement.* We thank Egon Börger and Klaus-Dieter Schewe for their comments, discussions, and advices.

## References

- [BCR00a] Börger, E., Cavarra, A., Riccobene, E.: An ASM semantics for UML activity diagrams. In: AMAST, pp. 293–308 (2000)
- [BCR00b] Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Abstract State Machines, pp. 223–241 (2000)
- [BGS<sup>+</sup>03] Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N., Veanes, M.: Validating use-cases with the AsmL test tool. In: QSIC, pp. 238–246 (2003)
- [Boe76] Boehm, B.W.: Software engineering. IEEE Trans. Computers 25(12), 1226–1241 (1976)
- [Bör03] Börger, E.: The ASM refinement method. Formal Aspects of Computing 15, 237–257 (2003)
- [BS03] Börger, E., Stärk, R.: Abstract state machines - A method for high-level system design and analysis. Springer, Berlin (2003)
- [BST02] Bidoit, M., Sannella, D., Tarlecki, A.: Architectural specifications in CASL. Formal Asp. Comput. 13(3-5), 252–273 (2002)
- [CJ05] Chauvel, F., Jézéquel, J.-M.: Code generation from UML models with semantic variation points. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 54–68. Springer, Heidelberg (2005)
- [DT01] Düsterhöft, A., Thalheim, B.: Conceptual modeling of internet sites. In: Kunii, H.S., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 179–192. Springer, Heidelberg (2001)
- [EB04] Elaasar, M., Briand, L.: An overview on UML consistency management. Technical Report SCE-04-018, Ottawa University (2004)
- [FRT05] Fiedler, G., Raak, T., Thalheim, B.: Database collaboration instead of integration. In: APCCM 2005 (2005)
- [HKKR05] Hitz, M., Kappel, G., Kapsammer, E., Retschitzegger, W.: UML @ Work, 2nd edn. dpunkt, Heidelberg (2005)
- [ISO01] ISO/IEC. 9126-1 (Software engineering - product quality - part 1: Quality model). ISO/IEC JTC1/SC7 N2519 (2001)
- [KN02] Kollingbaum, M., Norman, T.: Supervised interaction - create a web of trust for contracting agents in electronic environments. In: Proc. of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems, pp. 272–279. ACM Press, New York (2002)
- [Mal70] Malzew, A.I.: Algebraic systems. Nauka, Moscow (1970)
- [Obe01] Ober, I.: An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. PhD thesis, Polytechnique de Toulouse (2001)
- [pUM07] The precise UML group (2007), <http://www.cs.york.ac.uk/puml/>
- [Rei84] Reichel, H.: Structural induction on partial algebras. Mathematical research, vol. 18. Akademie-Verlag, Berlin (1984)
- [SMSJ03] Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
- [ST93] Schewe, K.-D., Thalheim, B.: Fundamental concepts of object oriented databases. Acta Cybernetica 11(4), 49–81 (1993)

- [ST99] Schewe, K.-D., Thalheim, B.: Towards a theory of consistency enforcement. *Acta informatica* 36, 97–141 (1999)
- [ST07] Schewe, K.-D., Thalheim, B.: Development of collaboration frameworks for web information systems. In: *IJCAI 2007 (20th Int. Joint Conf on Artificial Intelligence, Section EMC 2007 (Evolutionary models of collaboration), Hyderabad*, pp. 27–32 (2007)
- [Tha00] Thalheim, B.: *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin (2000)
- [Tsi00] Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. Technical Report 2000/3, Technical University of Berlin, Computer Science (2000)
- [Ves05] Vestenicky, V.: Schema integration as view cooperation. PhD thesis, Charles University Prague, Computer Science (2005)

# A Step towards Merging xUML and CSP || B

Helen Treharne<sup>1</sup>, Steve Schneider<sup>1</sup>, Neil Grant<sup>2</sup>, Neil Evans<sup>2</sup>, and Wilson Ifill<sup>2</sup>

<sup>1</sup> Department of Computing, University of Surrey

<sup>2</sup> AWE, Aldermaston, Reading

H.Treharne@surrey.ac.uk

**Abstract.** Much research work has been done on linking UML and formal methods but few have focused on using formal methods to check the integrity of the UML models so that the models can be verified. In this paper we focus on executable UML and on the issues related to concurrent state machines. We show that one integrated formal methods approach, CSP || B, has the potential to be tailored to support reasoning about concurrent state machines and in turn expose any weaknesses in the UML model. We identify future avenues of research so that a system methodology based on executable UML can be enhanced by formal reasoning.

## 1 Introduction

Much research work has been undertaken on examining the relationship between formal methods and informal notations, ranging from linking formal methods with object-oriented paradigms, statecharts, control diagrams and the unified modeling language (UML) [12]. We believe that the research associating formal methods and UML typically falls into two categories. Firstly, there have been attempts to provide a formal semantics for UML and its various versions. The underpinning has either been targeted at the UML meta level or at the level of the various UML diagrams themselves, mainly class diagrams, sequence diagrams, and state diagrams. For example, the B-Method [1] has been used to provide a formal description of the class meta model [10]. There are many examples that provide formal representations for one or more of the UML diagrams. For instance, the recent work of Faitelson et. al [7] focuses on using predicates to describe the associations in class diagrams and Börger et al. [3,4] have concentrated on formalising UML state machines using ASM. The approach presented in this paper falls into this category.

Secondly, UML has often been used to provide a graphical front end for formal notations. This is clearly appealing for formal methods researchers wishing to communicate their ideas in a widely accessible way. For example UML-B [13] uses UML and the B method as an action and constraint language with the goal of deriving a B formal specification. However, in this particular approach the UML notation used does not adhere to UML 2.0 or any accepted subset of it. The UML-like notation used is often an abuse of the standard UML notation. There is significant merit in using graphical notations to describe a formal specification



because it provides a way of clearly describing the overall architecture and the key interaction between components, provided it is not projected as providing a formal underpinning of UML.

There is already a plethora of different approaches attempting to integrate UML and formal methods but none of them focus on using a formal model to provide feedback on the corresponding informal model. Our main motivation is to develop an approach in which the formal representation of a UML model can be used to verify the integrity of the model and assess whether the model is capturing desirable behaviour. We have chosen to focus on state machines and class diagrams and restrict ourselves to using a subset of UML referred to as Executable UML (xUML) [11] since this is one of the preferred methods of our industrial collaborators, AWE. The longer term aim of the collaboration is to merge xUML and formal methods into an integrated systems development methodology. Currently, an xUML model is validated by review and by running simulations on its executable model. In order to develop our methodology we must address issues such as concurrency and sharing of common objects which are prevalent but informally defined in the xUML semantics. In the paper we define a precise model representing the underlying concurrency semantics of an xUML state machine. Furthermore, we believe that the general principles that emerge from the case study presented are a step forward to addressing the above issues.

We use CSP || B [16] as the basis for our formal representation. It is an approach that has been developed to integrate the state-based method B [1] and the process algebra CSP [9]. The approach has focused on identifying compositional verification techniques so that a specification can be verified in smaller parts. We have developed several examples [5,6] which demonstrate the applicability of our compositional techniques. In this paper we extend the scope of CSP || B and identify a new style of specification which lends itself as a formal representation of state machines. Then, by applying the compositional verification techniques we can reason about the state machines of particular xUML models.

The rest of the paper is structured as follows: Section 2 introduces CSP || B and the necessary notation to support the case study, Section 3 introduces the UML subset that we use, Sections 4 and 5 present the case study and its formal representation, Section 6 summarises the verification of the case study model, and Section 7 identifies further work that needs to be done in order to continue to make progress towards integrating CSP || B into a systems development methodology.

## 2 Notation

We identify some preliminary notation relevant to CSP || B. A detailed introduction to CSP operators and the traces model is given in [15]. Similarly, [1] provides details of the B Method. Both CSP and B are supported by industrial tools: FDR [8] is the model checker for CSP and the B-Toolkit [14] is one of the tools available to support the B-Method.

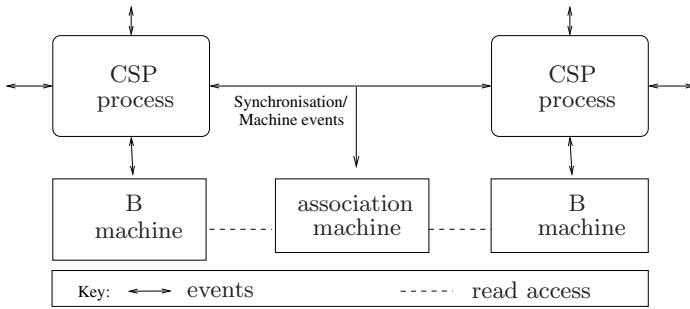


Fig. 1. CSP || B architecture

The CSP || B approach can be used to specify a complex system which is made up of two separate specifications: a number of CSP process descriptions and a collection of B machines (see Figure 1). Our aim when using CSP||B is to factor out “data-rich” aspects of a system into B machines so that the CSP descriptions focus on control flow and interaction patterns. A *machine* in B is a specification construct which encapsulates some variables and provides operations that query and manipulate those variables. For the purposes of this paper we can think of the B operations as methods of UML classes. A B operation takes the form **PRE**  $P$  **THEN**  $S$  **END** where  $P$  is a predicate and  $S$  represents the statements that update the variables. The invariant of a machine captures the constraints on its variables.

The CSP unit of interaction is an *event* which processes perform and on which they may synchronise. Events can be unstructured (such as *enter*) or are generally of the form of a channel name and some values that are passed along that channel. Thus, the occurrence of *addToQueue.hs1.stopMonitor* can be understood as passing the values *hs1* and *stopMonitor* along the channel *addToQueue* (provided *hs1* and *stopMonitor* are of the appropriate type). The occurrence of events is atomic. A sequence of events is referred to as a *trace*.

Processes can be constructed from the following syntax:  $process\ c?x!v \rightarrow P(x)$  binds an input value to  $x$  and outputs a value  $v$  along channel  $c$ , and having accepted  $x$ , it will behave as process  $P(x)$ . The  $!$  is abbreviated with a dot(.) when appropriate. In order for a CSP process to interact with a B machine  $c?x!v$  can be considered as a special kind of event, one which communicates with the B Machine. This event corresponds to the B operation  $x \leftarrow c(v)$ , where the output value  $v$  from the CSP description corresponds to the input parameter of the operation, and the input value  $x$  corresponds to the output of the operation. The B parameters can be identified as being input or output by matching with the corresponding  $!$  and  $?$  in the CSP. There is also prefix choice,  $i : I \rightarrow P(i)$  which provides a choice over the events in the set  $I$  as a prefix to the behaviour of  $P(i)$ . The external choice,  $P_1 \square P_2$ , is initially prepared to behave either as  $P_1$  or as  $P_2$ , with the choice being made on the occurrence of the first event.  $P(i)$  is a process name where  $i$  is an expression. We also allow indexed external choice where  $\square_{i \in I} P_i$  is an external choice between any finite number of processes.

CSP provides a number of parallel composition operators which can be used to combine processes. In this paper we will use the following:

- the *parallel composition* operator,  $P_1 || \{A\} || P_2$ , executes  $P_1$  and  $P_2$  concurrently, requiring that they synchronise on events in the set  $A$ .
- the indexed form of interleaving  $|||_i P_i$  allows us to construct combinations of similar processes which act independently.

In this paper we use a `let within` clause in order to localise some of the CSP definitions so that the CSP descriptions are more modular.

In Section 6 we use CSP traces refinement to examine various behaviours of concurrent state machines in an xUML model. In CSP a process  $Q$  is a trace refinement of another process  $P$  if all possible sequences of events which  $Q$  can perform are also possible for  $P$ . This relationship is written  $P \sqsubseteq_T Q$ . If we consider  $P$  to be a property which determines allowable behaviour, then we can think of a successful refinement check, i.e.  $P \sqsubseteq_T Q$ , as saying that  $Q$  is a safe model; implicitly no wrong events will be allowed. Conversely, we can also use  $P$  to describe what should not be possible. Then if  $P \sqsubseteq_T Q$  is a check that passes it means that  $Q$  will not exhibit any undesirable behaviour. If the  $P \sqsubseteq_T Q$  is a check that fails it means that  $Q$  is capable of engaging in undesirable behaviour.

The CSP||B approach supports compositional verification enabling us to focus on the CSP descriptions and B descriptions in isolation. Theoretical results in 16 allow us to deduce results about the system as a whole from the individually verified parts. The benefit of this is that we can apply CSP verification tools and B tools to appropriate parts of the abstract model without ever having to consider the model as one large specification during its verification. In this paper we conduct trace refinement checks on the CSP descriptions in FDR and then deduce that the result of the check holds for both the CSP and B descriptions.

### 3 Executable UML Overview

Executable UML (xUML) 11 is a coherent subset of UML that has been developed so that it is possible to define execution semantics without ambiguity whilst leaving the possibility for targeting the models of systems to different types of implementation languages, including C, Ada and VHDL. xUML includes an Action Specification Language (ASL) which provides all the necessary conditional logic and primitive actions to manipulate the UML object model. One of the benefits of using ASL is the ability to describe a system independently from its runtime platform. Typically, there are six parts to an xUML model which describe a system:

1. Use Case Diagrams capture the requirements of the system,
2. Domain Models partition the system into separate parts,
3. Sequence Diagrams define the interfaces of the various domains,
4. Class Diagrams specify the classes in each domain,

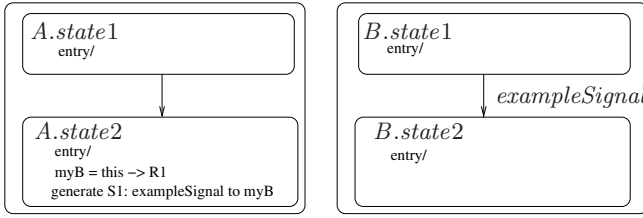


Fig. 2. Two state machines

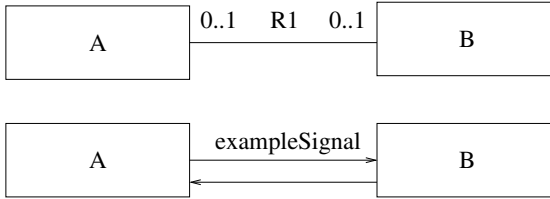


Fig. 3. Example Class and Collaboration Diagram

- 5. Collaboration Diagrams define the class interfaces,
- 6. State Machines specify the behaviour of each class.

In this paper we focus on class diagrams and state machines. The novelty of our approach is the ability to analyse a collection of state models within a system when they are composed concurrently.

### 3.1 State Machine Overview

Figure 2 illustrates two state machines (without creation and deletion states). The state machines are related to the two classes in Figure 3. The relationship *R1* represents the association between class *A* and *B*. Clearly there is a relationship between diagrams in an xUML model. From Figure 3 we can see that the association, *R1*, and the signal, *exampleSignal*, on the collaboration diagram are being used in the definition of the state machines in Figure 2.

In general, a state machine comprises a creation transition, a creation state, any number of intermediate states, signals, a deletion transition, and a deletion state. Each state can have an entry action which enables the definition of a sequence of ASL commands. Actions are executed upon receipt of a signal or as a result of object creation or deletion. In the entry action of *A.state2*, in Figure 2, *myB* is a local variable representing the instance handle associated with *B*'s state machine. The first assignment of the entry action assigns a value to *myB*. This instance handle can be found by navigating from *this* through *R1* (*this -> R1* in the figure), where *this* refers to the instance handle of the object associated with *A*.

The generation of a signal is also defined as part of the entry action. Signals enable the transition between states. It is possible to send a signal to

a specified object using the “`generate <<signalId>>: <<signalname>> to <<instancehandle>>`” construct. For example, *exampleSignal* sends a signal to *myB* which we have already shown to be a valid instance handle. If B’s state machine is in *B.state1* and has done its entry action then the processing of the *exampleSignal* allows us to move from *B.state1* to *B.state2*.

### 3.2 Concurrency Models

An xUML model of a system must allow for the most general level of concurrency that could exist at runtime without which any analysis of the system may not reveal potentially unexpected behaviour. In xUML three different concurrency models have been identified [11]: simultaneous operation, interleaved operation, and sequential operation. In this paper we focus on simultaneous operation because it is the most general of the models which allows objects to execute their entry actions concurrently. It means that we can analyse whether the most complex interaction possible within a system should be permitted. Consider a set of three objects, *A*, *B*, and *C*, each with an associated state machine. Suppose in *A*’s state machine a signal is sent to both *B* and *C*. The simultaneous operation model means that the ASL commands in *B* and *C*’s actions could be running at the same time. Similarly, if *A* is in the middle of processing its entry action and sends a signal to *B* then *B* can also begin processing its entry action. Thus, *A* and *B*’s actions are allowed to run at the same time.

In order to provide an accurate representation of the underlying communication mechanism for processing signals we must consider signals in more depth.

### 3.3 Modelling Signals

When a signal is generated by an instance then its destination instance is stated in the ASL statement. The destination instance will process the signal, which could be another instance in the model or the instance that generated the signal. The latter is referred to as a *self-directed* signals and is labelled: “`generate <<signalId>>: <<signalname>> to this`”. All signals can carry parameters.

Note that a signal does not interrupt the processing of an entry action by an instance; this is the case in all three concurrency models identified in Section 3.2. Rather a signal is queued and processed once that instance is ready. If more than one signal is received by a particular instance, from instances other than itself, they are queued and processed in the order in which they are received.

Self-directed signals are queued in a similar way to other signals. However, they differ because any self-directed signal which is already queued will always be processed by an instance before any signal from other instances. Once all the self-directed signals are processed then the state model for a particular instance will be in a position to deal with any outstanding signals received from other instances. Without this xUML constraint many more combinations of signal ordering would exist.

In xUML models (self-directed) timeout signals can also be generated. Timeout signals do not take precedence over other self-directed signals, and they are

```

SignalQueues(i) =
let   Q(tr1, tr2) =
      if (#tr1 + #tr2 ≥ CAPACITY)
      then if (tr1 == ⟨⟩)
            then removeFromQueue.i!head(tr2) → Q(tr1, tail(tr2))
            else removeFromQueue.i!head(tr1) → Q(tail(tr1), tr2)
      else
      if (tr1 == ⟨⟩)
      then if (tr2 == ⟨⟩) then
            addToSelfQueue.i?sig → Q(⟨sig⟩, ⟨⟩)
            □ addToQueue.i?sig → Q(⟨⟩, ⟨sig⟩)
          else
            addToSelfQueue.i?sig → Q(⟨sig⟩, tr2)
            □ addToQueue.i?sig → Q(⟨⟩, tr2 ∪ ⟨sig⟩)
            □ removeFromQueue.i!head(tr2) → Q(⟨⟩, tail(tr2))
      else
            addToSelfQueue.i?sig → Q(tr1 ∪ ⟨sig⟩, tr2)
            □ addToQueue.i?sig → Q(tr1, tr2 ∪ ⟨sig⟩)
            □ removeFromQueue.i!head(tr1) → Q(tail(tr1), tr2)

within   Q(⟨⟩, ⟨⟩)
CAPACITY = 2

```

**Fig. 4.** Signal Queues for one instance

treated in exactly the same way as signals from other instances. This is important so that the model does not provide a way of being able to subvert the ordering of signal communication between instances. The syntax for timeout signals is slightly different: “generate <<signalId>>:Set\_Timer(..., this)”.

## 4 Modelling Signal Queues in CSP

From the above informal description of signal ordering and processing we clearly need to maintain two queues for each instance: one that processes self-directed signals ( $tr_1$ ) and another that processes general signals ( $tr_2$ ), as shown in Figure 4. Both  $tr_1$  and  $tr_2$  are modelled as sequences in CSP.

Initially, no signals are queued and as a consequence the queueing process,  $SignalQueues(i)$ , for an arbitrary instance  $i$  initialises both queues to empty sequences ( $\langle \rangle$ ). One of the most interesting cases in the evolution of the process is where the queues are not full and the self-directed signals queue is not empty. Then the allowable behaviour of an instance  $i$  is to accept a further signal, along either channel  $addToSelfQueue$  or  $addToQueue$ , and append it to the appropriate queue ( $tr_1 \cup \langle sig \rangle$  or  $tr_2 \cup \langle sig \rangle$  respectively). The notion of adding a signal to a queue is the formalisation of the ASL “generate SigId: sig to i” construct. The  $SignalQueues$  process also allows the processing of the signal at the head of the self-directed queue by communicating signal information ( $head(tr_1)$ ) along the channel  $removeFromQueue$ .

We can use the *removeFromQueue* channel for both queues because the conditional branching within the process  $Q$  will guarantee that the correct signal is processed. In Section 5.1, we shall see that a communication along the *removeFromQueue* channel represents processing a signal and subsequently the triggering of the entry action of a particular state.

The *SignalQueues* process is a general representation for processing signals. The only tailoring needed to apply it to a particular model is to define the values to be passed along the channels, and those values would be the signal names referenced in the state machines.

## 5 Case Study

A heating controller system is made up of two xUML classes, *HeatedSpace* and *HeatingPeriod*, as shown in Figure 5. An instance of *HeatedSpace* models a room that is being heated and has two attributes, the current room temperature and the desired room temperature. The *HeatingPeriod* represents the concept of an interval of time and the desired temperature during that time. The system supports many instances of each class. Each *HeatingPeriod* instance can be uniquely identified with a *HeatedSpace* instance. Each class is associated with a state machine namely *HeatingPeriodStateMachine* or *HeatedSpaceStateMachine*, illustrated in Figures 6 and 7 respectively.

The *HeatingPeriodStateMachine* is described in terms of three states and includes the notion of a creation state. A deletion state is omitted for reasons of space. The state machines of the heating period instances toggle between two states **Pending** and **Current** sending signals, *startMonitor()* and *stopMonitor()*, to their associated heated space instances to indicate when the heating period interval becomes active and inactive.

The *HeatedSpaceStateMachine* aims to maintain the temperature of the room at the desired temperature of the heating period. It involves three main states but many more transitions. The creation and deletion states have been omitted from the diagram. Therefore, in our system we will assume a statically populated set of heated space instances. Initially, a heated space instance is in the **NotMonitored** state. Upon receipt of a *startMonitor()* signal it moves into the **AboveTemp** state and executes the state's entry action. It compares the current room temperature with the desired room temperature and if it is too cold, it generates a self-directed signal *tooCold()* and calls an external method, *heatOn* to turn on the

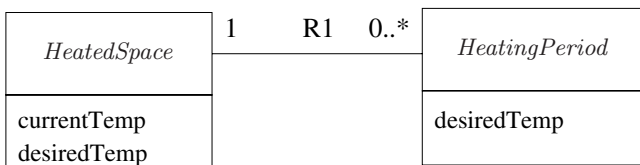


Fig. 5. Classes for Heating System Example

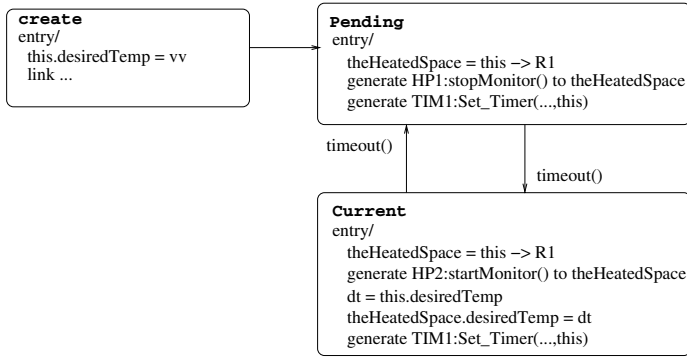


Fig. 6. Heating Period State Machine

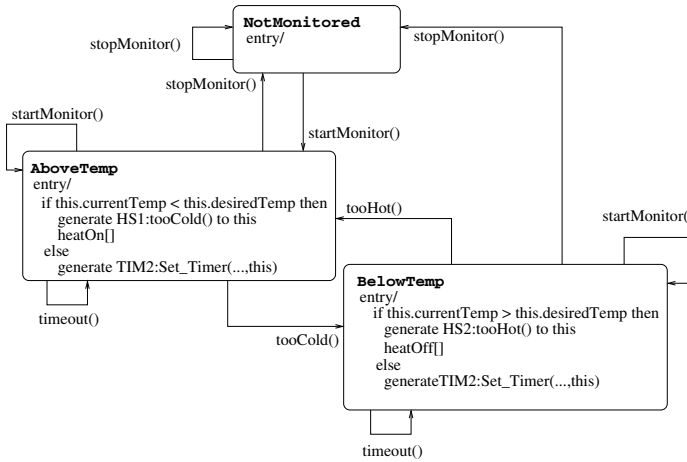


Fig. 7. Heated Space State Machine

heater, otherwise if it is too hot, it generates a *timeout()* signal so that it can re-execute the entry action later on.

The *BelowTemp* state executes its entry action upon receipt of a *tooCold()*, *startMonitor()* or a *timeout()* signal. The current room temperature is compared with the desired room temperature and this time if it is too hot, it generates a self-directed signal *tooHot()* and calls an external method, *heatOff*, to turn off the heater, otherwise a timer is set and a timeout signal is generated.

### 5.1 What Shall We Model Formally?

We are interested in examining the inter-relationships between signals and states and whether there are any ambiguities in the heating system. Therefore, CSP is used to model an instance’s state machine as a parameterised recursive process which captures the behaviour of the entry actions and signal interactions.



```

HP_Ctrl(hp) =
let  HP_StartEntry = □v ∈ NAT setDesiredTemp.hp!v → link.hp?hs → HP_StartState
    HP_StartState = HP_PendingEntry
    HP_PendingEntry = navigateToR1.hp?hs →
                        addToQueue.hs!stopMonitor →
                        addToQueue.hp!timeout → HP_PendingState
    HP_PendingState = (removeFromQueue.hp!timeout → HP_CurrentEntry)
                    □
                    (state.hp!pendingState → HP_PendingState)
    HP_CurrentEntry = navigateToR1.hp?hs →
                        addToQueue.hs!startMonitor →
                        getDesiredTemp.hp?dt →
                        setDesiredTemp.hs!dt →
                        addToQueue.hp!timeout → HP_CurrentState
    HP_CurrentState = (removeFromQueue.hp!timeout → HP_PendingEntry)
                    □
                    (state.hp!currentState → HP_CurrentState)
within  HP_StartEntry

```

**Fig. 8.** Heating Period Controller

Within the entry actions of these instances the values of their attributes are queried and modified by using method calls. The calls are modelled in the CSP but are handled by B components. The B components form part of a formal representation of an xUML model in order to provide an appropriate model of attributes and methods.

**Modelling a Heating Period State Machine.** The *HeatingPeriod* controller process, defined in Figure 8, is a parameterised process for a heating period instance. The parameter *hp* represents the heating period instance handle. Within the process definition there are two process equations for each state: one entry process and one state process (*HP\_StartEntry* and *HP\_StartState* respectively). Event sequences in *HP\_StartEntry*, *HP\_PendingEntry* and *HP\_CurrentEntry* are those representing the ASL entry actions. In the *HP\_PendingEntry* process we note that the event *navigate.hp?hs* determines which heated space the *hp* instance is related to, and this information is stored within a B component.

A B machine is defined to capture the relationship between instances. In our example we define a variable *R1* in a machine called *AssocB* and we capture the relationship identified in Figure 5 in its invariant as follows:

$$R1 : \text{heatingPeriodInstances} \rightarrow \text{heatedSpaceInstances}$$

where the function corresponds to the multiplicity between a heating period and a particular heated space. The *heatingPeriodInstances* and *heatedSpaceInstances* variables represent the sets of instances of the *HeatingPeriod* and *HeatedSpace* classes. They are defined in their own respective machines: *HeatingPeriodB*

and *HeatedSpaceB*. The *navigate* operation queries the value of *R1* as follows:

```

hs ← navigate ( hp ) ≐
  PRE hp ∈ heatingPeriodInstances
  THEN
    hs := R1 ( hp )
  END

```

Once an entry action has been completed we can proceed to an appropriate state process in which we must be prepared to respond to particular signals. For example, in the *HP\_PendingState* and *HP\_CurrentState* processes, the event *removeFromQueue.hp!timeout* represents the process performing a timeout signal. In the *HP\_StartState* we proceed immediately to perform events in the *HP\_PendingEntry* process. Typically, in an xUML model we would have to respond to an external stimulus in order to move from the creation state to the next state. This example has simplified the details related to creation and deletion so that we can concentrate on analysing the complex interactions between the intermediate states of multiple instances in Section 6.

Notice also that once an instance is in a given state, we use the channel *state* to communicate that we are in that particular state. For example, the event *state.hp1!pendingState* reports that the first heating period instance, *hp1*, is in the **Pending** state of its state machine. These state events are used in the analysis in order to track which state a state machine is in currently.

**Modelling a Heated Space State Machine.** The formalisation of the *HeatedSpaceStateMachine*, represented by the *HS\_Ctrl* parameterised process, follows a similar pattern to that of the *HP\_Ctrl* process. Figure 9 contains the interesting process definitions: *HS\_AboveTempEntry*, *HS\_AboveTempState* and *HS\_BelowTempEntry*. When the *HS\_AboveTempEntry* process evolves its behaviour to the *HS\_AboveTempState* process the model must be prepared to respond to several different signals. Communications along the *removeFromQueue* channel indicate processing a signal. For example, performing the event *removeFromQueue.hs1!stopMonitor* represents the stop monitor signal being processed by the first heated space instance, *hs1*, and we will be referring to this event again in Section 6.

In the process expressions we retrieve the current and desired temperature of the room, *hs*, by communicating values along the *getCurrentTemp* and *getDesiredTemp* channels. These communications relate to B operations which return the attribute values. The *getCurrentTemp* operation is defined in the *HeatedSpaceB* machine as follows:

```

currentTemp ← getCurrentTemp ( hs ) ≐
  PRE hs ∈ heatedSpaceInstances
  THEN
    currentTemp := currentTempB ( hs )
  END

```

```

HS_AboveTempEntry =
  getCurrentTemp.hs?currentTemp → getDesiredTemp.hs?desiredTemp →
  ( if ( currentTemp < desiredTemp ) then
    addToSelfQueue.hs!tooCold → heatOn → HS_AboveTempState
  else
    addToQueue.hs!timeout → HS_AboveTempState )

HS_AboveTempState =
  (removeFromQueue.hs!stopMonitor → HS_NotMonitoredEntry)
  □
  (removeFromQueue.hs!tooCold → HS_BelowTempEntry)
  □
  (removeFromQueue.hs!timeout → HS_AboveTempEntry)
  □
  (removeFromQueue.hs!startMonitor → HS_AboveTempEntry)
  □
  (state.hs!aboveTempState → HS_AboveTempState)

HS_BelowTempEntry =
  getCurrentTemp.hs?currentTemp → getDesiredTemp.hs?desiredTemp →
  ( if ( currentTemp > desiredTemp ) then
    addToSelfQueue.hs!tooHot → heatOff → HS_BelowTempState
  else
    addToQueue.hs!timeout → HS_BelowTempState )

```

**Fig. 9.** Fragment of Heated Space Controller

The *currentTempB* variable in the B machine retrieves the attribute value for a particular instance. The values from the B operations are compared and the heater is either switched on, represented by performing the *heatOn* event, or off, represented by performing the *heatOff* event.

**Modelling Combinations of State Machines.** We need to compose the descriptions of all the possible instances together so that we have a simultaneous concurrent model of the instances of the heating system. Note that each instance is an independent process but they can communicate with each other via signals. In Figure 10 we define, using CSP, a collection of independent heating period instances and a collection of independent heated space instances, and these collections synchronise with the collection of signal queues. The resulting process, *SYSTEM\_Ctrls*, is a partial description of the behaviour of the state machines. To complete the description we include the B formal model of the instances (including their attributes and methods) and their association constraints. This is illustrated in Figure 11 and in terms of a CSP || B model this would be expressed as:

$$SYSTEM\_Ctrls \parallel HeatedSpaceB \parallel HeatingPeriodB \parallel AssocB$$

where *HeatedSpaceB* and *HeatingPeriodB* are the B encapsulation of the formal description of the instances of the heating system, and *AssocB* is a representation of the associations between the instances.

$$\begin{aligned}
 SYSTEM\_Ctrls = & ((\parallel_{hp \in PERIODS} HP\_Ctrl(hp)) \parallel (\parallel_{hs \in SPACES} HS\_Ctrl(hs))) \\
 & [\{ \{ addToSelfQueue, addToQueue, removeFromQueue \} \}] \\
 & (\parallel_{i \in union(SPACES, PERIODS)} SignalQueues(i))
 \end{aligned}$$

Fig. 10. Full Heating System Description

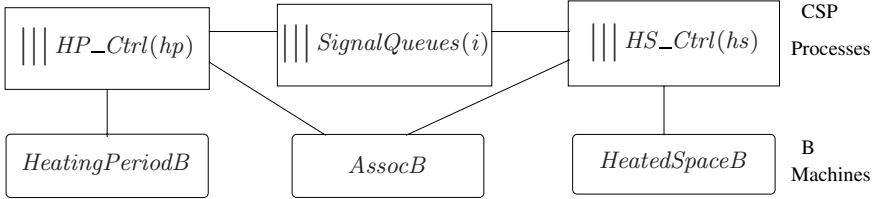


Fig. 11. Architecture of Heating System

## 6 Analysis

Our analysis focuses on two kinds of properties. Firstly, we check whether a signal is possible when the xUML model is in specific concurrent states. It enables us to highlight any undesirable behaviour within the model. Secondly, we analyse the integrity of attribute values when the xUML is in specific concurrent states. It ensures that the system is modifying and accessing its state safely. This is important if the xUML model allows the use of shared attributes to ensure that updates are controlled.

We discussed in Section 2 that a property can be shown to be refined by a model using CSP. Therefore, we demonstrate that the  $SYSTEM\_Ctrls$  process is a refinement of certain desired properties. We perform the analysis by examining the controllers in isolation but the theoretical foundations of CSP  $\parallel$  B ensure that the results of the analysis apply to the whole CSP  $\parallel$  B specification. Since the CSP  $\parallel$  B specification is a formal model of the xUML state machines and classes, the analysis in turn provides results about the xUML model itself.

The  $SIGNAL\_PROPERTY$  process captures the first kind of property, and is defined in Appendix A. It states that a particular signal  $c$  should not be possible when the system is in a particular pair of states  $a$  and  $b$ . Each refinement check related to this property is performed twice: once when we consider a single heating period ( $hp1$ ) and a single related heated space ( $hs1$ ) and also when we consider two heating periods ( $hp1$  and  $hp2$ ) and a single related heated space ( $hs1$ ). We refer to these checks as *Single Instance* and *Multiple Instance* checks respectively. We must carry out these refinement checks on a variety of instantiations which satisfy the association constraints of the class diagram so that we consider the impact of different instance combinations on the model’s behaviour.

One pair of states in the heating system is ( $Current, BelowTemp$ ) and  $stopMonitor()$  is a signal within the model. If the signal can be processed from that pair of states then the system exhibits undesirable behaviour because it could lead to the heater being switched on and the system not monitoring this.

Therefore, we need to verify that this signal is not allowed to be processed by the heating system when in this particular pair of states using the following:

$$\text{SIGNAL\_PROPERTY}(hp1.currentState, hs1.belowTempState, \\ \text{removeFromQueue.hs1.stopMonitor}) \sqsubseteq_T \text{SYSTEM\_Ctrls}$$

The *Single Instance* check passes but the *Multiple Instance* check fails giving rise to undesirable behaviour. This behaviour occurs because the heating periods *hp1* and *hp2* overlap. Currently, the xUML model does not explicitly state whether this should be allowed or not, and therefore the formal analysis has revealed a potential problem which would need to be addressed in the xUML model.

It will of course be important to perform a suite of checks in a systematic way based on different combinations of states and signals. Appendix [A.1](#) gives further examples related to the first property.

An example of the second kind of property, identified above, is that the desired temperature of an active heating period instance is the same as the desired temperature of the heated space instance. For example, if the heating period instance *hp1* is related to a heated space instance *hs1* and *hp1* is active, i.e. in **CurrentState**, then the desired temperature of those instances must not be different. We again used a CSP refinement check to analyse this property and noted that the system can evolve to a behaviour in which the instance states are different. The reason for this undesirable behaviour was again due to the behaviour of overlapping heating periods within the model. Each heating period sets the desired temperature of the heated space by direct assignment of the attribute of the heated space instance. Therefore, when two heating period instances make assignments only the most recent one is recorded resulting in a potential inconsistency of the temperature values.

## 7 Discussion

The work presented in this paper is exploratory work for an industrial project. We discussed how to develop a formal model from xUML state machines and class diagrams based on a running example. Further work is needed in order to identify general transformation patterns from xUML to CSP || B. Whilst developing the example we achieved a better understanding of the xUML concurrency models which were previously undocumented. Our signals queueing model has formalised how signals are processed and clarifies the fact that timeouts do not have to be serviced immediately.

The purpose of building the formal model was to be able to conduct rigorous analysis in order to provide confidence in the consistency of the xUML model and/or to reveal potential problems. Our analysis of the case study revealed that the xUML model did not prohibit overlapping heating periods but did not handle them adequately. Our future work will address how to feed this information back into the xUML model in a rigorous way.

The formal analysis was based on examining combinations of concurrent states from the state machines. Clearly identifying these combinations by hand will not

be acceptable in practice and therefore we need to explore how to automate their creation. We also need to investigate how the required properties of an xUML model can be automatically generated as CSP processes in order to be used in CSP refinement checks. Analysing system behaviour from concurrent states is not possible within the xUML tools available.

In our analysis we relied upon an abstraction of the timing details of the state machines. The HUGO/RT tool [2] automatically converts timed state machines into corresponding timed automata which can then be model checked using the UPPAAL tool. The HUGO/RT tool takes the XMI output from UML editors as its input. More hierarchical structures are permitted in the state machines supported by HUGO/RT but the computation being performed within a state is simplified.

In this paper we have focused on analysing the interactions that exist in a model. A further benefit of having a formal model is that we may be able to analyse for missing signals that are required to ensure that the xUML model is consistent. For example, we may be able to use deadlock freedom checks to verify that all generated signals are processed. It may also be possible to reason about possible inconsistencies in the associations of a class diagram.

**Acknowledgements.** The authors are grateful to AWE for funding this exploratory work. Thanks also to Ian Wilkie and Chris Raistrick for discussions on xUML. The reviewers also provided valuable feedback.

## References

1. Abrial, J.-R.: *The B Book: Assigning Programs to Meanings*. CUP, Cambridge (1996)
2. Knapp, A., Merz, S., Rauh, C.: Model Checking Timed UML State Machines and Collaborations. In: Damm, W., Olderog, E.-R. (eds.) *FTRTFT 2002*. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
3. Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASM. *Information & Software Technology* 46(5), 287–292 (2004)
4. Cavarra, A., Riccobene, E., Scandurra, P.: A framework to Simulate UML Models: Moving from a Semi-formal to a Formal Environment. In: *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, pp. 1519–1523 (2004)
5. Evans, N., Treharne, H.: Investigating a File Transfer Protocol Using CSP and B. *SoSym Journal* (2005)
6. Evans, N., Treharne, H., Laleau, R., Frappier, M.: How to Verify Dynamic Properties of Information Systems. In: *2nd International Conference on Software Engineering and Formal Methods*. IEEE, China (2004)
7. Faitelson, D., Welch, J., Davies, J.: From predicates to programs: the semantics of a method language. In: *Proceedings of SBMF 2005*. *Electronic Notes in Theoretical Computer Science* (2005)
8. Formal Systems (Europe) Ltd.: *FDR2 User Manual* (2003)
9. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (1985)

10. Laleau, R., Polack, F.: Coming and going from UML to B: a proposal to support traceability in rigorous IS development. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 517–534. Springer, Heidelberg (2002)
11. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I.: Model Driven Architecture with Executable UML. Cambridge University Press, Cambridge (2004)
12. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1998)
13. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. ACM Transactions on Software Engineering and Methodology (2006)
14. Neilson, D., Sorensen, I.H.: The B-Technologies: a system for computer aided programming. B-Core (UK) Ltd (1999), <http://www.b-core.com>
15. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley, Chichester (1999)
16. Schneider, S., Treharne, H.: CSP Theorems for Communicating B machines. FACS 17(4) (2004)

## A Detailed Explanation of the Analysis Performed

$$\begin{aligned} \text{SIGNAL\_PROPERTY}(a, b, c) = \\ (state?x \rightarrow \text{if } (x == a) \text{ then } SP(a, b, c) \\ \text{else } \text{SIGNAL\_PROPERTY}(a, b, c)) \end{aligned}$$

□

$$(y : (\Sigma - \{ | \text{state} | \}) \rightarrow \text{SIGNAL\_PROPERTY}(a, b, c))$$

$$\begin{aligned} SP(a, b, c) = \\ (state?x \rightarrow \text{if } (x == b) \text{ then } SP1(a, b, c) \\ \text{else } \text{SIGNAL\_PROPERTY}(a, b, c)) \end{aligned}$$

□

$$(y : (\Sigma - \{ | \text{state} | \}) \rightarrow \text{SIGNAL\_PROPERTY}(a, b, c))$$

$$SP1(a, b, c) = y : (\Sigma - \{c\}) \rightarrow \text{SIGNAL\_PROPERTY}(a, b, c)$$

The process checks that if the first state has been observed, *state.a*, then it monitors whether the second state is observed, *state.b*. If it is observed then the signal event, represented by *c*, must not be allowed to occur. In all other cases any event from the set of all events in the system,  $\Sigma$ , is allowed to be observed apart from those which simply report the state of the system.

### A.1 Analysis of Property 1

In order to carry out a full analysis we identify a combination of states. This is a Cartesian product of the names of the states in the heating period state machine and heated space machine (from Figures 6 and 7 respectively):

$$\{\text{Pending, Current}\} * \{\text{NotMonitored, AboveTemp, BelowTemp}\}$$

For each particular pair of states, we focus on one of the states and a signal that is possible for that state, and ask whether it is desirable for a particular signal to

**Table 1.** Verifying desirable behaviour of *SYSTEM\_Ctrls*

Test	<i>Single Instance</i>	N states	N transitions	<i>Multiple Instances</i>	N states	N transitions
1	PASSED	182	641	PASSED	1,827	9,383
2	PASSED	399	1339	PASSED	11,946	55,617
3	PASSED	727	6,982	PASSED	6,545	32,188
4	PASSED	653	5,642	FAIL	5,304	26,367

be allowed in that pair of states. If it is desirable behaviour then we need to check that the signal is possible when the model is populated with multiple instances. Conversely, if it is not desirable behaviour then we need to check that this signal is not possible when the model is populated with multiple instances. We assess whether allowing a particular signal is desirable or undesirable behaviour of a model for each pair of concurrent states.

Table 1 summarises an example set of tests that assess the behaviour of the heating system model. The tests are split into both *Single Instance* checks and *Multiple Instance* checks. The first pair of tests are related to self-directed signals whilst the second pair are related to signals to be processed that are generated by other instances. Tests 1 and 2 are related to testing the *tooCold()* signal in the **Current** state but in different pairs of concurrent states. Test 1 considers the pair (**Current**, **BelowTemp**) and shows that the signal is not possible in that state. Test 2 considers the pair (**Current**, **AboveTemp**) and shows that the signal is possible in that state. Appropriate refinement checks were applied in both cases and their success confirms that the model behaves as expected.

Tests 3 and 4 focus on verifying whether the xUML model contains undesirable behaviour in relation to processing other signals. They are related to the *stopMonitor()* signal in the **BelowTemp** state but in different pairs of concurrent states. We have already discussed the results from Test 4 in Section 6. Test 3 considers the pair (**Pending**, **BelowTemp**) and allowing the signal in this state is considered as allowable behaviour. Therefore, the refinement check:

$$\text{not}(\text{SIGNAL\_PROPERTY}(hp1.pendingState, hs1.belowTempState, \\ \text{removeFromQueue.hs1.stopMonitor}) \sqsubseteq_T \text{SYSTEM\_Ctrls})$$

was performed which states that if the model is in the two states then the *stopMonitor()* signal is possible because it can be processed when the heating system is in this pair of concurrent states. Note, it is easier to write the desired property, captured in the process *SIGNAL\_PROPERTY*, in terms of what is not possible and so we negate the result in order to achieve the result required for our analysis. Test 3 passes in both sub-tests which means that the xUML model exhibits the desired behaviour when processing the *stopMonitor()* signal in this combination of states. Test 3 also gives us confidence that *stopMonitor()* signal is appropriate in other parts of the model.



# CoreASM Plug-In Architecture

R. Farahbod<sup>1</sup>, V. Gervasi<sup>2</sup>, U. Glässer<sup>1</sup>, and G. Ma<sup>1</sup>

<sup>1</sup> Computing Science, Simon Fraser University, Burnaby, B.C., Canada

<sup>2</sup> Dipartimento di Informatica, Università di Pisa, Italy

**Abstract.** Abstract State Machines are known for their versatility in modeling of algorithms, architectures, languages, protocols, and virtually all kinds of sequential, parallel, and distributed systems. CoreASM is a novel executable ASM language which emphasizes freedom of experimentation and supports the evolutionary nature of design as a product of creativity. The CoreASM engine, the heart of the CoreASM tool suite, is based on an extensible architecture which supports various extensions through *plug-ins*. In this paper, we explore the plug-in architecture of the CoreASM engine and demonstrate its potentials by looking into two implemented plug-ins.

## 1 Introduction

CoreASM [1] is a lean, executable specification language together with a supporting tool environment for high-level design, experimental validation and formal verification (where appropriate) of Abstract State Machine (ASM) [2] models [4].

The CoreASM language focuses on the early phases of the software design process, emphasizing freedom of experimentation and the evolutionary nature of design being a creative activity. It encourages rapid prototyping of abstract machine models for testing and design space exploration facilitating agile software development [3]. By minimizing the need for encoding in mapping the problem space to a formal model, it allows writing highly abstract and concise specifications—starting with mathematically-oriented, abstract and untyped models, gradually refining them down to more concrete versions with a degree of detail and precision as needed.

The CoreASM environment consists of a platform-independent engine for executing the language and a GUI for interactive visualization and control of simulation runs. The engine comes with a sophisticated and well defined interface, called Control API, thereby enabling future development and integration of complementary tools, e.g., for symbolic model checking and automated test generation. The design of CoreASM is novel and the underlying principles are unprecedented among the existing executable ASM languages [14].

The purpose of this paper is to elucidate and illustrate the central role and the characteristic features of the CoreASM plug-in architecture as the conceptual foundation of the extensibility framework. The design is streamlined towards

---

<sup>1</sup> CoreASM is an Open Source project and is readily available at [www.coreasm.org](http://www.coreasm.org).

flexible extensibility of the language definition and underlying execution engine. This aspect deserves special attention, as it has not been addressed in detail in any of our previous papers [15,6].

In principle, there are three basic dimensions being considered for extending and altering CoreASM by means of plug-ins mechanisms, respectively related to: (i) data structures, (ii) control structures, and (iii) the execution model. The possibility of conveniently extending data structures as needed is the most prominent one, extensively discussed in the theoretical ASM literature, e.g. in [7,8], where the concept of *background* refers to an implicitly given part of an abstract machine state, assuming that it provides whatever standard means are normally supposed to be available in a given application context [7]:

“A realistic description of algorithms would involve quite a rich background, including numbers, sets, multisets, maps, sequences, and the like, since all these things are generally taken to be available when designing algorithms.”

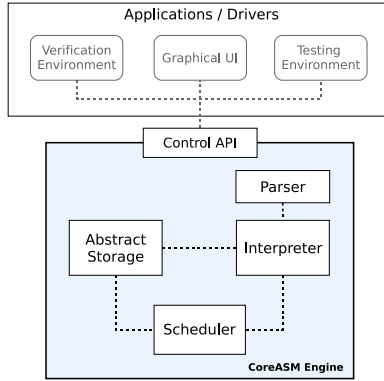
Control structures can be extended with respect to both new syntactic constructs that are semantically meaningful and those that do provide syntactic sugar only (i.e., the semantics of which could also be expressed by means of in-language transformations). Finally, the need for altering or extending the execution model is justified by pragmatic considerations. The execution model refers to dynamic features of CoreASM, including scheduling policies, exception handling, and instrumentation of program execution for analytical purposes.

In Section 2 we present the basic components of the CoreASM engine, explore the execution lifecycle of the engine and its control state model, and discuss the micro-kernel approach to the design of the engine. In Section 3 we look into the extensibility mechanism of the engine and introduce the CoreASM Plug-in Framework, and conclude this section by introducing two recently developed plug-ins of the engine. Section 4 discusses related work, both in terms of ASM execution engines and of extensible architectures. Section 5 concludes the paper.

## 2 CoreASM Architecture

The CoreASM engine consists of four basic modules: *Parser*, *Interpreter*, *Scheduler*, and *Abstract Storage* (Figure 1). The latter three jointly perform asynchronous execution cycles of the engine to simulate an ASM run. The engine interacts with the environment through a component, called *Control API*, that acts as an interface between the engine and its environment and provides an API for various operations such as loading a CoreASM specification, starting an ASM run, or performing a single execution step.

The architecture of the CoreASM engine is partitioned along two dimensions. The first one identifies the four main modules and their relationships. The second dimension distinguishes between what is in the *kernel* of the system—thus implicitly defining the extreme bare bones ASM model—and what is instead provided by plug-ins extending the engine (see Section 3).



**Fig. 1.** Overall Architecture of CoreASM

In the following subsections, we look into the main components of the engine, explain its control state model, and discuss the microkernel design.

## 2.1 CoreASM Engine

Inside the engine, the parser, when reading a CoreASM specification, generates for each individual program an annotated parse tree as input for the interpreter. Each node of a parse tree potentially has a reference to the plug-in that defines the corresponding syntax.

The interpreter evaluates programs and rules, possibly calling upon plug-ins to perform expression evaluation or to interpret certain rules. By traversing a parse tree, it generates a multiset of *update instructions*, each of which represents either an update, or an arbitrary instruction to be processed at a later stage by means of plug-ins that generate the actual updates<sup>2</sup>. The interpreter interacts with the abstract storage to retrieve data from the current state and to incrementally create the update set that, once it is complete and consistent, will produce the next state.

The abstract storage maintains a representation of the current state of the machine that is being simulated. A state is modeled as a map from locations to values (elements) from a universe `ELEMENT`. Additionally, it provides auxiliary information on the locations in the current state, such as the range and domain of functions or the background to which a particular value belongs.

The scheduler orchestrates the individual computation steps of an ASM run. In a sequential run, upon receiving a *STEP* command from the Control API, it invokes the interpreter and then waits for the interpreter to finish the evaluation of the program. Next, it instructs the abstract storage to aggregate the update instructions and fire the resulting update set (if consistent). It then notifies the environment through the Control API of the results.

<sup>2</sup> Where no confusion can arise, in the following we use the generic term “updates” to refer both to actual updates and to update instructions.

In a partially ordered run [2], the scheduler also controls the concurrent execution of ASM agents. At the beginning of each computation step, it chooses a subset of agents which will actively participate in the computation of the next update set. The scheduler directly interacts with the abstract storage to retrieve the current set of agents, to assign the current executing agent, and to collect the update set generated by the interpretation of all the agents’ programs. Updates are then fired and the environment is notified as in case of the sequential run.

### 2.2 Engine Life-Cycle

Overall, the process of executing a CoreASM specification on the engine consists of three macro-steps, each of which includes a number of micro-steps as follows:

1. Initializing the engine (Figure 2)
  - (a) Initializing the kernel
  - (b) Loading the plug-ins library catalog
  - (c) Loading and activating the essential plug-ins (*core* plug-ins)
2. Loading a CoreASM specification (Figure 3)
  - (a) Parsing the specification header
  - (b) Loading further needed plug-ins as declared in the header
  - (c) Parsing the specification body
  - (d) Initializing the abstract storage
  - (e) Setting up the initial state
3. Execution of the specification
  - (a) Execute a single step
  - (b) If termination condition not met, repeat from 3a

At the end of the execution of each step, the resulting state is optionally made available by the abstract storage module for inspection through the Control API. The termination condition can be set through the user interface of the CoreASM engine, choosing between a number of possibilities (e.g., a given number of steps are executed; no updates are generated; the state does not change after a step; an interrupt signal is sent through the user interface).

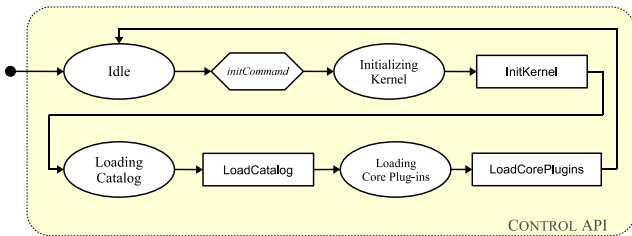
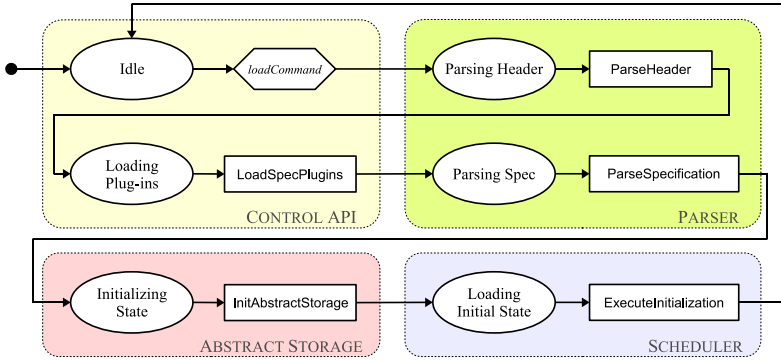


Fig. 2. Control State ASM of Initializing CoreASM Engine



**Fig. 3.** Control State ASM of Loading a CoreASM Specification

Figures 2-4 abstractly specify the execution process (described informally at the beginning of this section) in terms of a variant of control state ASMs [2, Sect. 2.2.6] [3].

*Engine Initialization.* The execution starts in the *Idle* state of the Control API module (Figure 2), waiting for a control command (e.g., *init*, *load*, or *step*) from the environment, which could be an interactive GUI or a debugger, to start the corresponding task. Receiving an *init* command (Figure 2) will change the engine state to *Initializing Kernel*. In this state, the engine initializes its kernel, loads the plug-in catalog, and finally loads the core plug-ins.

*Loading Specification.* Upon receiving a *load* command (Figure 3) the engine will load a new specification. It first parses the specification header to get the list of plug-ins that must be loaded. The required plug-ins are then loaded and the rest of the specification is parsed. To prepare the engine for its first step, the abstract storage is initialized and an initial state is created.

*Executing Specification.* The control state ASM of Figure 4 (presented in 4 parts for clarity) models the execution of a single CoreASM step. A *step* command triggers the start of a computation step (see Figure 4(a)) and changes the control state to *Starting Step* which transfers the control flow to the scheduler.

The scheduler retrieves the current set of agents from the abstract storage and initializes the selected set of agents to an empty set. If no agent is available to perform a computation, the step is considered complete; otherwise, the scheduler (through *SelectAgents*) chooses a set of agents to execute in the current step. The scheduler then chooses an agent from this set and changes the state to *Initializing SELF* (in the abstract storage, Figure 4(c)) which assigns the chosen agent as the value of *self*. After the execution of the agent, the computed updates are accumulated by the *AccumulateUpdates* rule and control is moved back to *Choosing Agent* until the programs of all the selected agents have been executed.

<sup>3</sup> We refer the reader to [4] for the actual ASM rules.

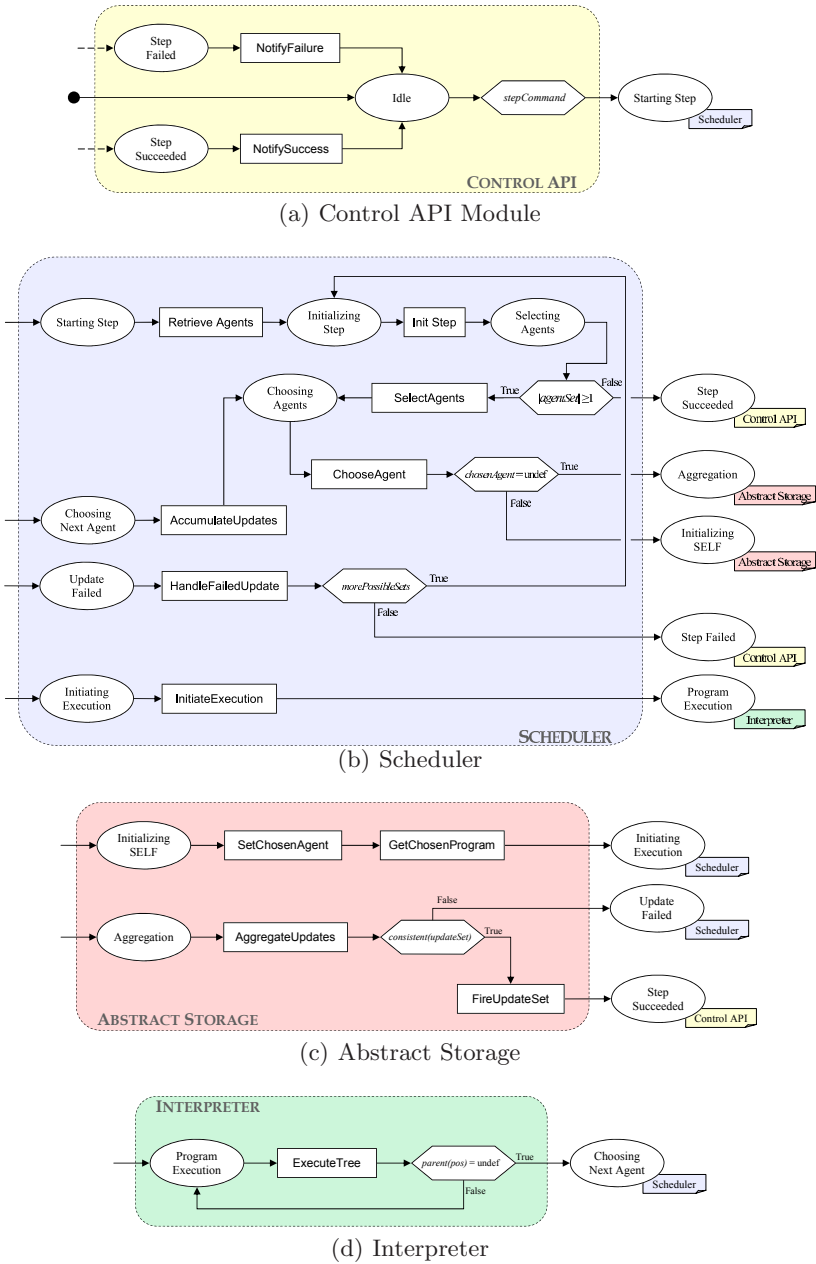


Fig. 4. Control State ASM of a STEP command

The program of the chosen agent is executed in the *Program Execution* state of the interpreter (Figure 4(d)). During the execution, computed update instructions are accumulated, and when all selected agents have performed their computation, control moves to the *Aggregation* state in the abstract storage. In this phase, the final update set is computed and if found consistent, the resulting updates are applied to the current state.

If an inconsistent set of updates is generated in a step, the scheduler selects a different subset of agents and the step is re-initiated. The process is iterated until a consistent set of updates is generated, in which case the computation proceeds to the *Step Succeeded* state of the Control API, or all possible combinations have been exhausted, in which case the *Step Failed* state is entered instead.

Depending on the outcome of the previous stage, either the *NotifySuccess* or the *NotifyFailure* rule in the Control API notifies the environment of the success or failure of the step, and Control API returns to the *Idle* state awaiting further commands from the environment (e.g., another *STEP* command to continue).

### 2.3 Microkernel Approach

The architecture of CoreASM is partitioned along two dimensions referred to as *modular decomposition* and *conservative refinement* respectively. In particular, our plug-ins progressively extend in a conservative way the capabilities of the language accepted by the CoreASM engine, in the same spirit in which successive layers of the Java [9] and C# [10] languages have been used to structure the language definition into manageable parts.<sup>4</sup>

Most of the functionality of the engine is implemented through plug-ins to a minimal kernel. This kernel contains only the bare essentials, that is, all that is needed to execute only the most basic ASM:

- the concepts of *functions* and *universes* are native to the kernel, as they are needed to define the state of an ASM;
- universes are represented through their characteristic functions, hence *booleans* are also included in the kernel;
- the special value *undef* is in the kernel;
- an ASM program is defined by a finite number of rules, hence the domain of *rules* is included in the kernel as well.

It should be noted that the kernel includes the above mentioned domains, but not all of the ‘expected’ backgrounds. While, for example, the domain of booleans (that is, *true* and *false*) is in the kernel, boolean algebra ( $\wedge$ ,  $\vee$ ,  $\neg$ , etc.) is not, and is instead provided through a background plug-in. In the same vein, while universes are represented in the kernel through set characteristic functions, the background of finite sets is implemented in a plug-in, which provides expression syntax for defining them, as well as an implicit representation for storing sets in the abstract state, and implementations of the various set theoretic operations (e.g.,  $\in$ ) that work on such an implicit representation.

<sup>4</sup> While CoreASM plug-ins can extend the engine through a conservative refinement, the CoreASM architecture does not restrict the plug-ins to such a refinement.

The kernel includes only two types of rules: assignment and **import**. This particular choice is motivated by the fact that without updates established by assignments there would be no way of specifying how the state should evolve, and that **import** has a special status due to its privileged access to the Reserve. All other rule forms (e.g., **if**, **choose**, **forall**), as well as sub-machine calls and macros, are implemented as plug-ins in a standard library, which is implicitly loaded with each CoreASM specification.

Finally, there is a single scheduling policy implemented in the kernel—the pseudo-random selection of an arbitrary subset of agents—which is sufficient for multi-agent ASMs where no assumptions are made on the scheduling policy.

### 3 Extensibility and Plug-Ins

The term *plug-in* and the concept behind it is widely used these days. According to the Merriam-Webster dictionary, the term refers to “*a small piece of software that supplements a larger program.*”, whereas the CNET glossary defines it as “*a type of program that tightly integrates with a larger application to add a special capability to it.*”

In fact, plug-in-based architectures can go beyond “supplementing” and “adding a special capability” to a main program. In contrast to the traditional plug-in architectures in which plug-ins extend the functionality of a host application, in new *Pure Plug-in Architectures*, like that of the Eclipse Platform<sup>5</sup>, almost all the functionalities of the application are provided by plug-ins [11][12].

Plug-in based architectures have three characteristics [12][13] that make them different from other types of modular architectures (e.g., pluggable components [14]):

1. Plug-ins are optional to the main application. In other words, the main application is independent of the plug-ins, allowing plug-ins to be dynamically added or updated without changing the main application.
2. Plug-ins are not known at compile-time of the main application; i.e., nothing about a specific plug-in is hard coded into the main application’s source code.
3. Plug-ins are not stand-alone programs; i.e., in order to function, they have to be deployed in the environment provided by the main application which they were designed for.

There are many advantages in adopting a plug-in based architecture. One advantage, which is also shared to some degree with other modular architectures, is that the complexity of large systems can be reduced by modularization of the system into plug-ins. In traditional modular architectures, all modules (components) have to be designed and developed to build the system. The main benefit of having a plug-in based architecture, specifically from a software development point of view, is that functionality can be developed and added at a later time — i.e., after the application is shipped or released — without changing or rebuilding the whole system. A well-designed architecture also allows third parties

<sup>5</sup> See [www.eclipse.org](http://www.eclipse.org)



to develop new plug-ins that further extend the functionality of the main application or customize it for different environments [12,13,15]. This feature is particularly important for modeling environments, such as CoreASM, that are built to be used and customized in various application domains.

There are two main requirements that motivates a plug-in based architecture for CoreASM. The first requirement is the extensibility of the CoreASM language and state. ASM has been used in various domains, some of which required the introduction of special rule forms and data structures into ASM. To follow the same spirit and to preserve this freedom of experimentation that comes with ASM, the CoreASM language has to be easily extensible by third parties so that it can naturally fit into different application domains.

The second requirement is to have an extensible CoreASM engine whose behavior can be modified by plug-ins and extensions. Such extensibility allows various modeling tools and environments to closely interact with the engine and also gives researchers the freedom to experiment with variations to the engine's functionality such as scheduling policy or consistency check of updates.

In this section we present the plug-in architecture of CoreASM and exemplify its capabilities by looking at two standard plug-ins: the Signature Plug-in and the IO Plug-in.

### 3.1 CoreASM Plug-In Framework

A CoreASM plug-in is a Java class implementing one or more of the interfaces defined by the CoreASM extensibility framework (see Table 1). The framework supports two extension mechanisms: plug-ins can either extend the functionality of specific components of the engine, by contributing additional data or behavior to those components, or they can extend the control state ASM of the engine itself, by interposing their own code in between state transitions.

In the rest of this section we look at various plug-in interfaces and explore the mechanisms through which they extend the CoreASM engine.

**Parser Extensions.** Plug-ins can implement the *Parser Plug-in* interface and/or the *Operator Provider* interface to extend the parser by respectively contributing additional grammar rules and/or new operator descriptions. For any parser plug-in *pp*, *pluginGrammar(pp)* holds the set of all the grammar rules contributed by *pp*. For any operator provider *op*, *pluginOperators(op)* holds the descriptions (syntax and semantics) of new operators contributed by *op*.

Before parsing a specification, the engine gathers all the grammar rules and operator descriptions provided by all parser plug-ins and operator providers. The parser component then combines these grammar rules and operator tokens with the kernel grammar and builds a new 'parser' to scan the specification. While building the abstract syntax tree, this parser labels the nodes that are created by plug-in-provided grammar rules with the plug-in identifier; these labels can later be used by the interpreter to evaluate such nodes.

Parser plug-ins and operator providers are probed by the *LoadSpecPlugins* rule before the engine starts parsing the specification (see Figure 3). This rule

**Table 1.** CoreASM Plug-in Interfaces

<b>Plug-in Interface</b>	<b>Extends</b>	<b>Description</b>
<i>Parser Plug-in</i>	Parser	provides additional grammar rules to the parser
<i>Interpreter Plug-in</i>	Interpreter	provides new semantics to the interpreter
<i>Operator Provider</i>	Parser, Interpreter	provides grammar rules for new operators along with their precedence levels and semantics
<i>Vocabulary Extender</i>	Abstract Storage	extends the state with additional functions, universes, and backgrounds
<i>Aggregator</i>	Abstract Storage	aggregates partial updates into basic updates
<i>Scheduler Plugin</i>	Scheduler	provides new scheduling policies for multi-agent ASMs
<i>Extension Point Plugin</i>	all components	extends the control state model of the engine

iterates over all the plug-ins required by the loaded specification and after ensuring dependency requirements, loads the plug-ins by calling the `LoadPlugin` rule presented below. The latter initializes the plug-in, then loads all the provided grammar rules and operator descriptions to be processed by the parser in the next control state.

Control API

---

```

LoadPlugin(p) ≡
  if p ∉ loadedPlugins then
    InitializePlugin(p) seq
      add p to loadedPlugins
      if isParserPlugin(p) then
        add pluginGrammar(p) to grammarRules
      if isOperatorProvider(p) then
        add pluginOperators(p) to operatorRules

```

---

**Interpreter Extensions.** Plug-ins can extend the interpreter component of the engine by implementing either the *Interpreter Plug-in* interface or the *Operator Provider* interface (or both). These plug-ins provide the semantics for rules and operations contributed as per Section 3.1. Traversing the abstract syntax tree, the `ExecuteTree` rule of the interpreter (see Figure 4(d)) uses these semantic rules to evaluate nodes that correspond to the extended grammar rules.

The semantics contributed by a plug-in *p* which implements the Interpreter Plug-in interface can be obtained through *pluginRule*(*p*). As already mentioned earlier, nodes of the parse tree corresponding to grammar rules provided by a plug-in are annotated with the plug-in identifier. If a node is found to refer to a

plug-in, the interpreter obtains the semantic rules provided by that plug-in and executes it; otherwise, the default kernel interpreter rules are used.

The `ExecuteTree` rule of the interpreter is presented below. In this rule, the current position in the abstract syntax tree is denoted by the nullary function  $pos$ , and assignment to  $pos$  is used to move evaluation to a different node. We refer the reader to [51] for more details on this process.

---

Interpreter

```

ExecuteTree  $\equiv$ 
  if  $\neg$ evaluated( $pos$ ) then
    if  $plugin(pos) \neq undef$  then
      let  $R = pluginRule(plugin(pos))$  in  $R$ 
    else
      KernelInterpreter
  else
    if  $parent(pos) \neq undef$  then  $pos := parent(pos)$ 

```

---

A similar approach is also used by the `KernelInterpreter` rule to obtain semantics of extended operators from Operator Providers. A detailed discussion on how the engine deals with operators and their extensions is provided in [16].

**Abstract Storage Extensions.** *Vocabulary Extender* plug-ins can extend the vocabulary of the CoreASM state by contributing new backgrounds, universes, and functions to the abstract storage. Such plug-ins in fact extend the initial state and signature of the simulated machine. In the abstract storage, the following functions bind the names of functions and universes in the CoreASM state to the mathematical objects that represent them. Backgrounds are considered as special universes and hence are handled by the same mapping.

$$stateUniverse : STATE \times NAME \rightarrow UNIVERSEELEMENT$$

$$stateFunction : STATE \times NAME \rightarrow FUNCTIONELEMENT$$

The value of these functions is initialized by the `InitAbstractStorage` rule (see Figure 3). After creating the default universe and functions (i.e., “Agents”, “program”, and “self”), this rule iterates over all vocabulary extender plug-ins and extends the CoreASM state with the vocabulary they provide:

---

Abstract Storage

```

InitAbstractStorage  $\equiv$ 
  InitializeState
  forall  $p \in specPlugins$  do
    if isVocabularyExtender( $p$ ) then
      forall  $(bkgName, bkg) \in pluginBackgrounds(p)$  do
        stateUniverse( $state, bkgName$ ) :=  $bkg$ 
      forall  $(uName, universe) \in pluginUniverses(p)$  do
        stateUniverse( $state, uName$ ) :=  $universe$ 
      forall  $(fName, f) \in pluginFunctions(p)$  do
        stateFunction( $state, fName$ ) :=  $f$ 

```

---

Plug-ins can also implement the *Aggregator* interface and provide aggregation rules to be applied on update instructions before they are submitted to the state. The aggregator plug-ins are called to aggregate update instructions by the `AggregateUpdate` rule in the *Aggregation* state of the engine (see Figure 4(c)). Aggregators are used, for example, to implement partial updates; for more detail on this issue, we refer the reader to [16].

**Scheduler Extensions.** *Policy plug-ins* — also called *Scheduler plug-ins* — extend the scheduler of the engine by providing new scheduling policies that affect the selection of agents in multi-agent ASMs. They provide an extension to the scheduler that is used to determine at each step the next set of agents to execute. In practice, a scheduler plug-in provides a concrete implementation of a `choose` in the `SelectAgents` step in Figure 4(b). It is worthwhile to note that only a single scheduling policy can be in force at any given time, whereas an arbitrary number of plug-ins of the remaining types can be all in use at the same time.

**Extension Point Plug-ins.** In addition to modular extensions of specific components, plug-ins can also extend the control state of the engine by registering themselves for *Extension Points*. Each mode transition in the execution engine is associated to an extension point. At any extension point, if there is any plug-in registered for that point, the code contributed by the plug-in for that transition is executed before the engine proceeds into the new mode. Such a mechanism enables arbitrary extensions to the engine’s life-cycle, which facilitates implementing various practically relevant features such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the behavior of the simulated machine (e.g., coverage analysis or profiling). A plug-in, for example, could monitor the updates that are generated by a step before they are actually applied to the current state of the simulated machine, possibly checking conditions on these updates and thus implementing a kind of watches (i.e., displaying updates to certain locations) or watch-points (i.e., suspending execution of the engine when certain updates are generated), which are useful for debugging purposes. As an additional example, a plug-in could provide syntax for declaring assertions and invariants. Assertions have to be checked when the corresponding node is evaluated, hence the plug-in would also implement the Interpreter extension to give semantics to assertions. In contrast, invariants have to be checked at each step (not when a particular rule is executed), for example immediately before applying updates: thus, the plug-in would hook on the `FireUpdateSet` extension point to check that the declared invariants really hold in each state.

As we mentioned earlier, we have used a variant of control state ASMs to present a high-level specification of the `CoreASM` engine. A control state ASM is an ASM whose rules are all of the form presented in Figure 5 (see [2, Section 2.2.6]). Such a control state ASM can be formulated in textual form by a parallel composition of FSM rules, where FSM is defined as:

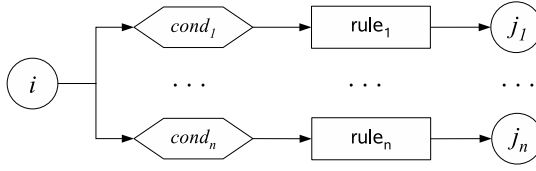


Fig. 5. Control State ASMs

$$\begin{aligned} \mathbf{FSM}(i, \mathbf{if\ cond\ then\ rule}, j) &\equiv \\ \mathbf{if\ } ctL\_state = i \mathbf{\ and\ } cond \mathbf{\ then} \\ &\quad rule \\ &\quad ctL\_state := j \end{aligned}$$

Thus, the control state ASM of Figure 5 can be formulated as a parallel composition of the following FSM rules:

$$\begin{aligned} &\mathbf{FSM}(i, \mathbf{if\ } cond_1 \mathbf{\ then\ } rule_1, j_1) \\ &\mathbf{FSM}(i, \mathbf{if\ } cond_2 \mathbf{\ then\ } rule_2, j_2) \\ &\dots \\ &\mathbf{FSM}(i, \mathbf{if\ } cond_n \mathbf{\ then\ } rule_n, j_n) \end{aligned}$$

To model the CoreASM engine, we introduce a variation of control state ASMs, called an *Extensible Control State ASM*, which is a control state ASM with an additional (and potentially dynamic) set of *extension point plug-ins* contributing supplementary rules that are executed before the machine switches to a new state (i.e., before *ctl\_state* gets a new value).

Rules of extensible control state ASMs, although pictured with the same control state diagrams as shown in Figure 5, are formulated in textual form by a set of *Extensible Finite State Machine* (EFSM) rules, where EFSM is defined as follows:

---

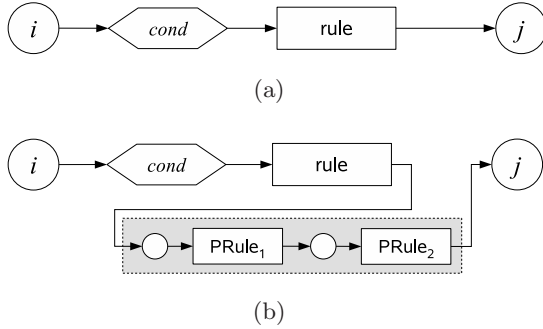
EFSM

$$\begin{aligned} \mathbf{EFSM}(i, \mathbf{if\ cond\ then\ rule}, j) &\equiv \\ \mathbf{if\ } ctL\_state = i \mathbf{\ and\ } cond \mathbf{\ then} \\ &\quad rule \mathbf{\ seq\ } \mathbf{Proceed}(i, j) \end{aligned}$$

$$\begin{aligned} \mathbf{Proceed}(i, j) &\equiv \\ \mathbf{forall\ } p \in extensionPointPlugins \mathbf{\ do} \\ &\quad marked(p) := isRegistered(p, i, j) \quad // \text{ mark the plug-ins registered for this point} \\ &\quad \mathbf{seq} \\ &\quad \mathbf{iterate} \\ &\quad \quad \mathbf{choose\ } p \in extensionPointPlugins \mathbf{\ with\ } marked(p) \mathbf{\ do} \\ &\quad \quad \quad marked(p) := false \\ &\quad \quad \quad \mathbf{let\ } R = extensionRule(p) \mathbf{\ in} \\ &\quad \quad \quad R(i, j) \\ &\quad \mathbf{seq} \\ &\quad ctL\_state := j \end{aligned}$$


---

An EFSM rule, instead of updating the control state of the machine in parallel with the execution of the transition rule, first executes the transition rule and



**Fig. 6.** (a) An Extensible Control State ASM and (b) its extended form

then non-deterministically iterates over all the extension point plug-ins and one by one executes their extension rules before switching the control state of the machine to a new state<sup>6</sup>

As an example, the extensible control state ASM of Figure 6(a) can be executed with a set of extension point plug-ins  $\{p_1, p_2\}$  contributing rules  $PRule_1$  and  $PRule_2$  which may extend the machine (during its execution) to the control state ASM of Figure 6(b).

The *Signature* and *IO* plug-ins from the standard CoreASM library, among others, implement the Extension Point interface to extend the control state ASM of the engine. We will look into these plug-ins in some detail in Sections 3.2 and 3.3.

**Plug-in Service Interface.** In many cases, there is a legitimate need for the environment where the CoreASM engine resides (e.g., the GUI of a simulator or of a debugger) to interact directly with some plug-ins. To support this interaction, the CoreASM extensibility framework introduces the concept of a *Plug-in Service Interface* through which plug-ins can expose part of their functionality to the environment of the engine.

The Plug-in Service Interface allows CoreASM plug-ins to define and provide their own interfaces to the environment. Applications utilizing the engine can access these interfaces through Control API and directly interact with such plug-ins. As an example, the IO Plug-in provides its own interface to expose the output of its **print** rules to the environment of the engine (see Section 3.3). A GUI for the engine, for example, can utilize this interface to obtain the printed output and display it in a console window.

As each plug-in exposes different functionalities, users of the Plug-in Service Interface have to know in advance what to expect from a specific plug-in, which is in keeping with the assumption that the environment will access specific services from a specific plug-in, as in the example above.

<sup>6</sup> At this point, we do not assume any order on the execution of plug-in rules and the non-deterministic **choose** rule clearly states that in the model.

### 3.2 Signature Plug-In

In principle, CoreASM functions are untyped alike ASM functions. While this is desirable in initial specification phases focusing on exploring the problem space, domain and range types of functions often add useful semantic information to a refined specification, for instance, to improve its understandability, to implement runtime type checking, and also to facilitate model checking. The Signature plug-in provides means to declare functions with their associated signatures, thereby adding type information to CoreASM. Moreover, it also allows to define new universes and enumerated backgrounds directly in a specification, rather than introducing them by a separate plug-in.

The Signature plug-in extends the parser, the interpreter and the abstract storage. Extending the grammar of the CoreASM language with its own syntactic patterns, the Signature plug-in creates new nodes in the AST. These nodes are not evaluated during the execution of the ASM, since they do not represent regular rules or expressions; rather they are interpreted before an ASM run, specifically during the transition from *Parsing Spec* to *Initializing State*. During the initialization of the abstract storage, the engine queries plug-ins for the vocabulary elements they provide. Hence, the interpretation of Signature declarations directly modifies the initial state (and vocabulary of the machine).

**Functions.** To declare functions, the Signature plug-in extends the CoreASM language with a syntactic pattern<sup>7</sup> with the following form, which can appear in the header of a specification:

	Function Declaration
$\langle \langle \mathbf{function} \ x_{name} : x_{d_1} * x_{d_2} * \dots * x_{d_n} \rightarrow x_r \rangle \rangle$	→
	$\text{CreateFunction}(x_{name}, \text{controlled}, \langle x_{d_1}, \dots, x_{d_n} \rangle, x_r, \text{undef})$

There are similar patterns for the definition of *static* and *monitored* functions.

One can also specify the initial value(s) of a function by including an initialization expression at the end of function declaration. The initialization expression

<sup>7</sup> The notation we use here has been introduced in [11], and due to space limitation we cannot fully present it again here. It will suffice to say that the semantics is given by ASM rules guarded by syntactical patterns; a variable *pos* indicates the subtree which is being evaluated, and is used to navigate the syntax tree. Patterns are delimited by  $\langle \rangle \rightarrow$  symbols; inside a pattern, variables named *x*, *e*, *v* indicate that the corresponding node or subtree is an identifier, an expression, a value. An empty box indicates an unevaluated node; a boxed letter indicates an unevaluated node which is expected to result in the corresponding element. Prefix superscripts name locations. In the ASM rules, each symbol is bound to the corresponding value in the pattern. Evaluation of ASM rules results in assigning a triple (location,updates,value) to the evaluated node; this operation is denoted as  $\llbracket pos \rrbracket := (l, u, v)$ .

may be a basic expression for nullary functions, or a map expression for  $n$ -ary functions. Before the function is added to the state, Signature plug-in uses the interpreter to evaluate the expression and sets the initial value(s) of the the function.

---

Function Declaration with Initialization

$\langle\langle$  **function controlled**  $x_{name} : x_{d_1} * \dots * x_{d_n} \rightarrow x_r$  *initially*  $v$   $\rangle\rangle \rightarrow$   
 $\text{CreateFunction}(x_{name}, \text{controlled}, \langle x_{d_1}, \dots, x_{d_n} \rangle, x_r, v)$

---

The interpretation of function declaration patterns is defined by the **CreateFunction** rule. This rule creates a new function element, assign the value of its function class, its signature, and if an initial value is provided, it also sets the initial value of the function. Finally, it adds the function to the list of functions provided by the Signature plug-in.

---

CreateFunction

**CreateFunction**( $name, functionClass, domain, range, initialValue$ )  $\equiv$   
**let**  $f = \text{new}(\text{FUNCTIONELEMENT})$  **in**  
 $fClass(f) := functionClass$   
**let**  $s = \text{new}(\text{SIGNATURE})$  **in**  
 $sigDomain(s) := domain$   
 $sigRange(s) := range$   
 $signature(f) := s$   
**if**  $initialValue \neq \text{undef}$  **then**  
 $\text{setFunctionValue}(f, domain, initialValue)$   
**add** ( $name, f$ ) **to**  $pluginFunctions(\text{SignaturePlugin})$

---

**Universes and Enumerations.** To declare universes, the Signature plug-in extends the CoreASM language with the following declaration forms:

---

Universe Declaration

$\langle\langle$  **universe**  $x_{name}$   $\rangle\rangle \rightarrow \text{CreateUniverse}(x_{name}, \{\})$   
 $\langle\langle$  **universe**  $x_{name} = \{x_{e_1}, \dots, x_{e_n}\}$   $\rangle\rangle \rightarrow \text{CreateUniverse}(x_{name}, \{x_{e_1}, \dots, x_{e_n}\})$

---

The second pattern above allows the specification writer to declare a universe along with a set of named initial member elements. Of course, a declared universe can still be extended using standard methods, namely by using the **extend** rule that imports a new element to a universe [2, Table 2.4] or by setting the value of the corresponding universe membership predicate to *true* for a given element.

The universe declaration patterns are interpreted by the **CreateUniverse** rule, which creates a new universe with the specified name. If initial member elements are specified, for each member element a static function that refers to the member is also created. The new universe is added to the list of universes provided by the Signature plug-in.<sup>8</sup>

---

<sup>8</sup> In CoreASM specification, *state* refers to the current state of the engine.



CreateUniverse

---

```

CreateUniverse(name, members) ≡
  let u = new(UNIVERSEELEMENT) in
    add (name, u) to pluginUniverses(SignaturePlugin)
    forall elementName ∈ members do
      let e = new(ELEMENT) in
        uMember(u, e) := true
        let f = new(FUNCTIONELEMENT) in
          add (elementName, f) to pluginFunctions(SignaturePlugin)
          fClass(f) := static
          FSetValue(f, ⟨⟩, e)

```

---

To support declaration of enumerated backgrounds, the Signature plug-in provides the following declaration form:

Enumeration Declaration

---

```

⟨ enum xname = {xe1, ..., xen} ⟩ → CreateEnumeration(xname, {xe1, ..., xen})

```

---

The CreateEnumeration rule is similar in spirit to CreateUniverse, as enumerable backgrounds are analogous to static universes.

**Possible Further Extensions.** It is interesting to note that, since the Signature plug-in internally manages the functions and universes that are defined by a specification writer, it can provide this information to external components, such as a GUI, through its Plug-in Service Interface. In a GUI, a user may find it useful to be able to visually differentiate between functions declared by the user and those defined by the kernel or by other plug-ins.

As noted earlier, the Signature plug-in could also be extended to implement runtime type checking of update sets. For each update to a user declared function, the Signature plug-in would check to see that the update value matches the specified range type of that function. This check would occur during the engine's transition from the *Aggregation* state to the *Step Succeeded* state.

### 3.3 IO Plug-In

In an open system view, the system operates in a given environment. The environment affects system runs through actions or events and the system can as well affect the environment by its output. In abstract state machines, the interaction between the system (the machine) and the environment is captured through *monitored* (also called *in*), *shared*, and *out* functions. Monitored functions are controlled by the environment and they are read-only for the machine. They are channels through which the machine observes the environment. In a given state, the values of all monitored functions are determined [2]. Out functions are updated only by the machine and they are read-only for the environment. Shared functions are both controlled and read by the machine and the environment.

The IO Plug-in utilizes this machine-environment interaction mechanism of ASM and provides two simple channels of communication between a CoreASM machine and its environment: a **print** rule that outputs values to the environment, and an *input* function to get values from the environment. In both cases, textual representations of values are used.

**Input and Output.** To provide an output channel for CoreASM specifications, the IO Plugin extends the state of the simulated machine by introducing an *output* function (*output*: STRINGELEMENT) which in any given step holds the output of the previous step. Output values are assigned to *output* by **print** rules. Every **print** rule generates an update instruction to append a given value to *output*.

IO Plugin

---


$$\langle \mathbf{print}^\alpha \square e \rangle \rightarrow pos := \alpha$$

$$\langle \mathbf{print}^\alpha v \rangle \rightarrow \mathbf{let} \ l = (\text{"output"}, \langle \rangle) \ \mathbf{in} \\ \llbracket pos \rrbracket := (\mathit{undef}, \{\langle l, v, \mathit{printAction} \rangle\}, \mathit{undef})$$


---

Conversely, to receive input from the environment the IO Plugin introduces a monitored function *input* with the following signature:

$$\mathit{input} : \text{STRINGELEMENT} \rightarrow \text{STRINGELEMENT}$$

For any given value as its argument, this *input* function queries an input value from the environment (presenting the argument as a prompt or key to the input value); this allows the machine to observe more than one input value by mapping different input values to different arguments.

**Extending the Engine.** The IO Plug-in implements Parser and Interpreter extensions to add the ‘**print value**’ pattern to the language and to provide semantics for it, respectively. The plug-in also extends the CoreASM state with the new *input* and *output* functions; this is obtained by providing a Vocabulary extension.

All the *printAction* instructions produced by the **print** rule need to be aggregated into one single update to the *output* function. To achieve this, the IO Plug-in provides an Aggregator extension so that it will be called by the engine during the aggregation phase at the end of each step (see Figure 4(c)). As an aggregator, the plug-in combines all the “printed” values into one single value and produces a single update to the *output* function.

At any simulated state, the *output* function holds the output produced in the last computation step. Thus, if the environment does not capture the output of the machine in a synchronous fashion, it loses it. In order to avoid loss of output data, the IO Plugin keeps the history of output values in a data stream and makes this history available to the environment. The plug-in maintains the output history by extending the lifecycle of the engine using Extension Points: namely, it hooks on the extension point prior to *Step Succeeded* (see Figure 4(c)).

— which is when all the updates are successfully applied to the state — and when called at that point, appends the current value of *output* to the output history.

We previously mentioned that the *input* function queries the environment for input values. This is done through a Plug-in Service Interface. The IO Plug-in implements its own version of the plug-in service interface and provides an extension point to the environment through which the environment can, in its turn, provide input to the IO Plug-in whenever the *input* function is being evaluated by the engine. As an example, Carma<sup>9</sup> utilizes this plug-in service interface to open an input dialog window every time the engine evaluates the *input* function (for the first time with the given argument in the current step). The input provided by the user is then assigned as the value of the *input* function for the current state of CoreASM.

## 4 Related Work

The idea of using plug-ins to develop extensible software has been adopted by many applications such as Mozilla Firefox, Eclipse, Adobe Photoshop, Gimp, and many others.<sup>10</sup> Although the idea has been around for almost two decades, there are still different interpretations of the concept. In 2002, following the style of [17], Mayer et al. [12] presented the plug-in concept as a design pattern. The pattern explains how one can design an application that can be extended at runtime by dynamically loaded plug-ins. It consists of a plug-in loader and zero or more plug-in interfaces and implementations. The article also provides a sample implementation of the pattern in Java.

Today, Eclipse is one of the most popular extensible development environments. Mostly known as a powerful Java integrated development environment, Eclipse is in fact a universal plug-in architecture for creating almost anything [11]. The architecture of Eclipse is an example of pure plug-in architectures. Its runtime engine is implemented as a number of core plug-ins, and except for a tiny bootstrap loader, everything else in Eclipse is provided by plug-ins. Eclipse plug-ins not only can extend the functionality of the kernel, they can also extend other plug-ins. However, despite the flexibility of the Eclipse framework, the complexity of its API makes it somewhat difficult for third parties to write even the simplest extension plug-ins. Nevertheless, many modeling and development frameworks, such as the design-driven environment Together<sup>11</sup> from Borland, have been ported to Eclipse due to the richness of the framework.

In its latest incarnation, the Eclipse extensibility framework is based on OSGi<sup>12</sup>, a dynamic modules systems for Java which, in addition to defining

<sup>9</sup> Carma is a command-line user interface for the CoreASM engine. Please visit [www.coreasm.org](http://www.coreasm.org) for more information.

<sup>10</sup> See [www.firefox.com](http://www.firefox.com), [www.eclipse.org](http://www.eclipse.org), [www.adobe.com](http://www.adobe.com), and [www.gimp.org](http://www.gimp.org) respectively.

<sup>11</sup> See [www.borland.com/together](http://www.borland.com/together)

<sup>12</sup> See [www.osgi.org](http://www.osgi.org)

a plug-in based architecture, provides specifications and services for run-time reconfiguration of a compliant application. Plug-ins can be added to or removed from an application without any need to stop it, and a security layer helps ensure the necessary trust between the various independent components. CoreASM does not manage security at all, assuming that the plug-ins in its library are trusted, but has dynamic capabilities as well: in fact, the set of plug-ins active in a certain run of a specification is entirely described by the specifications itself (via `use` directives), and plug-ins are loaded and unloaded at run time as necessary.

Language extensibility is not a new concept [18]. Various programming languages such as Smalltalk<sup>13</sup>, Common Lisp<sup>14</sup>, Python<sup>15</sup>, Fortress<sup>16</sup>, XLR<sup>17</sup>, and Seed7<sup>18</sup> all support some form of extensibility from definition of new macros to introduction of new syntactical structures. However, what we are offering in CoreASM is the possibility of extending and modifying the syntax and semantics of the language, keeping only the bare essential parts of the ASM language as static. To the best of our knowledge, among all these languages only the Seed7 programming language supports the introduction of new syntax and their semantics into the language. In terms of language extensibility, Seed7 goes beyond CoreASM as it allows new language constructs to be defined using the Seed7 language itself.

Among the various tools for running ASM models [1], Asmeta [19], XASM (eXtensible ASM) [20], and AsmL (ASM Language) [21] provide some form of extensibility. Asmeta focuses on defining a metamodel for ASM based on the Model-Driven Engineering (MDE) [22] guidelines. Since it is a metamodel-based framework, various tools and applications can be developed utilizing the already implemented features. AsmetaL (Asmeta Language) is not an extensible language but it can utilize external Java functions to model static ASM functions.

The XASM language provides an interface to C allowing both C-functions to be used in XASM programs as well as Xasm modules to be called from within C-programs. The extensibility is based on the concept of *components* and modularity. There is no mechanism to extend the XASM language or the execution model of its specifications.

AsmL, developed by the Foundations of Software Engineering group at Microsoft Research, is a strongly typed language based on the concept of ASMs. Built on the Microsoft .NET framework [23], AsmL incorporates numerous object-oriented features and constructs of Microsoft .NET for rapid prototyping of component-oriented software. Although a limited form of extensibility is provided through interaction with external .NET classes, neither the language nor the underlying execution model are extensible.

---

<sup>13</sup> [www.smalltalk.org](http://www.smalltalk.org)

<sup>14</sup> [www.common-lisp.net](http://www.common-lisp.net)

<sup>15</sup> [www.python.org](http://www.python.org)

<sup>16</sup> [projectfortress.sun.com](http://projectfortress.sun.com)

<sup>17</sup> [xlr.sourceforge.net](http://xlr.sourceforge.net)

<sup>18</sup> [seed7.sourceforge.net](http://seed7.sourceforge.net)

The design of CoreASM and its extensibility framework is exceptional among the existing executable ASM languages and modeling environments. Extensibility of the CoreASM language and its engine facilitates minimizing the need for *encoding* in mapping the problem space to a formal model and also supporting data operations at the natural level of abstraction of application domains. In addition, CoreASM plug-ins like JASMine [24] can enrich the language by providing access to external Java objects and classes from ASM specifications without polluting the basic ASM computation model.

As more and more plug-ins will be developed for the CoreASM engine, its plug-in architecture needs to be improved to better handle inconsistencies and overlapping features and also to support more sophisticated plug-in interactions. In future work, we will look into utilizing the concepts and ideas of Feature-Oriented Software Development (FOSD) [25,26], such that every plug-in would provide a list of features. We believe that this would facilitate the integration of complementing features and the detection of inconsistent or overlapping ones.

## 5 Conclusion

We have presented the extensibility architecture of the CoreASM engine through which both the interpreter and the execution model of the CoreASM language can be extended by third-party plug-ins.

The extensibility architecture provides a number of specific extension points for parsing and interpreting CoreASM specifications, and providing syntax and semantics for new operators and data types with associated static functions (i.e., parts of the ASM implicit background with its initial signature). These extensions allow external plug-ins to change the language by adding new rule forms and operations, thus realizing *domain-specific* ASM dialects. Such languages have often been used with indubitable merits in the literature: many published specifications of large systems have introduced background elements or non-standard rule forms that were well suited to express the intended behavior at the level of abstraction appropriate in the given domain. By allowing customization of the CoreASM language along the same lines, we provide the benefits of executable specifications without renouncing the expressivity of a domain-specific language, and thus avoid the introduction of a further encoding level which would stand between the conceptual specification and its executable version.

The extensibility architecture also supports the customization of scheduling policies for multi-agent or distributed ASM. While most common policies are provided in a standard library, in certain applications the exact scheduling of agents is important, and it is much easier to state the expected properties of the scheduling policy in the given context (and provide a concrete implementation of a scheduler satisfying those properties) than to re-encode an agent scheduler as part of the ASM itself.

Finally, the entire life cycle of a CoreASM execution has been specified as a control-state ASM, and the extensibility architecture provides an homogeneous extension mechanism for any arbitrary step in the engine. We have defined a

general EFSM() macro (for Extensible Finite State Machine) which, substituted for the classical FSM() macro from [2], provides any control-state ASM with the kind of extensibility we have implemented in the CoreASM engine. A number of capabilities are provided by such extensions, of which we have provided examples in the paper.

In summary, the CoreASM extensibility framework provides utmost flexibility for extending the language definition and the execution engine in order to tailor it to the particular needs of virtually any conceivable domain-specific application context—very much the way Abstract State Machines were meant to be used.

**Acknowledgments.** We would like to thank Mashaal Memon for his contribution to the CoreASM project and for many constructive discussions regarding the plug-in architecture. Our sincere appreciation to Egon Börger for persistent encouragement and valuable feedback on the CoreASM project.

## References

1. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae*, 71–103 (2007)
2. Börger, E., Stärk, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003)
3. Fowler, M.: *The New Methodology* (April 2003), <http://martinfowler.com/articles/newMethodology.html>
4. Farahbod, R., Gervasi, V., Glässer, U.: Design and Specification of the CoreASM Execution Engine, Part 1: the Kernel. Technical Report SFU-CMPT-TR-2006-09, Simon Fraser University (May 2006), <http://www.coreasm.org>
5. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: Proc. of the 12th Int'l Workshop on Abstract State Machines (2005)
6. Farahbod, R., Gervasi, V., Glässer, U., Memon, M.: Design exploration and experimental validation of abstract requirements. In: Proceedings of the 12th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2006), Luxembourg, Grand-Duchy of Luxembourg, Essener Informatik Beiträge (June 2006)
7. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. *ACM Transactions on Computation Logic* 4(4), 578–651 (2003)
8. Blass, A., Gurevich, Y.: Background, Reserve, and Gandy Machines. In: Clote, P.G., Schwichtenberg, H. (eds.) *CSL 2000*. LNCS, vol. 1862, pp. 1–17. Springer, Heidelberg (2000)
9. Stärk, R., Schmid, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, Heidelberg (2001)
10. Börger, E., Fruja, N.G., Gervasi, V., Stärk, R.F.: A High-level Modular Definition of the Semantics of C#. *Theoretical Computer Science* 336(2/3), 235–284 (2005)
11. Birsan, D.: On Plug-ins and Extensible Architectures. *ACM Queue* 3(2), 40–46 (2005)
12. Mayer, J., Melzer, I., Schweiggert, F.: Lightweight plug-in-based application development. In: Aksit, M., Mezini, M., Unland, R. (eds.) *NODE 2002*. LNCS, vol. 2591, pp. 87–102. Springer, Heidelberg (2003)

13. Marquardt, K.: Patterns for plug-ins. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, EuroPLoP 1999 (1999)
14. Völter, M.: Pluggable Components — A Pattern for Interactive System Configuration. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999) (1999)
15. Chatley, R., Eisenbach, S., Magee, J.: Modelling a Framework for Plugins. In: Proceedings of Specification and verification of component-based systems (September 2003)
16. Memon, M.A.: Specification language design concepts: Aggregation and extensibility in coreasm. Master's thesis, Simon Fraser University, Burnaby, Canada (April 2006)
17. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
18. Standish, T.A.: Extensibility in programming language design. SIGPLAN Not. 10(7), 18–21 (1975)
19. Formal Methods laboratory of University of Milan: Asmeta (2006), <http://asmeta.sourceforge.net/> (Last visited June 2008)
20. Anlauff, M.: XASM – An Extensible, Component-Based Abstract State Machines Language. In: Gurevich, Y., Kutter, P., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 69–90. Springer, Heidelberg (2000)
21. Microsoft FSE Group: The Abstract State Machine Language (2003), <http://research.microsoft.com/fse/asml/> (Last visited June 2008)
22. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer 39 (February 2006)
23. Microsoft Corp.: Microsoft .NET Framework (Last visited December 2006), <http://www.microsoft.com/net>
24. Farahbod, R., Gervasi, V.: JASMine: Accessing Java Code from CoreASM. In: Abrial, J.-R., Glässer, U. (eds.) Börger Festschrift. LNCS, vol. 5115, pp. 170–186. Springer, Heidelberg (2009)
25. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling stepwise refinement (2003)
26. Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An algebra for feature-oriented software development. Number MIP-0706. University of Passau (2007)

# JASMine: Accessing Java Code from CoreASM

Vincenzo Gervasi<sup>1</sup> and Roozbeh Farahbod<sup>2</sup>

<sup>1</sup> Dipartimento di Informatica  
Università di Pisa  
Pisa, Italy

<sup>2</sup> School of Computing Science  
Simon Fraser University  
Burnaby, B.C. Canada

**Abstract.** In this paper we introduce JASMine, a CoreASM plug-in providing means to access Java objects and classes from inside an ASM specification. We discuss why this access is desirable, and provide a formal specification of the new rule forms as well as some notes on the actual implementation. JASMine ensures that the ability to access the “Java world” does not pollute the mathematical purity of the basic ASM computation model; differences between the JASMine approach and the other major research effort in the same direction, namely the way AsmL interacts with the .NET framework, are also discussed.

## 1 Introduction

The CoreASM environment [5,6] consists of an open-source, platform-independent *execution engine* for the CoreASM *language*, together with a suite of tools for editing, testing and proving properties of CoreASM specifications.

The engine comes with a sophisticated and well defined interface, thereby enabling development and integration of complementary tools, e.g., for symbolic model checking [4] and automated test generation [10]. The design of CoreASM is novel and the underlying principles are unprecedented among the existing executable ASM languages, including the most advanced ones: AsmL [14], the ASM Workbench [3], XASM [1], and AsmGofer [15]. In particular, CoreASM stresses *fidelity to the mathematical definition of ASM* and *extensibility of the language* above other concerns. The syntax and semantics of the CoreASM language closely follow those given in [2], thus preserving the mathematical purity of the language. Yet, the engine supports extensibility mechanisms [9] (through plug-ins) which make it possible to extend the language by adding new constructs in a modular fashion, thus enabling creation of domain-specific dialects which are more suited to writing specifications for a given domain.

CoreASM provides its own set of plug-ins implementing most common mathematical objects and structures, e.g. *numbers*, *sets*, *sequences*, *maps*. While these sorts are usually sufficient for modeling most algorithms and systems, complex specifications may need more advanced features, not necessarily data-oriented. For example, an executable specification for a new peer-to-peer protocol may need access to *network sockets* and *files*; a specification that is used as an executable stub for a software module that still has to be implemented or for a



missing piece of hardware may need to put up an on-screen *window* showing its current state; a complex numerical algorithm which is already specified by some standard may be moved out of a specification and a *concrete implementation* written in a standard programming language may be used in its place.

There is thus a clear need to allow *interaction* between CoreASM specifications and concrete code, including operating systems functions, external libraries, and custom code. Among the various tools for running ASM models [8], AsmL (ASM Language) [14], XASM (eXtensible ASM) [11], and AsmGofer [15] provide some support for interaction with external programming languages. AsmL, built on the Microsoft .NET framework [13], incorporates numerous object-oriented features and constructs of Microsoft .NET and supports interaction with external .NET classes. The XASM language allows external C-functions to be used in XASM specifications. However, the arguments and return values of C-functions can only be of a specific C-type that represents elements of the super-universe in XASM. Newer versions of XASM support interaction with Java classes but the support is only limited to invoking Java object constructors. AsmGofer [15], an ASM interpreter embedded in the functional programming language “Gofer”, supports the use of functional programming in the definition of types and functions.

This paper sketches a solution based on integrating Java with CoreASM through a CoreASM plug-in named JASMine. The paper is structured as follows. The next section presents the most important design choices with their rationale, and provides the formal semantics for the new ASM rule forms introduced by JASMine. Section 3 shows in some detail how the JASMine plug-in is implemented. Section 4 presents a simple example of how JASMine can be used in a CoreASM specification to interact with Java objects; this is followed by some consideration on pragmatic aspects and directions for future work. Finally, Section 6 presents some conclusions and examples of relevant applications of JASMine in real-life scenarios.

## 2 JASMine: Design and Semantics

The Java Class Library provides an extremely rich (and continuously growing) set of APIs and efficient implementations for almost any computing task. Moreover, Java offers platform-independence, support on a wide variety of architectures, and many modern language features that make it an attractive target for the integration of ASM specifications with concrete code.

However, there is a marked risk that by intertwining too strictly the “ASM world” of elements, functions and predicates and the “object world” of an object-oriented language, the very nature of the ASM paradigm is changed in fundamental ways. This is, for example, what happened in AsmL [14], where rules and methods, elements and objects, sets and the Set object of the .NET framework become confused.

In contrast, we do not want interaction with Java to *pollute* the CoreASM word. In particular,

- we want to maintain typelessness of the language: it must be possible to treat Java objects as regular ASM values, and to pass untyped ASM elements as arguments to Java methods (with type checking performed at run time only);

- we want to maintain the parallel model of execution of ASM: the notion of *step* must be preserved, as well as the assumption that the ASM state and environment is observed in a stable snapshot, and updates are applied in parallel and only when no conflict arise;
- we want to avoid the introduction of extraneous fundamental concepts: the notions of *state*, *update* and *step* should suffice to describe the computation.

The fundamental choice of preserving — unadulterated — the ASM computation model sets strong constraints on how JASMine works, which will be described later in more detail.

Four basic capabilities are needed for a minimal reasonable level of interaction, namely:

1. Instantiating new objects, invoking their constructors, storing a reference to the new object in the ASM state.
2. Accessing (reading and writing) public fields of objects, including static fields of classes.
3. Invoking public methods of objects and static methods of classes, passing the needed arguments, and storing the result in the ASM state.
4. Converting between certain ASM types and the corresponding Java types and back, as needed to support expression evaluation and updates.

The mechanisms we propose to provide these capabilities constitute a *conservative extension* of CoreASM, in the sense that the semantics of the non-JASMine parts of a specification are not altered by the extension<sup>1</sup>.

Notice that the integration that JASMine provides between ASM and Java is far less complete than the one existing between, for example, AsmL and .NET: in particular, it is not currently possible to define new Java classes or interfaces through ASM specifications, nor is it possible to use Java inheritance in CoreASM specifications. Interfaces and abstract classes cannot be accessed at all.

We do not see these limitations as particularly relevant in practice. In fact, the design goal of JASMine is to allow *interaction* between ASM and Java, rather than full *integration*, and we believe the JASMine plug-in to serve well in this capacity.

The following subsections describe in turn the constructs implementing the four capabilities mentioned above. We will provide here a formal semantics using the notation introduced in [7]. For brevity's sake, we do not fully present the notation again here. It will suffice to say that the semantics is given by ASM rules guarded by syntactical patterns; a variable *pos* indicates the subtree which is being evaluated, and is used to navigate the syntax tree. Patterns are delimited by  $(\ ) \rightarrow$  symbols; inside a pattern, variables named *x*, *l*, *e*, *v* indicate that the corresponding subtree shall evaluate, respectively, to an identifier, a location (i.e., an assignable l-value), an expression, a value (i.e., an evaluated expression). An empty box indicates an unevaluated node; a boxed letter from

<sup>1</sup> In other terms, a specification which does not interact with Java, and thus does not use the JASMine constructs, has the same semantics whether it includes the JASMine plug-in or not.

the set above indicates an unevaluated node which is expected to result in the corresponding element. Prefixed superscripts (usually in Greek letters) are used to name positions in the syntax tree. In the ASM rules, each of these symbols is bound to the corresponding value in the pattern. Evaluation results in assigning a triple (location,updates,value) to the evaluated node; this operation is denoted as  $\llbracket pos \rrbracket := (l, u, v)$ . These indications are only meant to support the intuition; the interested reader should see Section 3.1 of [7] for a precise definition.

### 2.1 Creation of Java Objects

Java objects in JASMine are seen as part of the *environment*, not of the *state*. This is a fundamental design choice, which differs from what others have done (e.g., AsmL), and helps in cleanly separating the structures-based state of ASM, which only changes between steps and through non-conflicting updates, from the independently evolving state of Java, which can change at any time and also due to external events (e.g., a timer or GUI actions).

JASMine introduces a new background (hence, a new kind of element in the ASM state) called `JObject` which holds a *reference* to the real Java object. Only this immutable reference enters the ASM state as a value, and only through a special update command, hence the basic ASM computation cycle is preserved. As a consequence, creation of a new object is not considered an expression (as is the `new` operator in Java) but rather a rule, since it results in an update.

In formal terms, using the notation described above, creation of a new Java object is accomplished as follows:

---

	CreationRules
$\langle\langle \text{import native } \alpha \square \text{ into } \beta \square \rangle\rangle \rightarrow pos := \beta$	
$\langle\langle \text{import native } \alpha x \text{ into } \beta l \rangle\rangle \rightarrow \begin{array}{l} \text{if } isJavaClassName(x) \text{ then} \\ \quad \text{if } hasEmptyConstructor(x) \text{ then} \\ \quad \quad \text{EvaluateImport}(l, x, \langle \rangle) \\ \quad \text{else} \\ \quad \quad \text{Error('Constructor not found.')} \\ \text{else} \\ \quad \text{Error('Java class not found.')} \end{array}$	
$\langle\langle \text{import native } \alpha x(\lambda_1 \square_1, \dots, \lambda_n \square_n) \text{ into } \beta \square \rangle\rangle \rightarrow pos := \beta$	
$\langle\langle \text{import native } \alpha x(\lambda_1 \square_1, \dots, \lambda_n \square_n) \text{ into } \beta l \rangle\rangle \rightarrow \begin{array}{l} \text{if } isJavaClassName(x) \text{ then} \\ \quad \text{choose } i \in [1..n] \text{ with } \neg evaluated(\lambda_i) \\ \quad \quad pos := \lambda_i \\ \quad \text{ifnone} \\ \quad \quad \text{if } hasConstructor(x, \langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle) \\ \quad \quad \quad \text{EvaluateImport}(l, x, \langle \lambda_1, \dots, \lambda_n \rangle) \\ \quad \quad \text{else} \\ \quad \quad \quad \text{Error('Constructor not found.')} \\ \text{else} \\ \quad \text{Error('Java class not found.')} \end{array}$	

---

Here, we use the  $jValue$  function to abstract from the task of potentially converting CoreASM elements to Java objects (see Section 2.4). The actual evaluation of the **import native** statement is defined by the following macro, which takes as parameters a location where to store the reference to the new Java object (as a JObject value), an identifier representing the name of the class, and a sequence of positions of values, which will be the actual parameters for the constructor call:

---

EvaluateImport

**EvaluateImport**( $l, x, \langle \lambda_i, \dots, \lambda_n \rangle$ )  $\equiv$   
 let  $u = \text{DefUpd}(\text{CREATE}, (l, x, \langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle))$  in  
 let  $jtl = (\text{"jasmChannel"}, \langle \rangle)$  in  
 $\llbracket pos \rrbracket := (\text{undef}, \llbracket \langle jtl, u, jasmAction \rangle \rrbracket, \text{undef})$

---

Notice in the specification above how the execution of the rule does not really instantiate the new object (whose constructor could have side effects, and thus alter the Java state), but instead accumulates a special update instruction (a *deferred update*) akin to the update instructions used for aggregation and partial updates [12]. Actual instantiation will be performed at update application time, as will be shown later on. The designated location ("jasmChannel",  $\langle \rangle$ ) accumulates all the JASMine-related update instructions that are performed during a step, whereas the DefUpd macro produces an encoding of its parameters, suitable for later execution of the relevant update.

While the subject will be discussed more fully in the following, it is worthwhile to remark here that this strategy ensures that any action that can perturb the environment (e.g., instantiation of a new Java object) will only be taken if the step turns out to be effective, i.e. if no conflicting updates are generated in that step.

## 2.2 Access to Fields of Java Objects

Reading a field in a Java object does not have side effects and thus can be treated as a pure expression as far as the ASM computation cycle is concerned<sup>2</sup>. In particular, the value in the field can be computed immediately at expression evaluation time. In contrast, writing into a field has observable side effects, and thus cannot be performed *during* a step, but only *between* steps; the corresponding value is then stored in the field at update application time through another deferred update. The following rules detail the semantics used for field access in JASMine.

---

<sup>2</sup> In a multi-threaded context, field values can change at any moment, even without any write action by the ASM specification. To guarantee the stability of the environment, values read from Java fields are cached by JASMine when first read, and the same value is used if the same field read expression on the same Java object is evaluated multiple times in the same step.

FieldReadExpression

---

```

( $\alpha \square \rightarrow^\beta x$ )  $\rightarrow pos := \alpha$ 
( $\alpha v \rightarrow^\beta x$ )  $\rightarrow$  if isObject(v)
  if hasField(jObj(v), x)
    if ImplicitConversionMode then
       $\llbracket pos \rrbracket := (undef, undef, asmValue(GetField(jObj(v), x)))$ 
    else
       $\llbracket pos \rrbracket := (undef, undef, newJObject(GetField(jObj(v), x)))$ 
  else
    Error('No such field.')
else
  Error('Not a Java object.')
```

---

As can be observed, field access expressions are evaluated by first evaluating the reference to the `JObject`, and then (after checking that the given value is actually a `JObject` and that the corresponding class has an accessible field with the given name) the value in the field of the Java object is retrieved, possibly converted to its ASM counterpart based on the configuration of the plug-in (see Section 2.4), and finally used as the value of the whole expression. Access to static class fields are handled similarly, and we skip here the corresponding rules for brevity.<sup>3</sup> Assignments are treated through deferred updates:

FieldWriteRule

---

```

(store  $\alpha \square$  into  $\beta \square \rightarrow^\gamma x$ )  $\rightarrow$ 
  choose  $\lambda \in \{\alpha, \beta\}$  with  $\neg evaluated(\lambda)$ 
     $pos := \lambda$ 
  ifnone
    if isObject(value( $\beta$ )) then
      if hasField(jObj(value( $\beta$ )), x) then
        let u = DefUpd(STORE, (value( $\beta$ ), x, jValue(value( $\alpha$ ))) in
          let jtl = ("jasmChannel",  $\langle \rangle$ ) in
             $\llbracket pos \rrbracket := (undef, \{\langle jtl, u, jasmAction \rangle\}, undef)$ 
        else
          Error('No such field.')
      else
        Error('Not a Java object.')
```

---

Notice how write access to fields is treated as a partial update to the internal structure of the `JObject` element. Before the engine applies the updates to the state, the JASMine plug-in as the corresponding aggregator will have to check that no conflicting assignments to the same field of a given `JObject` element are performed, and moreover that the `JObject` as a whole is not updated to

---

<sup>3</sup> Reading a static field of a class that has a static block and is not initialized can potentially have side effects. Currently, we do not handle this special case and treat static fields and object fields the same with regard to read access.

a different value in the same step<sup>4</sup>. Once more, write access to static fields of classes is very similar and we do not detail it here.

### 2.3 Invoking Methods of Java Objects

As remarked above, invocation of methods in Java objects can have side effects which can change both the internal state of the object and of other objects as well (i.e., by calling other methods or accessing public fields). For this reason, method invocation is handled through a deferred update, as described below. Two forms of method invocation exists: one for *void* methods, which have no return value, and one for methods returning a value. The simplest version for void methods invocation is specified as follows:

---

VoidMethodInvocationRule

```

(invoke  $\alpha \llbracket c \rrbracket \rightarrow^\beta x(\lambda_1 \llbracket c_1 \rrbracket, \dots, \lambda_n \llbracket c_n \rrbracket)$ )  $\rightarrow$ 
  choose  $\lambda \in \{\alpha, \lambda_1, \dots, \lambda_n\}$  with  $\neg \text{evaluated}(\lambda)$ 
     $pos := \lambda$ 
  ifnone
    if  $isJObject(value(\alpha))$ 
      if  $hasMethod(jObj(value(\alpha)), x, \langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle)$ 
        let  $u = DefUpd(INVOKE,$ 
           $(undef, value(\alpha), x, \langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle))$  in
          let  $jtl = (\text{"jasmChannel"}, \langle \rangle)$  in
             $\llbracket pos \rrbracket := (undef, \langle \langle jtl, u, jasmAction \rangle \rangle, undef)$ 
        else
           $Error(\text{'No such method.'})$ 
      else
         $Error(\text{'Not a Java object.'})$ 

```

---

The version for non-*void* methods is only slightly more complex. We provide a special update instruction (in the vein of **add ... to ...**) so that the actual method call is only performed if the update set is guaranteed to be consistent (see section 2.5 for detailed conditions).

This solution may be inconvenient at times. For example, it is not possible to assign directly the result of a method invocation to a field of the same or of a different object, as two separate **invoke** and **store** instructions are needed, and in two different steps. In other words, the effect of any rule altering the state of the “Java world” is only observable in the *next* step of the machine, which of course discourages programming in a sequential style: instead, any needed sequentiality will have to be made explicit, e.g. by using an FSM representation of the ASM. Also, field updates and method invocations performed in the same step will be performed — in due time — in an unspecified order, since update instructions in CoreASM constitute an unordered multiset. This behavior, too, may surprise the unaware Java programmer at his first approach with ASMs, as will be discussed in Sections 4 and 5.

<sup>4</sup> The same situation is found in other cases, e.g. when both  $a := \{1, 2\}$  and **add 3 to a** appear in the same step.

Nevertheless, we believe that the soundness of the semantics that is given by the deferred updates approach is worth the inconvenience, and can actually help even novice specifiers in drawing a clear line between what needs to be specified and the actual behavior (possibly, over-specified) of the implementation.

Formally, invocation of non-*void* methods is specified as follows:

	NonVoidMethodInvocationRule
<pre> (<b>invoke</b> <math>\alpha \llbracket e \rrbracket \rightarrow^\beta x(\lambda_1 \llbracket e_1 \rrbracket, \dots, \lambda_n \llbracket e_n \rrbracket)</math> <b>result into</b> <math>\gamma \llbracket l \rrbracket</math>) <math>\rightarrow</math>   <b>choose</b> <math>\lambda \in \{\alpha, \gamma, \lambda_1, \dots, \lambda_n\}</math> <b>with</b> <math>\neg \text{evaluated}(\lambda)</math>     <math>pos := \lambda</math>   <b>ifnone</b>     <b>if</b> <i>isObject</i>(<i>value</i>(<math>\alpha</math>))       <b>if</b> <i>hasMethod</i>(<i>Obj</i>(<i>value</i>(<math>\alpha</math>)), <math>x, \langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle</math>)         <b>if</b> <i>loc</i>(<math>\gamma</math>) <math>\neq</math> <i>undef</i>           <b>let</b> <math>u = \text{DefUpd}(\text{INVOKE}, (\text{loc}(\gamma), \text{value}(\alpha), x,</math>             <math>\langle jValue(value(\lambda_1)), \dots, jValue(value(\lambda_n)) \rangle))</math> <b>in</b>             <b>let</b> <math>jtl = (\text{"jasmChannel"}, \langle \rangle)</math> <b>in</b>               <math>\llbracket pos \rrbracket := (\text{undef}, \{\langle jtl, u, \text{jasmAction} \rangle\}, \text{undef})</math>           <b>else</b>             <i>Error</i>(‘Cannot update a non-location.’)         <b>else</b>           <i>Error</i>(‘No such method.’)       <b>else</b>         <i>Error</i>(‘Not a Java object.’)         </pre>	

As for the previous constructs, we do not detail here how static methods on classes are invoked, as the mechanism is totally analogous.

Due to space constraints, we do not detail also how Java exceptions thrown by invoked methods are handled. Intuitively, if an exception is returned two updates are produced, one storing the value of the exception (as an ASM *JObject*) in a designated location, and another one storing a different value to the same location. As a consequence, Java exceptions are mapped in ASM to conflicting updates, which can be caught via the TurboASM **try/catch** rule [27].

## 2.4 Type Conversion

JASMine operates in two type conversion modes: *implicit conversion* and *explicit conversion*. In the implicit mode, which is the default mode, JASMine automatically converts types between CoreASM and Java when needed. This reduces the hassle of type conversion and helps in writing more concise CoreASM specifications. Automatic type conversion, however, has its drawbacks in certain applications: it converts values even when such a conversion is not needed; e.g., when returned values of Java methods are to be passed as arguments in future calls to other Java methods. In the explicit mode, the user is responsible for explicitly converting values between Java and CoreASM using the provided CoreASM functions described further below.

JASMine constructs apply type conversion when needed, through the functions *javaValue* and *asmValue* that convert CoreASM values to Java objects and vice versa. These two functions are defined by cases as summarized in Table 1. In

**Table 1.** Type conversions between CoreASM and Java

Java type	CoreASM background
bool, Boolean	Boolean
byte, short, int, long, float, double, Byte, Short, Integer, Long, Float, Double	Number
char, Character	currently not supported
String	String
Set interface	Set
List interface	Sequence
Map interface	Function (dynamic)
arrays	currently not supported
any other object	JObject

most of the rules presented in this paper, the  $jValue$  function abstracts the details of type conversion based on the conversion mode.

It is anticipated that in most cases JObjects will be obtained from and passed to Java methods, and only few objects will be explicitly created in CoreASM through **import native** rules. A fuller definition of the conversion operations is provided in Section 3.2, as these have complex interactions with the facilities offered by the Java Reflection API, in particular with regard to the duality between primitive types (e.g., int) and wrapper classes (e.g., Integer) in Java.

Here we will mention that the JObject background offers two functions,  $toJava : ELEMENT \rightarrow JOBJECT$  and  $fromJava : JOBJECT \rightarrow ELEMENT$  which perform the same conversion on arbitrary values. As these conversions do not have any side effect, they can be performed immediately, and their semantics is trivial:

---

	TypeConversionRules
$\langle toJava(\alpha \square) \rangle \rightarrow pos := \alpha$	
$\langle toJava(\alpha v) \rangle \rightarrow \llbracket pos \rrbracket := (undef, undef, javaValue(v))$	
$\langle fromJava(\alpha \square) \rangle \rightarrow pos := \alpha$	
$\langle fromJava(\alpha v) \rangle \rightarrow$ <b>if</b> $isJObject(v)$	
$\llbracket pos \rrbracket := (undef, undef, asmValue(jObj(v)))$	
<b>else</b>	
$\llbracket pos \rrbracket := (undef, undef, uu)$	

---

## 2.5 Aggregation of Deferred Updates

As we have seen, any modification to the “Java world” is performed through special updated instructions, called deferred updates (but not to be confused with ASM updates), to ensure a stable state and environment inside a single ASM step. Three types of deferred updates are used by JASMine: instantiation (CREATE), field writing (STORE) and method invocation (INVOKE).

Each type of deferred update carries the information necessary for its execution; in particular, CREATE carries information on the Java class to create and



on the location of the new ASM element to create; `STORE` carries information about the `JObject` whose field is to be modified, about the name of the field to modify, and about the new value to be written in the field; `INVOKE` carries information about the `JObject` on which the method has to be invoked, about the name of the method, and about the (possibly empty) list of arguments to pass to the method.

The following compatibility conditions must be met for a set of updates to be considered consistent:

1. No other update is permitted on the ASM location used in a `CREATE`. Notice that this includes JASMine deferred updates (i.e., it is not possible to import twice to the same location) as well as regular updates (i.e., it is not possible to assign a different value through the assignment operator `:=` or other update rules to a location used in a `CREATE`).
2. If multiple `STORE`s are performed on the same field of the same object, they must all assign the same value.
3. Any location used to store the result of an `INVOKE` cannot appear in any other update.

Notice that this latter condition is sufficient, but not necessary to guarantee consistency. In fact, we disallow even multiple updates that would write the same value (which are normally permitted under standard ASM semantics). The reason for this more restrictive choice is that in general it is impossible to know which value will be returned by a method call without actually calling the method, and we want the method to be called only if a consistent set of updates is generated. Hence, we require a stronger guarantee than what is strictly needed.

If the set of update instructions is consistent, the prescribed operations are performed *in unspecified order*. Notice that the first condition above ensures that newly-created `JObjects` are not used in the same step, so there is no need to specify a special ordering with `CREATE` update instructions performed before `STORE` and `INVOKE` ones.

A common troublesome case is when multiple method invocations are performed: if the particular sequence is order-sensitive, ordering will have to be specified explicitly by means of `seq` rules or by using a finite state automaton. In most cases, though, the specific order will be immaterial (e.g., `Point.setX()` and `Point.setY()`), and in these cases multiple invocations can well be specified in the same step. We regard this as a desirable feature for a specification: in fact, the implementer will know that fields can be written and that methods can be invoked in any order as long as they are specified to happen in a single ASM step, whereas the ordering between different steps is significant, and should be respected in the implementation.

### 3 Implementing JASMine

In its capacity as a bridging technology, JASMine has to interact closely with both the CoreASM engine and the Java virtual machine. We will discuss these interactions in the following.

### 3.1 Interacting with the CoreASM Engine

The CoreASM extensibility architecture [9] dictates that plug-ins extending the basic CoreASM language have to implement one or more interfaces, depending on which elements of the language (both syntax and semantics) and of the computation cycle are contributed. In particular, JASMine provides the following extensions:

- It implements the *parser plug-in* interface to extend the parser with new syntax for native import, field read/write, and method invocation. The syntax rules contributed to the language correspond to the syntactical patterns shown in Section 2.
- It implements the *interpreter plug-in* interface and contributes the semantics for the new syntactical patterns. The semantics contributed correspond to the ASM rules shown in Section 2.
- It implements the *vocabulary extender* interface to extend the CoreASM state with the `JObject` background and the monitored *jasminChannel* function. In particular, the two casting function *toJava* and *fromJava* are introduced as part of the `JObject` background. Moreover, element equality, ordering and conversion to a `String` value are forwarded to the Java object represented by any given `JObject` value.
- It implements the *aggregator* interface to provide aggregation rules which encode all the JASMine update instructions computed in one step into one single update to the *jasminChannel* location.
- To actually communicate with the Java virtual machine, the value of *jasminChannel* must be read after every successful step and the actions encoded therein must be parsed and applied to the corresponding Java objects. To perform this, the JASMine plug-in extends the lifecycle of the CoreASM engine and reads the value of *jasminChannel* whenever the control state of the engine is switched to *Step Successful*, i.e. whenever a step is completed with a consistent set of updates; it then executes all the CREATE, STORE and INVOKE operations stored in *jasminChannel*.

### 3.2 Interacting with the JVM

Interaction between JASMine and the Java Virtual Machine is limited to a few, well-defined operations, and is mostly mediated by the Java Reflection API [16].

The application of updates encoded in *jasminChannel* entails the following steps:

1. for CREATE updates, the classical `Class.forName()` method is invoked, passing a string representation of the imported class name. Once a `Class` object for the desired class is obtained, if the nullary version of `import native` was used the `Class.newInstance()` method is invoked to obtain the instance. Otherwise, `Class.getConstructor()` is called to retrieve the corresponding constructor, then the constructor's `newInstance()` method is called, with the given arguments, to obtain the instance. A new `JObject` element encapsulating the new instance is then created and assigned to the ASM location provided in the CREATE record.

2. for STORE updates, the class of the referenced object is obtained by calling `getClass()` on the reference held by the `JObject`; the `Field` object is then retrieved through `Class.getField()`, and finally `Field.set()` (or one of its primitive types variant) is called to assign the value from the STORE record.
3. for INVOKE updates, the class of the referenced object is obtained as above, then the matching `Method` object is retrieved through `Class.getMethod()` (notice that in this way only public methods can be retrieved), and finally `Method.invoke()` is called, with the appropriate parameters from the INVOKE record. If the method was non-void, the resulting value is then stored in the ASM location provided in the INVOKE record.

It is worthwhile to remark that fields and methods name resolution is entirely delegated to the Reflection API, and thus follows the normal resolution algorithm in Java (see [11], sections 8.2 and 8.4).

Evaluation of field read access is performed immediately upon encountering the corresponding expression, by first obtaining the `Field` object as for STORE updates, then invoking `Field.get()` (or one of its primitive types variants) to retrieve the field value, which is then returned as the expression's value. These operations constitute the `GetField` macro used in the semantics (Section 2.2).

The various functions used in sections 2.1 to 2.3 (*isJavaClassName*, *hasEmptyConstructor*, *hasConstructor*, *hasField*, *hasMethod*) are directly mapped to the corresponding Reflection API methods. All these predicates are implemented by trying to access the given class, constructor, field or method and possibly catching the various exceptions (`ClassNotFoundException`, `NoSuchMethodException`, `NoSuchFieldException`) thrown by the Reflection API methods.

The function *jObj* returns a reference to the Java object encapsulated by a `JObject`.

Finally the conversion functions *javaValue* and *asmValue* are implemented by cases, as summarized in Table 1. In particular, when converting from CoreASM elements to Java values (*javaValue* function), Booleans and numbers are simply converted to the corresponding primitive types in Java (with possible boxing if a wrapper class is needed); numbers are generally converted to double, then downcast as needed to fit smaller types. CoreASM's strings are wrappers around Java strings, so the conversion is trivial. More complex mathematical structures (e.g., set or sequences) are generally implemented in CoreASM as wrappers to the various Java Collections API objects, so in this case also conversion amounts to unwrapping the underlying object. Any other CoreASM value is upcast to `Object` and passed as-is, thus realizing an opaque container for the ASM value from the point of view of Java code.

Conversion from Java values to CoreASM elements (*asmValue* function) is similar, except that any unrecognized Java object is wrapped in an opaque `JObject` from the point of view of ASM code. This allows access to fields and invocation of methods of objects returned from other Java methods, as in

```
invoke calendar->getCurrentDate() result into today
```

followed, in a subsequent step, by

```
wday := today->weekDay
invoke today->add(7) result into nextWeek
```

## 4 A Simple Example

In this section we show a simple example of an ASM using JASMine constructs. Our example will first instantiate an object from a Java class we have written (and whose `.class` file is put on the classpath of the CoreASM engine), and then invoke two methods on that object. Notice that, despite its simplicity, this is typical of the expected usage pattern: complicated computations, library interactions, and especially sequential processing should be confined in Java code. The CoreASM specification should interact with the “Java world” at high level, through methods of some semantic significance, leaving the nitty-gritty to ad-hoc Java code.

The example specification is as follows:

```
1 CoreASM JASMineExample
2 use StandardPlugins
3 use JASMinePlugin
4 init InitRule
5
6 rule InitRule =
7     if mode = undef then
8         mode := 1
9         import native org.jasmine.example.Foo into foo
10    else if mode = 1 then
11        mode := 2
12        invoke foo->setMsg("How are you?")
13        invoke foo->getTime() result into t
14    endif
```

Here, line 3 instructs the CoreASM engine to use the JASMine plug-in (in addition to the standard plug-ins mentioned on line 2), as is customary in CoreASM’s extensible architecture [\[9,7\]](#).

Then, as `mode` is not initially defined, lines 8-9 are executed. In particular, line 9 instantiates an object of the `org.jasmine.example.Foo` class — more in detail, it generates a deferred update (assigning a new `JObject` holding a reference to the new Java object to the location `foo`). The first ASM step is now complete. The set of update instructions generated includes assigning 1 to `mode` as well as the following special JASMine update instruction:

```
< ("jasmChannel", <>),
  DefUpd(CREATE, ("foo", <>), "org.jasmine.example.Foo", <>),
  jasmAction >
```

The JASMine update aggregator is called to handle the special update instruction above. The aggregator creates a new `JObject` (call it  $v$ ) and adds the the current

update set a new update assigning this `JObject` to `foo`. It also saves a reference to `v` and defers the instantiation of the Java object `org.jasmine.example.Foo` to the end of the computation step.

As there is no inconsistency, the updates are then applied to the state. the JASMine plug-in comes in again (as an extension point plug-in) and decodes the value of `jasMChannel`. Decoding the CREATE action, JASMine instantiates the new Java object and modifies `v` so that it refers to this new Java object.

The engine then continues execution in the next step. This time, due to the new value of `mode`, lines 11-13 are executed. In addition to setting the value of `mode` to 2, deferred updates for the two method invocations are generated, namely

```
⟨ ("jasMChannel", ⟨⟩),
  DefUpd(INVOKE, (undef, v, "setMsg", ⟨ "How are you?" ⟩),
    jasMAction )
```

and

```
⟨ ("jasMChannel", ⟨⟩), DefUpd (INVOKE, (⟨ "t", ⟨⟩), v, "getTime", ⟨⟩), jasMAction)
```

Again, as there is no conflict, updates will be applied and the JASMine plug-in will execute (in an unspecified order) both method calls. It is worthwhile to remark here that the arguments (e.g., the "How are you?" string passed to `setMsg()`) will be converted to the corresponding Java type prior to the call. Moreover, as result of the aggregation phase an update will be generated assigning the result from the `getTime()` to the location `t`. For example, if the current time is represented by the (Java) integer with value `time`, the update generated by the JASMine aggregator will be

```
⟨ (⟨ "t", ⟨⟩), time, updateAction)
```

(notice that this is an update instruction encoding a standard ASM update, and that — as for invocation — the Java integer value is converted to an ASM number in the process).

After the updates to `mode` and `t` are applied to the state, no update is produced anymore, and the computation is finished.

## 5 Pragmatics and Future Work

As we have remarked in our previous discussion, we have chosen faithfulness to the theoretical ASM model as a guiding principle in defining the semantics of JASMine. However, this choice has important pragmatic implications which merit to be discussed.

In particular, JASMine presents a *stable view of the Java environment* to the ASM. This is required by ASM semantics, but may be inconvenient in practice, as any action performed on a Java object (e.g., storing a value in a field or

invoking a method) will produce observable effects only in the *next* step of the machine: thus, many programming patterns typical of sequential programming cannot be applied. This is also true in the case of TurboASM rules: hence, the  $n$ -th step in a **seq** or **iterate** rule will *not* observe the effects on the environment of the previous  $n - 1$  steps, as the corresponding updates are being deferred as described in Section 2.5. This is due to the impossibility of rolling back the Java environment to a previous state, which prevents speculative execution of the inner steps of a TurboASM step (which is what is done instead for the ASM state). For example, a **while** cycle like

```

1  import native java.io.File into file
2  ...
3  while (lastModified <= lastActed)
4      invoke file->lastModified() result into lastModified
5  ...

```

which could be used to wait for a modification to a file, will not work as expected: in fact, invocations to `lastModified()` will be deferred until the end of the step, hence after the **while**, probably defeating the programmer's intention.

In terms of style, one could argue that such behavior should be either encapsulated inside a single Java method `waitModification()` (to be invoked through JASMine), or — if the details of how modifications are detected are significant enough — lifted up to the top level of the ASM specification.

However, another possibility would be to offer alternative semantics. As part of future work, we intend to specify and implement three other strategies for managing modifications to the Java environment inside a sub-machine, namely: (i) immediately executing updates due to field stores and method invocations; (ii) disallowing field stores and method invocations in sub-machines altogether, or (iii) caching field stores so that subsequent field reads will return the last *speculatively* stored value, disallowing or deferring method invocations. The specification writer should be able to declare, in the specification itself, which of the four semantics is desired. There is no clear winner among these alternative semantics: (i) breaks the stable environment postulate (as does AsmL), whereas (iii) is not regular, treating field stores and method invocations in different ways, and (ii) is too restrictive, by imposing sufficient but not necessary conditions on the specification.

Also as part of future work we intend to complete the support of certain Java constructs which would have a natural mapping in CoreASM. Among those, access to Java enumerations as ASM universes and access to static methods and fields of uninstantiable classes (e.g., the `Math` class) are immediately useful candidates. We are also exploring the possibility of a similar integration with the other major framework, namely Microsoft's .NET.

## 6 Conclusions

In this paper we have introduced JASMine, a plug-in for the CoreASM environment allowing access to Java objects and classes from ASM specifications. The

extension has been achieved without compromising on the CoreASM project goals, namely providing an executable ASM language which preserves the pure ASM semantics, and ensuring freedom of experimentation through typelessness and extensibility.

We regard JASMine as an *enabling technology* which opens a number of novel application areas for CoreASM. Possible applications include:

- CoreASM specifications can be used as drivers for “real” software, e.g. as part of a test suite, as integration bridges, or mediators between independent components;
- it is possible to realize automated black-box testing, whereas a specification (in CoreASM) and a corresponding implementation (in Java) are executed in step-wise parallel fashion, with the progress of both being checked one against the other at each step;
- accessing the rich set of graphical UI components available as Java libraries, it becomes easy to implement GUIs for CoreASM specifications, for example to have an animated, executable specification in a rapid prototyping scenario;
- a huge collection of high-quality code is available as Java libraries; these can be used as implementation of very complex functions (e.g., an MPEG4 decoder) for which no specific background plug-in is available in CoreASM;
- in a similar way, uninteresting (but needed) parts of a CoreASM specification could be directly written in Java, especially when they consist of “standard” programming code;
- the CoreASM engine itself can be invoked and controlled from Java code through its Control API: hence, it is possible to build two-ways integration, whereas a CoreASM can both be called from and call Java code.

We believe JASMine to be both a clear demonstration of the usefulness of and flexibility afforded by the CoreASM extensible architecture, and a valuable addition to every specifier’s toolbox, empowering him or her to make recourse to executable yet pure ASMs in a number of problems hitherto difficult to tackle.

**Acknowledgments.** We would like to thank Antonio Cisternino for his contribution to an early definition and implementation of the JASMine plug-in, and Egon Börger for many discussions clarifying some of the more difficult issues that we faced in the design of the JASMine semantics. We would also like to thank the anonymous reviewers for their valuable feedback on this paper.

## References

1. Anlauff, M.: XASM - an extensible, component-based abstract state machines language. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 69–90. Springer, Heidelberg (2000)
2. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
3. Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 311–325. Springer, Heidelberg (1999)

4. Del Castillo, G., 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)
5. Farahbod, R., et al.: The CoreASM Project, <http://www.coreasm.org>
6. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: Proc. of the 12th Int'l Workshop on Abstract State Machines (2005)
7. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae* 77, (March/April 2007) (to be published)
8. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 71–103 (2007)
9. Farahbod, R., Gervasi, V., Glässer, U., Ma, G.: CoreASM plug-in architecture (2007) (submitted to the same Festschrift volume)
10. 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)
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Prentice Hall, Englewood Cliffs (2005)
12. Memon, M.A.: Specification language design concepts: Aggregation and extensibility in coreasm. Master's thesis, Simon Fraser University, Burnaby, Canada (April 2006)
13. Microsoft Corp. Microsoft. NET Framework, <http://www.microsoft.com/net> (Last visited December 2006)
14. Microsoft FSE Group. The Abstract State Machine Language, <http://research.microsoft.com/fse/asml/> (Last visited June 2003)
15. Schmid, J.: Executing ASM Specifications with AsmGofer, <http://www.tydo.de/AsmGofer/> (Last visited September 2005)
16. Sun Microsystems, Inc. The Java 2 Platform Standard Edition 5.0 API Specification. Sun Microsystems, Inc. (2004), <http://java.sun.com/j2se/1.5.0/docs/api>



# A Modular Verification Methodology for C# Delegates

Peter Müller and Joseph N. Ruskiewicz

ETH Zurich, Switzerland

`peter.mueller@inf.ethz.ch`, `joseph.ruskiewicz@inf.ethz.ch`

**Abstract.** Function objects are used to express higher-order features in object-oriented programs. C# provides the delegate construct to simplify the implementation of function objects. A delegate instance represents a method together with a target object. Sound reasoning about delegates requires that the precondition of the underlying method holds whenever a delegate is invoked. This is difficult to achieve if the method precondition depends on the state of the target object. Proving such a precondition when the delegate is invoked is in general not possible because properties of the target object are typically not known at the invocation site. Proving the precondition when the delegate is instantiated is not sufficient either because the state of the target might change before the delegate is invoked. In this paper, we present a verification methodology for C# delegates. Properties of the target object are expressed as invariant of the delegate. Our methodology keeps track when this invariant can be assumed to hold. It enables modular verification of interesting implementations and is proven sound.

## 1 Introduction

Higher-order features are a common programming idiom. Typical examples include a generic sort algorithm whose comparison method is passed as parameter, an algorithm that approximates an integral of a function which is passed as a method reference, and a GUI that stores references to methods that are called upon certain events.

In object-oriented programs, references to methods are encoded as *function objects*. A function object represents a method, possibly with some actual method arguments. Function objects are often implemented using the Command pattern, whose class diagram is shown in Fig. 1. A function object is an instance of class *ConcreteCommand*. As described by Gamma *et al.* [8], a function object stores exactly one actual argument of the underlying method, namely its target. The target is fixed when the function object is created. The function object is invoked by calling its *Execute* method, which will call *Action* on the stored target object.

In C#, the Command pattern is built into the programming language in the form of delegates [6]. Each delegate type corresponds to a *ConcreteCommand* class. It prescribes the signature of the underlying method, but not its name. The name—like the target object—is determined when the delegate is instantiated.

We illustrate delegates by an implementation of a simple storage system. We use the delegate *Archiver* (Fig. 2) to create function objects for the store methods of different archives. Method *Client.Log* takes an *Archiver* instance as parameter and invokes it. Class *TapeDrive* (Fig. 3) implements such an archive. The boolean field *IsLoaded* is

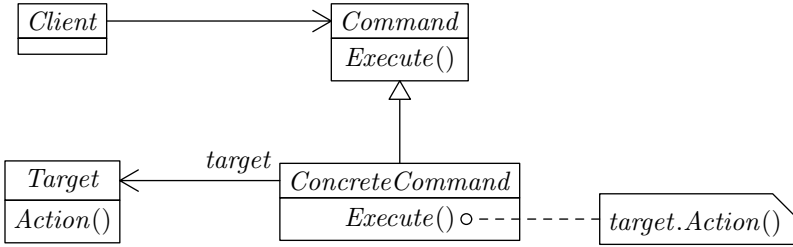


Fig. 1. The command pattern

```

delegate void Archiver(object p)
    requires p ≠ null ∧ IsPeerConsistent(this) ;

class Client {
    static void Log(Archiver logFile, string s)
        requires logFile ≠ null ∧ IsPeerConsistent(logFile) ;
        requires s ≠ null ;
        { logFile(s) ; }
}
    
```

Fig. 2. A client of the storage system. We explain the annotations along with the presentation of our methodology.

*true* if and only if a tape is loaded into the drive. Hence, the *Store* method requires *IsLoaded* to be *true*, and method *Eject* sets *IsLoaded* to *false*. In method *Main* (Fig. 4), *Archiver* is instantiated with method *Store* and target *tapeDrive*.

The invocation of a delegate instance triggers a call of the underlying method on the stored target object. In our example, the invocation of *logfile* in method *Client.Log* (Fig. 2) triggers a call of *tapeDrive.Store*. A sound verification methodology has to ensure that the *requires* clause of this method holds in the prestate of the call. This is difficult to achieve if the *requires* clause of the method depends on the state of the target object. This verification challenge was pointed out in an earlier paper [10], from which we took the storage example to show that our methodology can handle it.

The problem is illustrated by method *Store*, whose second *requires* clauses refers to the *IsLoaded* field of its target. Proving this condition when a delegate instance is invoked is not possible because properties of the target are typically not known at the invocation site. For instance, method *Client.Log* does not have any knowledge about the particular archive used by *logFile*. In fact, *logFile* could even represent a static method such that no target exists. Proving the condition when a delegate is instantiated with *tapeDrive.Store* is not sufficient either because the state of *tapeDrive* might change before the delegate is invoked. For instance, if method *Main* called *tapeDrive.Eject* before calling *Client.Log*, the invocation of the delegate would violate *Store*'s *requires* clause, even though the condition held when the delegate was instantiated.

---

```

class TapeDrive {
  bool IsLoaded ;

  void Store (object p)
    requires p ≠ null ∧ IsPeerConsistent(this) ;
    requires IsLoaded ;
  { ... }

  void Eject ()
    requires IsPeerConsistent(this) ∧ IsLoaded ;
    ensures ¬IsLoaded ;
  { expose (this for TapeDrive) { IsLoaded := false ; } }

  void ChangeMedia ()
    requires IsPeerConsistent(this) ∧ IsLoaded ;
  {
    expose (this for TapeDrive) {
      IsLoaded := false ; /* exchange media */ IsLoaded := true ;
    }
  }

  // Constructors and other methods omitted.
}

```

**Fig. 3.** An implementation of a tape archive. `expose` blocks are used by the Boogie methodology to denote regions in which an object invariant is potentially violated.

---

```

static void Main(string[] args) {
  TapeDrive tapeDrive := new TapeDrive() ;
  Archiver archiver := new Archiver(tapeDrive.Store) ;
  Client.Log(archiver, "HelloWorld") ;
}

```

**Fig. 4.** Main method for the storage example

---

In the Command pattern, the desired condition could be expressed as invariant of the function object; that is, class *ConcreteCommand* could declare the object invariant  $target \neq \text{null} \wedge target.IsLoaded$ . This invariant is strong enough to guarantee that *Store*'s `requires` clause is satisfied whenever the function object is invoked. However, modular reasoning about invariants that depend on the state of several objects (here, the function object and its target) is difficult, and existing solutions are not suitable for function objects. Ownership-based invariants [11, 13] would require the function object to own the target object. This means that the target could be modified only through a single function object, which is clearly too restrictive. Solutions based on visibility-based invariants [13] lead to complicated proof obligations that essentially quantify over all instances of a delegate type, which are difficult to verify. Solutions based on friendship invariants [3] introduce a ghost state in each target object to keep track of function objects attached to it. This ghost state is used to impose the appropriate proof obligations

about invariants of function objects whenever the state of the target is changed. Handling the ghost state leads to significant specification and verification overhead.

In this paper, we present a modular verification methodology for C# 1.0 delegates using delegate invariants. Our methodology is based on the Boogie methodology for object invariants [11] and can be adopted by the Spec# programming system [2]. It exploits the syntactic structure of delegates to generate simple proof obligations while keeping the annotation burden small. In particular, our methodology does not introduce any overhead for programs that do not use delegates, and very little overhead for the common applications of delegates.

**Overview.** This paper is organized as follows. Sec. 2 summarizes those parts of the Boogie methodology for object invariants that are used in the rest of the paper. Sec. 3 explains our methodology informally. The technical details including a soundness result are presented in Sec. 4. We review related work in Sec. 5 and offer conclusions in Sec. 6.

## 2 Background on Boogie Methodology

In this section, we summarize those parts of the Boogie methodology for object invariants [11] that are needed in the rest of this paper. The motivation for the design and the technical details are presented in an earlier paper [11].

**Meaning of Invariants.** To handle temporary violations of object invariants, the Boogie methodology introduces for every object the concrete field *inv* that represents explicitly whether the object invariant is required to hold. The *inv* field ranges over class names. If  $o.inv <: T$  for an object  $o$  of type  $T$  (where  $<:$  denotes the reflexive sub-type relation), then  $o$ 's invariants declared in class  $T$  and its superclasses must hold and we say  $o$  is *valid* for  $T$ . If  $o$  is not valid for  $T$  then the invariant of  $o$  declared in  $T$  is allowed to be temporarily violated and we say  $o$  is *mutable* for  $T$ . We say  $o$  is *fully valid* if it is valid for its dynamic type:  $o.inv = \text{typeof}(o)$ .

The *inv* field is modified only by a special block statement: **expose** ( $o$  for  $T$ ) {  $\mathcal{S}$  }. Before executing the statement  $\mathcal{S}$ , object  $o$  is *exposed*, that is,  $o.inv$  is set to  $T$ 's superclass. After the execution of  $\mathcal{S}$ ,  $o$  is *un-exposed* by checking that the object invariant declared in class  $T$  holds for  $o$  and then setting  $o.inv$  back to  $T$ .

Since the update of a field  $o.f$  potentially breaks the invariant of  $o$ ,  $o.f$  is allowed to be assigned to only at times when  $o$  is mutable for the class  $F$  that declares  $f$ . To enforce this policy, each update of  $o.f$  is guarded by an assertion  $\neg o.inv <: F$ . The assertions for field updates and **expose** statements ensure that throughout the program execution the following program invariant holds:

$$\text{PI: } (\forall o, T \bullet o.inv <: T \Rightarrow \text{Inv}_T(o))$$

where  $\text{Inv}_T(o)$  expresses that the invariant declared in class  $T$  holds for object  $o$ . The quantification ranges over non-null allocated objects.

**Aggregate Objects.** The Boogie methodology uses *ownership* to handle invariants of aggregate objects, that is, an aggregate object is the owner of its component objects. Updating a field of a component object potentially affects the invariant of the aggregate

object. Therefore, the Boogie methodology enforces that the aggregate object is mutable whenever one of its component objects is mutable.

The Boogie methodology encodes ownership by adding a field *owner* to each object. This field ranges over pairs  $\langle obj, typ \rangle$ , where *obj* is the owner object and *typ* is a superclass of the dynamic type of *obj* at which the ownership is established. Like *inv*, *owner* cannot directly be assigned to. The owner of an object is set when the object is created. Because it would be a distraction in this paper, we omit the program statement for changing the *owner* field (but see [11]).

The relation between the validity of an aggregate object and its components is expressed by the following program invariant:

$$P2: (\forall o, T \bullet o.inv <: T \Rightarrow (\forall p \bullet p.owner = \langle o, T \rangle \Rightarrow p.inv = \mathbf{typeof}(p)))$$

We say that two objects are *peers* if they have the same owner. A method *m* typically requires its target and all of its peers to be fully valid, which allows *m* to call methods with the same requires clause on these objects. Moreover, *m* typically requires the owner of its target to be *sufficiently exposed*, that is, the owner object is mutable for the owner type. This allows *m* to expose its target. An object *o* that satisfies these two requirements, is called *peer-consistent*. In specifications, peer-consistency of an object *o* is expressed by the following predicate:

$$IsPeerConsistent(o) \equiv (\forall p \bullet p.owner = o.owner \Rightarrow p.inv = \mathbf{typeof}(p)) \wedge (o.owner.obj \neq \mathbf{null} \Rightarrow \neg o.owner.obj.inv <: o.owner.typ)$$

By the program invariants of the Boogie methodology, peer-consistency of an object *o* implies that the invariants of *o*, *o*'s peers, and all objects owned by these objects are fully valid.

**Static Verification.** The proof rules of the Boogie methodology are formulated as assertions, which cause the program execution to abort if evaluated to *false*. For static verification, each assertion is turned into a proof obligation, which is proved using an appropriate program logic. As justified by earlier work [11], one may assume the program invariants for this proof. All of the proof obligations can be generated and shown modularly. That is, a class *C* can be verified based on the specifications of the classes used by *C*, but without knowing the complete program in which *C* will be used.

### 3 Main Concepts

In this section, we present our verification methodology informally. Our presentation will focus on single cast delegates, that is, delegates with exactly one underlying method and target. C# also provides for multicast delegates whose invocation triggers calls to several methods and targets. An extension of our methodology to multicast delegates follows directly from single cast delegates.

#### 3.1 Delegate Specifications and Refinement

In the Command pattern (Fig. 1), invocations of function objects are verified using the specifications of method *Execute*. Since *Execute* simply calls *target.Action*, the requires clause of *Execute* must be strong enough to guarantee *Action*'s requires clause,

and the converse holds for ensures clauses. In other words, the specification of *Action* must refine the specification of *Execute*.

To adapt this approach to delegates, we associate each delegate declaration with a specification similar to method specifications. In this paper, we focus on requires and ensures clauses for delegates and assume that frame conditions are encoded in the ensures clause. When a delegate  $D$  is instantiated with a method  $o.m$ , one has to prove that  $m$ 's specification (with  $o$  for **this**) refines  $D$ 's specification. More precisely, one has to prove that  $D$ 's requires clause is stronger than  $m$ 's and that  $D$ 's ensures clause is weaker than  $m$ 's when  $D$ 's requires clause holds. At the invocation site of the delegate, it suffices to prove that the requires clause of  $D$  holds, which implies that the weaker requires clause of  $m$  holds as well. Conversely, one may assume  $D$ 's ensures clause after the invocation.

As explained in Sec. 2 most methods in the Boogie methodology require their target to be peer-consistent. To support this idiom, we arrange for delegate instances and their target objects to be peers. Therefore, we may assume that the target is peer-consistent whenever the delegate instance is peer-consistent. The peer relationship between a delegate instance  $d$  and its target  $o$  is established when  $d$  is created and maintained afterwards. Therefore, the following program invariant holds in all execution states.

$$\text{P3: } (\forall o, d \bullet d.\text{target} = o \wedge d.\text{inv} = \text{typeof}(d) \Rightarrow d.\text{owner} = o.\text{owner})$$

The refinement of specifications is illustrated by the instantiation of delegate *Archiver* (Fig. 2) with *tapeDrive.Store* in method *Main* (Fig. 4). We ignore for the moment the second requires clause of method *Store* (Fig. 3), which will be discussed in the next subsection. *Archiver*'s requires clause implies *Store*'s first requires clause because (1) the first conjunct,  $p \neq \text{null}$ , appears in both requires clauses, and (2) because of program invariant P3, the target is peer-consistent whenever the delegate instance is peer-consistent. The default ensures clause, **true**, of *Store* trivially implies the default ensures clause of *Archiver*. Still ignoring *Store*'s second requires clause, *Store*'s specification refines the specification of *Archiver*, which allows us to verify the delegate instantiation in method *Main*. When the delegate is invoked in method *Client.Log* (Fig. 2), we have to prove that the requires clause of the delegate is satisfied, which follows trivially from the requires clause of *Client.Log*.

Equipping delegates with requires and ensures clauses, and checking a refinement relation when a delegate is instantiated allows us to verify most applications of delegates. We looked at all delegate instantiations in Microsoft's compiler framework CCI and the Spec# compiler. The vast majority of delegates are instantiated with static methods, for which the methodology introduced so far is sufficient as static methods do not have target objects. It is also sufficient for instance methods whose requires clauses do not refer to the state the target besides requiring validity or peer-consistency. In the rest of this section, we discuss how to handle the remaining cases such as method *Store*, whose second requires clause requires the *IsLoaded* field of the target to be *true*.

### 3.2 Delegate Invariants

We allow delegates to declare invariants that may refer to the state of the target. Analogously to C#, we assume that each delegate has an immutable field *target* that holds

a reference to the target. An invariant of the form **invariant for**  $T$  is  $P(\text{target})$  expresses that if the target is a non-null object of class  $T$  then it satisfies  $P$ . Such an invariant declared in a delegate type  $D$  is desugared into the invariant  $o.\text{target is } T \Rightarrow P((T)o.\text{target})$ . Note that our notation implicitly casts  $\text{target}$  to  $T$ . In our example, an invariant for *Archiver* could require that if its target is a reference to a *TapeDrive* object, then its *IsLoaded* field is *true*:

```
invariant for TapeDrive is target.IsLoaded ;
```

With this invariant, it is trivial to show that the specification of *Store* refines the specification of *Archiver*, in particular, that *Store*'s second requires clause *IsLoaded* is implied by *Archiver*'s require clause *IsPeerConsistent(this)*. By program invariant P1, peer-consistency of the *Archiver* instance implies that its invariant holds. With the appropriate substitution, this immediately yields *Store*'s second requires clause. Therefore, the instantiation of *Archiver* in method *Main* (Fig. 4) verifies.

**Delegate Subtypes.** As illustrated by the above invariant, delegate invariants specify a type for the *target* object in order to access its fields. This means that the specifier of the delegate has to foresee that the delegate might be instantiated with a method of that type. This deprives delegates of their flexibility. In particular, adding a new class *DiskDrive* with method *Save* to the program in general requires an additional invariant for *Archiver*, which cannot be added without changing the existing code.

To solve this problem, we allow programmers to declare subtypes of delegates, which may refine the specification of the supertype. Instead of adding the above invariant to *Archiver*, we declare the following subtype:

```
delegate TapeArchiver : Archiver  
invariant for TapeDrive is target.IsLoaded ;
```

Delegate subtypes prescribe identical signatures as their supertypes, which is not repeated in the subtype declaration. Moreover, subtypes inherit the specifications of their supertypes to enforce behavioral subtyping [5], but they may refine the inherited specifications. In particular, delegate subtypes are allowed to declare additional invariants.

To make use of the invariant of *TapeArchiver*, we have to adapt method *Main* (Fig. 4) to instantiate *TapeArchiver* rather than *Archiver*:

```
Archiver archiver := new TapeArchiver(tapeDrive.Store) ;
```

Since *TapeArchiver* is a subtype of *Archiver*, the instance *archiver* can be passed to method *Client.Log* without further adaptations. In particular, *Client.Log* (Fig. 2) need not be aware of the existence of the delegate subtype.

Note that delegate subtypes are merely a specification construct that allows us to associate invariants with delegates. In particular, they do not affect program execution. When a program is compiled, all occurrences of delegate subtypes can be replaced by their supertypes, and the subtype declarations can be eliminated.

**Maintaining Delegate Invariants.** Our verification methodology treats delegate instances basically like other objects. In particular, every delegate instance  $d$  has a field *inv* that indicates which invariants of  $d$  may be assumed to hold.



The invariant of a delegate instance  $d$  may depend on the immutable field  $d.target$  and on fields of the object referenced by  $d.target$ . Therefore, the only operations that potentially violate  $d$ 's invariant are modifications of the state of  $d$ 's target. Consequently, programs never have to expose  $d$  in order to change its own state, but  $d$  must be exposed before its target object is modified. In other words, whenever a field  $o.f$  is modified, we have to enforce that all delegate instances whose invariants depend on  $o.f$  are exposed. We achieve that as follows:

1. *Visibility requirement*: If a delegate  $D$  declares or inherits an invariant for  $T$  then class  $T$  must contain the *dependent clause* **dependent**  $D$ . Otherwise,  $D$ 's invariant is not admissible and will be rejected by the compiler. Conversely, if a class  $T$  contains **dependent**  $D$  then delegate  $D$  must declare or inherit an invariant for  $T$ . Otherwise,  $T$ 's specification is not admissible.
2. *Automatic expose*: We adapt the semantics of **expose** as follows. Besides object  $o$ , **expose** ( $o$  for  $T$ ) exposes each instance  $d$  of a delegate  $D$  where  $d.target = o$  and  $D$  is mentioned in a dependent clause of class  $T$ . At the end of the **expose** block,  $d$ 's invariants are checked and  $d$  is un-exposed.

The visibility requirement ensures that the dependent clause of a class  $T$  lists all delegates whose invariants are potentially broken by updates of fields of  $T$ . This allows us to determine in a modular way all the invariants that have to be checked at the end of an **expose** block.

The automatic expose guarantees that whenever a target object  $o$  is mutable for a class  $T$ , then all dependent delegate instances  $d$  are also mutable. The following program invariant states the contraposition of this property.  $D \in \text{dependents}(T)$  expresses that class  $T$  contains a depends clause **dependent**  $D$ .

$$\text{P4: } (\forall o, d, T, D \bullet d.target = o \wedge d.inv <: D \wedge D \in \text{dependents}(T) \wedge \text{typeof}(o) <: T \Rightarrow o.inv <: T)$$

The visibility requirement and automatic expose guarantee that P1 also holds for delegate instances, where  $Inv_D$  denotes the desugared delegate invariant of delegate  $D$ .

The visibility requirement seems to be a severe restriction since it forces a class  $T$  and a dependent delegate  $D$  to be implemented together as they refer to each other in their dependent clause and delegate invariant. However, the requirement is not overly restrictive for the practical examples we have considered. First, as stated above, most delegates do not have invariants at all. Second, if the implementer of  $T$  wants to use an existing delegate  $D$ , they can declare a subtype of  $D$  that contains the invariant for  $T$ . This is illustrated by *Archiver* and *TapeArchiver*. Third, if a delegate  $D$  needs to declare an invariant for an existing class  $T$ , it is not possible to add a dependent clause to  $T$ . In that case, one can declare a subtype  $S$  of  $T$  or a wrapper class  $S$  for  $T$  and establish the relation between  $D$  and  $S$ .

In our example, the invariant of *TapeArchiver* refers to the *IsLoaded* field of class *TapeDrive*. Therefore, we have to add the following dependent clause to *TapeDrive*:

**dependent** *TapeArchiver* ;

Method *ChangeMedia* (Fig. 3) illustrates how delegate invariants are verified. Because of the above dependent clause, the statement **expose** (**this** for *TapeDrive*) exposes



each *TapeArchiver* instance  $d$  where  $d.target = \mathbf{this}$ . The subsequent update of *IsLoaded* violates the invariant of  $d$ . However, since  $d$  is exposed, this violation is permitted by P1. At the end of the **expose** block,  $d$ 's invariant is asserted. Since *IsLoaded* is set to *true* before the end of the block, this assertion holds.

It is important to understand how our methodology prevents delegate invocations when the requires clause of the underlying method does not hold. Consider an execution of method *ChangeMedia* on target object  $o$ , and let  $d$  be a *TapeArchiver* instance representing  $o.Store$ . We show how our methodology prevents *ChangeMedia* from invoking  $d$  between the two updates of *IsLoaded*, that is, when *IsLoaded* is *false* and, thus, the second requires clause of *Store* does not hold. *TapeArchiver* requires *IsPeerConsistent(this)*. Therefore, an invoker of  $d$  must prove that  $d$  is fully valid. However, according to program invariant P4, this is not the case while its target  $o$  is mutable, and the invocation does not verify. Note that if *TapeArchiver* would not require *IsPeerConsistent(this)* then an instantiation with *Store* would not verify because *TapeArchiver*'s invariant is needed to prove the refinement relation.

**Disabling Delegates.** Method *ChangeMedia* can be verified because it re-establishes the invariants of all *TapeArchiver* instances before the end of the **expose** block. Other methods such as *Eject* violate delegate invariants without re-establishing them. Such methods can only be verified under the requirement that the target does not have any dependent *TapeArchiver* delegate instances. To ensure this requirement, we have to add the following requires clause to *Eject*:

**requires**  $(\forall d \bullet d.target = \mathbf{this} \Rightarrow \neg d.inv <: \mathit{TapeArchiver})$ ;

This strong requires clause enables the verification of the method body, but is difficult to be satisfied by callers of *Eject*. In particular, if in some execution state a *TapeArchiver* instance  $d$  refers to an object  $o$  then  $o.Eject$  cannot be called in any subsequent execution state, even if  $d$  is not used anymore. This is because program verifiers typically do not model garbage collection, which means that formally  $d$  will never be deleted.

To support methods that violate certain delegate invariants, we provide a statement **disable**( $D$  for  $o$ ), which disables all delegate instances with target object  $o$  that are valid for a delegate type  $D$ . A delegate instance  $d$  is disabled by exposing it—such that its invariant does not have to be maintained anymore—and by making  $d$  un-owned. The change of ownership is necessary to be able to un-expose  $d$ 's owner—recall that P2 requires owned objects to be fully valid when the owner is valid. We do not provide a statement to re-enable a delegate instance since one can simply create a new instance.

It is generally necessary to execute **disable**(*TapeArchiver* for  $o$ ) before each call to  $o.Eject$  to establish the above requires clause. This might seem tedious, but is only necessary if a delegate declares an invariant for class  $T$  and  $T$  contains methods that break this invariant. Such delegates are error-prone and we consider the overhead of adding **disable** statements acceptable in these rare cases.

## 4 Technical Treatment

In this section, we present the technical treatment of our methodology. We define precisely which delegate invariants are admissible, formalize the proof rules, and prove that our methodology is sound.

## 4.1 Admissible Delegate Invariants

Our methodology permits the invariant of a delegate instance  $d$  to depend on the field  $d.target$  as well as the state of  $d$ 's target object and all objects (transitively) owned by the target. However, to make the presentation and, in particular, the soundness proof self-contained; we use a slightly more restrictive definition of admissible delegates invariants here, which does not permit dependencies on the objects owned by the target.

**Definition 1 (Admissible Delegate Invariant).** *An invariant for  $T$  declared in or inherited by a delegate type  $D$  is admissible if and only if: (i) its sub-expressions type-check under the assumption that  $target$  is of type  $T$ ; (ii) each of the field-access expressions has the form  $\mathbf{this.target}$  or  $\mathbf{this.target.f}$ , where  $f$  is declared in  $T$  or a superclass of  $T$  and  $f$  is not one of the pre-defined fields  $inv$  or  $owner$ ; (iii)  $D$  is mentioned in the dependent clause of  $T$ .*

## 4.2 Proof Rules

We define the proof rules of our methodology by translating the relevant statements into pseudo code, which makes the assertions and state changes explicit.

**Delegate Instantiation.** The instantiation of a delegate  $D$  with an instance method  $o.m$  (Fig. 5) checks that the target  $o$  is non-null and valid for each class  $T$  that  $D$  depends on. The latter assertion is necessary to maintain program invariant P4. Note that the visibility requirement (Sec. 3.2) allows us to determine each dependee  $T$  modularly by inspecting the invariants of  $D$  and  $D$ 's supertypes. Next, a fresh object  $d$  is allocated, its  $target$  field is set to  $o$ , and its  $owner$  field is set to  $o.owner$  to make the delegate instance and the target peers. New delegate instances start off being fully valid. To maintain program invariant P1, we assert the invariant of  $D$  and  $D$ 's supertypes before setting the  $inv$  field of the new instance to  $D$ . Finally, we check that the specification of  $m$  refines the specification of  $D$ .  $Pre_D(d, p, h)$  and  $Pre_m(o, p, h)$  denote the requires clauses of  $D$  and  $m$ , respectively. The ensures clauses are denoted by  $Post_D(d, p, r, h, h')$  and  $Post_m(o, p, r, h, h')$ , where  $d$  and  $o$  are the targets of  $D$  and  $m$ , respectively,  $p$  is the (only) explicit parameter,  $r$  is the result,  $h$  is the heap of the prestate, and  $h'$  is the heap of the poststate.

---

```

d := new D(o.m) ≡
  assert o ≠ null ;
  #foreach T such that D ∈ dependents(T) { assert o.inv <: T ; }
  d := new D ;
  d.target := o ; d.owner := o.owner ;
  #foreach E such that D <: E { assert Inv_E(d) ; }
  d.inv := D ;
  assert (∀ p, h • Pre_D(d, p, h) ⇒ Pre_m(o, p, h)) ;
  assert (∀ p, r, h, h' • Pre_D(d, p, h) ∧ Post_m(o, p, r, h, h') ⇒ Post_D(d, p, r, h, h')) ;

```

**Fig. 5.** Pseudo code for delegate instantiation

---

**Delegate Invocation.** Delegate invocations are handled just like method calls (Fig. 6). The invoker must ensure that the requires clause holds before the invocation and may assume the ensures clause after the invocation. This reasoning is justified by the refinement relationship between the specifications of the delegate and the underlying method, which is checked when the delegate is instantiated. The `havoc` statement assigns arbitrary values to the variables for the current heap  $\mathbb{H}$  and the result of the invocation  $v$ . This is necessary to make the verifier “forget” any prior knowledge about the variables that are potentially modified by the delegate invocation. Before the `havoc`, the heap of the prestate is saved since the ensures clause may refer to it.

---

```

 $v := d(p) \equiv$ 
  assert  $d \neq \text{null} \wedge \text{Pre}_D(d, p, \mathbb{H})$  ;
   $h := \mathbb{H}$  ; havoc  $\mathbb{H}, v$  ;
  assume  $\text{Post}_D(d, p, v, h, \mathbb{H})$  ;

```

**Fig. 6.** Pseudo code for delegate invocation.  $D$  is the static type of the delegate instance  $d$ .

---

**Expose.** Our methodology extends the `expose` statement of the Boogie methodology to automatically expose and un-expose dependent delegates (Fig. 7). We first discuss the parts we adopted from the Boogie methodology and then explain the extensions.

The `expose` statement of the Boogie methodology implements a protocol that guarantees that owners are exposed before the objects they own, and that an object is exposed for a subclass before it is exposed for the superclass. Besides fields of  $o$  declared in  $T$ , the protocol allows  $\text{Inv}_T(o)$  to depend on fields of objects owned by  $\langle o, T \rangle$  and on fields of  $o$  that are inherited from a superclass of  $T$ . In both cases, the protocol ensures that  $o$  is exposed for  $T$  before  $\text{Inv}_T(o)$  is potentially violated by a field update.

In the pseudo code for `expose` ( $o$  for  $T$ ), this protocol is implemented as follows. First, we assert that  $o$  is non-null and valid for  $T$ , that is, has already been exposed for  $T$ 's subclasses. Next, we assert that  $o$ 's owner is sufficiently exposed. Finally,  $o$  is exposed by setting  $o.\text{inv}$  to  $T$ 's direct superclass, `super`( $T$ ). After the body of the `expose` block is executed, we assert that all objects owned by  $o$  in the type frame of  $T$  are fully valid. Then we un-expose  $o$  by setting  $o.\text{inv}$  back to  $T$ . This update is guarded by an assertion that  $\text{Inv}_T(o)$  holds (to maintain program invariant P1).

The automatic exposing of dependent delegates is done as follows. For each delegate type  $D$  in the dependent clause of  $T$ , we determine the set  $\text{Dep}_D$  of all delegate instances whose target is  $o$  and that are valid for  $D$ . These are the delegate instances whose invariants are potentially violated by assigning to fields of  $o$  declared in class  $T$ . We expose each of these delegate instances by setting its `inv` field to `object`. The automatic un-exposing is done analogously. For each delegate instance that has been previously exposed, that is, is in one of the sets  $\text{Dep}_D$ , we assert its delegate invariant and set its `inv` field back to  $D$ . While non-delegate objects are exposed for one class at a time, the automatic exposes for a delegate instance  $d$  goes in one step from  $D$  to `object` and back. Therefore, it is necessary to assert all invariants that are declared in  $D$  and  $D$ 's supertypes when  $d$  is un-exposed.

---

```

expose (o for T) { S } ≡
  assert o ≠ null ∧ o.inv = T ;
  assert o.owner.obj ≠ null ⇒ ¬o.owner.obj.inv <: o.owner.typ ;
  #foreach D ∈ dependents(T) {
    let DepD := { d | d.target = o ∧ d.inv = D } ;
    foreach d ∈ DepD { d.inv := object ; }
  }
  o.inv := super(T) ;

  S ;

  assert (∀ object p • p.owner = ⟨o, T⟩ ⇒ p.inv = typeof(p)) ;
  assert InvT(o) ;
  o.inv := T ;
  #foreach D ∈ dependents(T) {
    foreach d ∈ DepD {
      #foreach E such that D <: E { assert InvE(d) ; }
      d.inv := D ;
    }
  }

```

**Fig. 7.** Pseudo code for **expose**. The extensions for delegates are highlighted by a shaded background.

---

An important virtue of our methodology is that it causes no verification overhead for programs that do not use delegates, and very little overhead for programs whose delegates do not have invariants. In particular, the **#foreach** loops in Fig. 7 can be unrolled statically by a compiler using the dependent clause of class *T*. If *T* does not have a dependent clause, the pseudo code for **expose** (*o* for *T*) is identical to the Boogie methodology without delegates.

**Disabling Delegates.** As explained in Sec. 3, a delegate instance is disabled by exposing it and by making it un-owned, that is, setting its owner object to **null**. The statement **disable**(*D* for *o*) (Fig. 8) disables all delegate instances that are attached to a target object *o* and valid for delegate type *D*. Since disabling a delegate instance *d* changes its state, *d*'s owner has to be sufficiently exposed.

---

```

disable(D for o) ≡
  assert o ≠ null ;
  assert o.owner.obj ≠ null ⇒ ¬o.owner.obj.inv <: o.owner.typ ;
  foreach d such that d.target = o ∧ d.inv <: D {
    d.inv := object ; d.owner := ⟨null, -⟩ ;
  }

```

**Fig. 8.** Pseudo code for **disable**

---

### 4.3 Soundness

Soundness of our methodology means that it is justified to assume program invariants P1–P4 when proving the assertions introduced by the methodology. In the following, we sketch the proofs of these program invariants. The proofs run by induction over the sequence of states of an execution of a program that is well-formed: that is, syntactically correct, type correct, and all invariants are admissible. The induction base is trivial since there are no allocated objects in the initial program state. For the induction step, we assume that the program invariant holds before the next statement  $s$  to be executed, and show that  $s$  preserves it.

**Program Invariant P1.** For non-delegate objects, P1 is guaranteed by the Boogie methodology. The soundness proof [11] remains valid because creation, exposing, and modification (that is, disabling) of delegate instances are guarded by the same proof obligations as the corresponding operations on non-delegate objects.

We proceed by proving P1 for delegate instances. That is, we show that statement  $s$  preserves the implication  $o.inv <: T \Rightarrow Inv_T(o)$  for any delegate instance  $o$  and any type  $T$ . We consider all cases where  $s$  manipulates the state of an object; we omit all other cases for brevity.

*Delegate Instantiation.* Instantiation of a delegate  $D$  does not change the state of existing delegate instances. It remains to prove that the implication is established if  $o$  is the new delegate instance. After the instantiation, we have  $o.inv = D$ . The pseudo code asserts  $Inv_E(o)$  for all  $E$  where  $D <: E$ , in particular, for  $E = T$ . Therefore, the implication holds.

*Expose.* The statement **expose** ( $x$  for  $S$ ) changes the  $inv$  field of  $x$  as well as each delegate instance  $d \in Dep_D$ , but nothing else. Since admissible delegate invariants do not refer to  $inv$  fields (Def. 1),  $Inv_T(o)$  cannot be affected by these state changes. It remains to show that the implication is preserved if  $o = x$  or  $o = d$ .

In both cases, the first update of  $inv$  preserves the implication by making its left-hand side stronger. By the induction hypothesis, the body of the **expose** block preserves the implication. Setting  $x.inv$  from **super**( $S$ ) back to  $S$  affects the implication only if  $T = S$ . However, since  $Inv_S(x)$  is asserted before the update of  $x.inv$ , the implication is preserved. Setting  $d.inv$  from **object** back to  $D$  affects the implication only if  $T = E$  for some supertype  $E$  of  $D$ . Again, since  $Inv_E(d)$  is asserted for all such  $E$  before the update of  $d.inv$ , the implication is preserved.

*Field Update.* Consider an update  $x.f := e$ , where  $f$  is declared in a class  $F$ . We may assume that  $f$  is different from  $inv$  and  $target$ , which must not be directly assigned to. According to Def. 1, an invariant of  $T$  that mentions  $f$  must be an invariant for  $G$ , where  $G <: F$  and  $T$  is mentioned in  $G$ 's dependent clause ( $T \in dependents(G)$ ). The update of  $x.f$  is guarded by the assertion  $\neg x.inv <: F$ . By  $G <: F$ , we get  $\neg x.inv <: G$ . For  $o$ ,  $T$ 's invariant for  $G$  may depend on  $x.f$  only if  $o.target = x$ . Moreover, we may assume **typeof**( $o$ )  $<: G$ , otherwise the desugared invariant holds trivially. Therefore, by contraposition on P4, this implies  $\neg o.inv <: T$ , and, thus, the left-hand side of the implication is *false*.

*Disable.* The statement `disable(D for x)` changes the *inv* and *owner* fields of each delegate instance where  $d.\text{inv} <: D$  and  $d.\text{target} = x$ , but nothing else. Since admissible delegate invariants do not refer to *inv* and *owner* (Def. [1](#)),  $\text{Inv}_T(o)$  cannot be affected by these modifications. It remains to show that the implication is preserved if  $o = d$ . This is trivially the case since the update of  $d.\text{inv}$  makes the left-hand side of the implication stronger.  $\square$

**Program Invariant P2.** This program invariant is a consequence of the protocol that owners are exposed before the objects they own, the block structure of `expose`, and the fact that newly created objects and delegate instances are fully valid when the constructor terminates. Since these arguments are identical for objects and delegate instances, the soundness proof from the Boogie methodology [\[11\]](#) remains valid.  $\square$

**Program Invariant P3.** The *owner* field of an object is modified when an object or delegate instance is created and when a delegate instance is disabled. The proof is trivial for all three cases: (1) Newly created objects do not have any delegate instances attached. Therefore, the property holds trivially. (2) When a delegate is instantiated, its owner is set to the owner of its target, which establishes the property for the new instance. (3) When a delegate instance  $d$  is disabled,  $d.\text{inv}$  is set to `object`. Therefore, the left-hand side of the implication becomes *false*.  $\square$

**Program Invariant P4.** We prove that statement  $s$  preserves the implication of P4 for any object  $o$ , delegate instance  $d$ , class  $T$ , and delegate type  $D$ . It suffices to consider all statements  $s$  that modify the *inv* field.

*Object Creation.* Newly created objects do not have any delegate instances attached. Therefore, the implication trivially holds.

*Delegate Instantiation.* The instantiation  $e := \text{new } E(x.m)$  does not change the state of existing objects. We have to show that the implication is preserved for  $e = d$ . We may assume  $d.\text{inv} <: D$ ,  $D \in \text{dependents}(T)$ , and  $d.\text{target} = o$ , otherwise the left-hand side of the implication is *false*. The instantiation establishes  $e.\text{inv} = E$ . By  $d.\text{inv} <: D$  and  $e = d$ , we get  $E <: D$ .

From  $D \in \text{dependents}(T)$  we conclude that  $D$  is mentioned in a dependent clause of  $T$ . By the visibility requirement (Sec. [3.2](#)),  $D$  must declare or inherit an invariant for  $T$ . Since  $E <: D$  and invariants are inherited,  $E$  also declares or inherits this invariant. By Def. [1](#) we know that  $E$  must also be mentioned in a dependent clause of  $T$ , that is, we have  $E \in \text{dependents}(T)$ . The pseudo code for delegate instantiation (Fig. [5](#)) asserts  $x.\text{inv} <: T$ . Since  $x = d.\text{target} = o$ , this implies the right-hand side of the implication.

*Expose.* The statement `expose(x for S)` changes the *inv* field of  $x$  as well as each delegate instance  $e$  in one of the  $\text{Dep}_A$ , but nothing else. This case is trivial if  $o \neq x$  because in that case neither  $o.\text{inv}$  nor  $d.\text{inv}$  is changed.

For  $o = x$ , setting  $d.\text{inv}$  to `object` makes the left-hand side of the implication stronger and, therefore, preserves the implication. Setting  $o.\text{inv}$  to `super(S)` affects the implication only if  $S = T$ . The proof of this case is very similar to the proof for

delegate instantiation. Again, we may assume  $d.inv <: D$ ,  $D \in \text{dependents}(T)$ , and  $d.target = o$ . Let  $E = d.inv$ ; consequently, we have  $E <: D$ . Like for instantiation, we conclude  $E \in \text{dependents}(T)$  and, by  $S = T$ ,  $E \in \text{dependents}(S)$ . Since  $d.target = o$  and  $o = x$ , we know  $d \in \text{Dep}_E$  such that  $d.inv$  is set to **object**. Since  $D$  is a delegate type, this makes the left-hand side of the implication *false*.

By the induction hypothesis, the body of the **expose** block preserves the implication. The un-exposing after the body precisely un-does the modifications of the *inv* fields performed before the body. Therefore, it preserves the implication.

*Disable.* The statement **disable**( $E$  for  $x$ ) modifies the *inv* field of each delegate instance where  $e.inv <: E$  and  $e.target = x$ , but nothing else. The setting of  $e.inv$  to **object** only makes the left-hand side of the implication stronger.  $\square$

## 5 Related Work

Eiffel agents [7] are similar to C# delegates, but more general since they allow the programmer to decide for each parameter, including the target, whether the actual argument is provided when the agent is instantiated (closed parameter) or when it is invoked (open parameter). Adapting our methodology to agents would require agent invariants that depend on all closed parameter objects. The corresponding visibility requirement might be too restrictive for certain applications of agents.

The work closest to ours is the version of the Boogie methodology described by Leino and Müller [11]. Besides the ownership-based object invariants that we also use in this paper, their work also supports visibility-based object invariants. Like our delegate invariants, visibility-based invariants may depend on fields of peers, provided that a visibility requirement is met. However, Leino and Müller’s work requires programs to explicitly expose all objects whose visibility-based invariants are potentially affected by a field update. In general, without explicit references to the dependent peers, this obligation is hard to live up to. Our methodology exposes dependent delegate instances automatically when their target is exposed. Like the Boogie methodology, our work supports ownership transfer, but we omitted details due to space limitations.

The friendship methodology by Barnett and Naumann [3] simplifies the verification of visibility-based invariants by introducing ghost state to keep track of all dependent invariants of an object. This ghost state facilitates the exposing of dependent objects. The friendship methodology can handle implementations of function objects such as the Command pattern. Whereas the friendship methodology is very general, our methodology exploits the special syntactic structure of delegates to expose dependent delegate instances automatically, which removes the need to explicitly keep track of dependent invariants and, thus, reduces the annotation overhead.

Jacobs’s version of Spec#, SpecLeuven [9], permits sound reasoning about delegates. Delegate instances own their target objects, which prevents these objects from being owned by other objects and, in particular, from being used in other delegate instances. Our solution does not impose this restriction.

Leino and Schulte [12] use history constraints to verify object invariants that are neither ownership-based nor visibility-based. If a history constraint guarantees that



the state of an object  $o$  only evolves in ways that does not affect a dependent object invariant then there is no need to expose the dependent object before modifying  $o$ . We expect this approach to be a useful complement of our methodology, but not all targets have strong history constraints. For instance, Leino and Schulte's methodology cannot handle our *TapeDrive* example, because *IsLoaded* is not a monotonic property.

Visible state semantics require invariants to hold in the pre- and post-states of all method executions. When invariants are allowed to depend on several objects such as delegate invariants, one needs a way of determining which invariants are potentially affected by a field update. These are exactly the invariants that our methodology exposes automatically when a target is exposed. Our methodology can be adapted to a visible state semantics using visibility-based invariants as described by Müller *et al.* [13].

The work by Börger *et al.* [4] has captured the semantics of delegates in an ASM model. This work has been fundamental in understanding the delegate construct. However, it is not suggestive how to use an ASM model to verify programs modularly.

## 6 Conclusions

We have presented a methodology for specifying and verifying C# delegates. Our methodology uses delegate invariants to express properties of the target object and allows one to reason about delegate invariants in a sound and modular way.

Our methodology requires significantly less specification and verification overhead than other techniques that could handle the Command pattern. This simplification is possible because delegates are essentially a stylized Command pattern, for which we can build special support into the verification methodology. We expect that similar methodologies can be developed for other design patterns, provided that the components of a design pattern are marked as such or that the idiom is supported by a special language construct.

Our methodology solves one of the two challenges related to function objects reported earlier [10], namely how to verify invocations of function objects. The other challenge is how to specify and verify invokers of function objects. We plan to address this challenge in future work. We also plan to implement our methodology into the Spec# programming system.

C# 2.0 delegates are more expressive than the delegates of C# 1.0, which we considered in this paper. For instance, C# 2.0 provides anonymous delegates, which may refer to local variables of the method body enclosing their declaration. Extending our methodology to C# 2.0 delegates is future work.

**Acknowledgments.** We are grateful to Rustan Leino for very helpful discussions and suggestions, in particular, his idea to use delegate subtypes. Thanks also to Bart Jacobs for his comments. Müller's work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project during his stay at ETH Zurich.



## References

1. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. *JOT* 3(6) (2004)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004*. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Barnett, M., Naumann, D.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
4. Börger, E., Fruja, N.G., Gervasi, V., Stärk, R.F.: A high-level modular definition of the semantics of C#. *Theoretical Computer Science* 336(2-3), 235–284 (2005)
5. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: *ICSE*, pp. 258–267. IEEE Computer Society Press, Los Alamitos (1996)
6. C# language specification. ECMA Standard 334 (June 2005)
7. Eiffel analysis, design and programming language. ECMA Standard 367 (June 2005)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley, Reading (1995)
9. Jacobs, B.: *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven (2007)
10. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing* 19(2), 159–189 (2007)
11. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) *ECOOP 2004*. LNCS, vol. 3086, pp. 491–516. Springer, Heidelberg (2004)
12. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 80–94. Springer, Heidelberg (2007)
13. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. *Science of Computer Programming* 62, 253–286 (2006)

# On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York,  
Heslington, York, YO10 5DD, UK  
{dkolovos, paige, fiona}@cs.york.ac.uk

**Abstract.** The Object Constraint Language (OCL) can be used to capture structural constraints in the context of the abstract syntax of modelling languages (metamodels) defined in the MOF metamodeling architecture. While the expression language of OCL has been revised and updated a number of times since its inception, the constructs used for capturing constraints (invariants) have remained unchanged. In this paper we argue that the abstract and concrete syntax of OCL invariants should also be updated to address a number of shortcomings and render OCL more usable in a contemporary modelling environment. To support our arguments we have implemented the proposed extensions in the prototype Epsilon Validation Language (EVL). To demonstrate the benefits delivered, we present and discuss a concrete example.

## 1 Introduction

The Meta Object Facility (MOF) [1] is a self-defining language for specifying the abstract syntax of modelling languages such as UML. MOF enables capturing the concepts of a language and, to an extent, also expresses how they can be legitimately combined to form valid models. By design, MOF can only express a limited range of structural constraints, mainly with respect to containment and type conformance [1]. For more complex structural constraints that MOF cannot capture by itself, the Object Constraint Language (OCL) [2] is used. In OCL, structural constraints are captured in the form of *invariants* attached to MOF meta-classes.

To enable users to specify precise and concise constraints, OCL provides an expression language with powerful model querying and navigation facilities. Since its inception, the expression language and the type system of OCL have undergone several revisions and as a result they have been radically improved. By contrast, the syntax and semantics of *invariants* have remained almost unchanged since the early versions of the language. In this paper, we identify some of the shortcomings of the syntax and semantics of OCL *invariants* for capturing structural constraints and stress the need for evolving the standard. Our views on OCL modernization range from adding simple features such as support for detailed user feedback, to advanced features such as support for semi-automatically repairing inconsistent model elements.

The focus in this paper is on improving support for OCL (and OCL-like languages), in terms of supporting richer structures for encoding constraints, checking constraints, and obtaining feedback from the checking process. As such, we work in the domain

of models (e.g., UML, MOF, Eclipse EMF). However, the principles and limitations we identify in Section 2 and the lessons learned and improvements made to OCL, will also be useful in assessing and evolving constraint-based techniques in domains other than modelling, particularly analysis of object-oriented programs and formal models of object-oriented programs. In particular, the techniques illustrated in this paper could be applied to formal languages such as JML [3], Spec# [4], and Eiffel [5].

The rest of the paper is organized as follows. In Section 2 we identify shortcomings of OCL for capturing structural constraints in modelling languages. In Section 3 we present a coherent set of extensions that address the identified issues and present an implementation of those extensions in the prototype *Epsilon Validation Language (EVL)*. To demonstrate the usefulness and practicality of our approach, in Section 4 we present a working case study where we compare OCL with EVL on a common scenario. In Section 5, we discuss related work and in Section 6 we conclude and discuss directions for further research on the subject.

## 2 Shortcomings of OCL for Capturing Structural Constraints

In OCL, structural constraints are captured in the form of *invariants*. Each invariant is defined in the context of a meta-class of the metamodel and specifies a name and a body. The body is an OCL expression that must evaluate to a Boolean result, indicating whether an instance of the meta-class satisfies the invariant or not. Execution-wise, the body of each invariant is evaluated for each instance of the meta-class and the results are stored in a set of <Element, Invariant, Boolean> triplets. Each triplet captures the Boolean result of the evaluation an Invariant on a qualified Element. An exemplar OCL invariant for UML, requiring that abstract operations only belong to abstract classes, is shown in Listing 1.1.

**Listing 1.1.** OCL constraint on UML operations

```
context Operation
  inv AbstractOperationInAbstractClassOnly :
    self.isAbstract implies self.owner.isAbstract
```

While in its current version OCL enables users to capture particularly complex invariants, it also demonstrates a number of shortcomings, as follows.

### 2.1 Poor Support for User Feedback

OCL does not support specifying meaningful messages that can be reported to the user in case an invariant is not satisfied for certain elements. Therefore, feedback to the user is limited to the name of the invariant and the instance(s) for which it failed. Weak support for proper feedback messages implies that the end users must be familiar with OCL so that they can comprehend the meaning of the failed invariant and locate the exact reason for the failure. This is a significant shortcoming as in practice only a very small number of end users are familiar with OCL.

## 2.2 No Support for Warnings/Critiques

Contemporary software development environments typically produce two types of feedback when checking artefacts for consistency and correctness: errors and warnings. Errors indicate critical deficiencies that contradict basic principles and invalidate the developed artefacts. By contrast, warnings (or critiques) indicate non-critical issues that should nevertheless be addressed by the user. To enable users to address warnings in a priority-based manner, they are typically categorized into three levels of importance: High, Medium and Low (although other classifications are also possible).

By contrast, in OCL there is no such distinction between errors and warnings and consequently all reported issues are considered to be errors. This adds an additional burden to identifying and prioritizing issues of major importance, particularly within an extensive set of unsatisfied invariants.

## 2.3 No Support for Dependent Constraints

Each OCL invariant is a self-contained unit that does not depend on other invariants. However, there are cases where this design decision is particularly restrictive. For instance consider the invariants I1 and I2 displayed in Listing 1.2. Both I1 and I2 are applicable on UML classes with I1 requiring that: *the name of a class must not be empty* and I2 requiring that: *the name of a class must start with a capital letter*. In the case of those two invariants, if I1 is not satisfied for a particular UML class, evaluating I2 on that class would be meaningless. In fact it would be worse than meaningless since it would consume time to evaluate and would also produce an extraneous error message to the user. In practice, to avoid the extraneous message, I2 needs to replicate the body of I1 using an *if* expression (lines 2 and 5).

Listing 1.2. Conceptually related OCL constraints

```

1 context Class
2   inv I1 : self.name.size() > 0
3
4   inv I2 :
5     if self.name.size > 0 then
6       self.name.substring(0,1) =
7       self.name.substring(0,1).toUpper()
8     else
9       true
10    endif

```

## 2.4 Limited Flexibility in Context Definition

As already discussed, in OCL invariants are defined in the context of meta-classes. While this achieves a reasonable partitioning of the model element space, there are cases where more fine-grained partitioning is required. For instance, consider the following scenario. Let  $IA_{1..N}$ ,  $IB_{1..M}$  be invariants applying to classes that are stereotyped as  $\langle\langle A \rangle\rangle$  and  $\langle\langle B \rangle\rangle$  respectively. Since OCL only supports partitioning the model element space using meta-classes, all  $IA_{1..N}$ ,  $IB_{1..M}$  must appear under the same context

(i.e. *Class*). Moreover, each invariant must explicitly define that it addresses the one or the other conceptual sub-partition. Therefore, each of  $IA_{1..N}$  must limit its scope initially (using the *self.isA* expression) and then express the real body. In our example the simplest way to achieve this would be by combining a scope-limiting expression with the real invariant body using the *implies* clause as demonstrated in Listing 1.3.

**Listing 1.3.** Demonstration of OCL constraints with duplication

```
context Class
  inv I1 : self.isA implies <real-invariant-body>
  inv I2 : self.isA implies <real-invariant-body>
  ...
  inv IN : self.isA implies <real-invariant-body>

  def isA :
    let isA : Boolean =
      self.stereotype->exists(s | s.name = 'A')
```

Furthermore, if the real body of the invariant needs to assume that *self* is stereotyped with <<A>>, this technique is not applicable because OCL does not support lazy evaluation of boolean clauses [2] and therefore although the first part of the expression (*self.isA*) may fail for some instances, the second part will still be evaluated thus producing runtime errors. In this case, an *if* expression must be used, further complicating the specified invariants.

## 2.5 No Support for Repairing Inconsistencies

While OCL can be used for detecting inconsistencies, it provides no means for repairing them. The reason is that OCL has been designed as a side-effect free language and therefore lacks constructs for modifying models. Nevertheless, there are many cases where inconsistencies are trivial to resolve and users can benefit from semi-automatic repairing facilities.

This need has been long recognized in the related field of code development tools. In such tools, errors are not only identified but also context-aware actions are proposed to the user for automatically repairing them. To our view, this feature significantly increases the usability of such tools and consequently enhances users' productivity.

Following this discussion on the shortcomings of OCL for capturing structural constraints in modelling languages, in Section 3 we propose an extended version of OCL that overcomes them.

## 3 Extending OCL: The Epsilon Validation Language (EVL)

To address the issues discussed in Section 2, in this section we propose a set of extensions to the abstract and concrete syntaxes of OCL. To experiment with the proposed extensions in practice, we have implemented them in the context of a prototype validation language with tool-support: the Epsilon Validation Language (EVL). Here we should state that EVL is not a rival to OCL; instead we view EVL as a flexible prototype on which we can easily implement and evaluate novel approaches to specifying

and managing model constraints. The purpose of this paper is not to promote this new language but to demonstrate useful ideas, using a working prototype, that can hopefully contribute to the evolution of OCL and similar languages. From this perspective, in this section we discuss the abstract syntax of EVL and how each additional or extended construct in it contributes to addressing the shortcomings identified in Section 2.

### 3.1 Infrastructure

EVL has been designed atop the Epsilon platform [6], and therefore instead of pure OCL, it uses the OCL-based Epsilon Object Language (EOL) [7] as a query and navigation language. Using EOL instead of pure OCL delivers several practical advantages. First, EOL supports statement sequencing, thus allowing users to disentangle complex queries into sequences of simpler queries. From our experience, this enhances readability and maintainability and also facilitates a smoother transition to the field of model-management for developers with background in object oriented or procedural programming. For this community, the purely declarative style of OCL, which is significantly influenced by the style of functional programming languages [8], has been shown to be challenging to use. Nevertheless, the design decision to support statement sequencing does not affect existing OCL users who can still specify purely declarative queries.

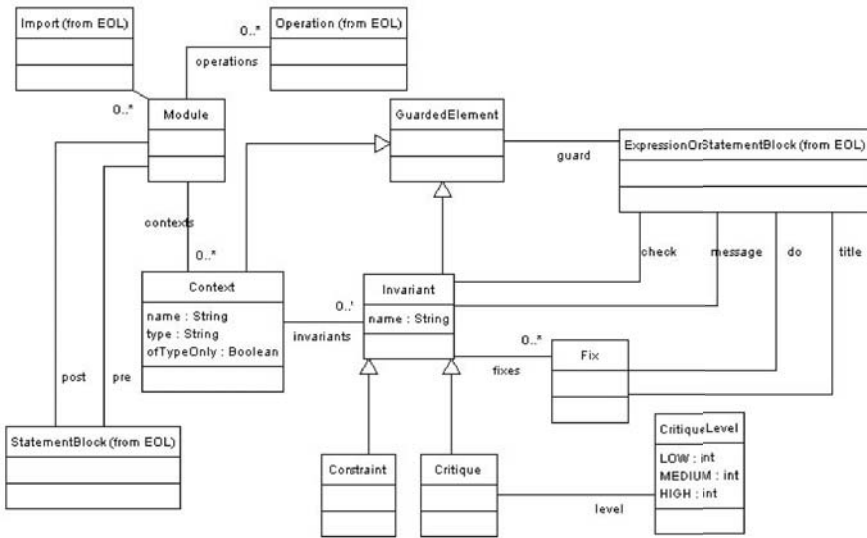
A further advantage is that since EOL can access more than one model simultaneously [7], EVL can be also used to express inter-model constraints. However, this feature is considered to be out of the scope of this paper and is not further discussed here. EOL also provides model modification features that are used in the sequel to address the issue of semi-automatic repairing of inconsistencies. Finally, EOL supports user input and output operations. This allows users to produce diagnostic messages to examine the flow of control, which is one of the most widely-employed techniques for understanding and debugging purposes.

### 3.2 Abstract Syntax of EVL

In this section we discuss the main concepts of the abstract syntax of EVL, which is also presented graphically in Figure 1.

*Context.* A context specifies the type (or kind) of instances on which the contained invariants will be evaluated. Each context can optionally define a guard which limits its applicability to a narrower subset of instances of its specified type. Thus, if the guard fails for a specific instance of the metaclass, none of its contained invariants are evaluated.

*Invariant.* As with plain OCL, each EVL invariant defines a name and a body (check). In this extended version, it can additionally define a guard which further limits its applicability to a subset of the instances of the metaclass defined by the embracing *context*. To achieve the requirement for detailed user feedback (Section 2.1), each invariant can optionally define a message as an *ExpressionOrStatementBlock* that should return a String. To support semi-automatically fixing elements on which invariants have failed (Section 2.5), an invariant can optionally define a number of *fixes*. Finally, as displayed



**Fig. 1.** Abstract Syntax of EVL

in Figure 1. *Invariant* is an abstract class that is used as a super-class for the specific types *Constraint* and *Critique*. This is to address the issue of separation of errors and warnings/critiques (Section 2.2).

*Guard*. Guards are used to limit the applicability of invariants (Section 2.4). This can be achieved at two levels. At the *Context* level it limits the applicability of all invariants of the context and at the *Invariant* level it limits the applicability of a specific invariant.

*Fix*. A fix defines a title using an *ExpressionOrStatementBlock* instead of a static String to allow users specify context-aware titles (e.g. *Rename class customer to Customer* instead of the generic *Convert first letter to upper-case*). Moreover, it defines a *do* part where the fixing functionality can be defined using EOL. The developer is responsible for ensuring that the actions contained in the *fix* actually repair the inconsistency.

*Constraint*. *Constraints* in this extended metamodel are used to capture critical errors that invalidate the model. As discussed above, *Constraint* is a sub-class of *Invariant* and therefore inherits all its features.

*Critique*. Unlike *Constraints*, *Critiques* are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model. This separation addresses the issue raised in Section 2.2. Moreover, to enable users define different levels of importance in critiques, the *CritiqueLevel* enumeration supports a 3-level classification. Fixed-level classification has been preferred in EVL over infinite level classification (e.g. using Integer levels) since it is more common in development tools and easier to visualize.

*Pre and Post.* An EVL module can define a number of named *pre* and a *post* blocks that contain EOL statements which are executed before and after evaluating the invariants respectively.

### 3.2.1 Additional built-in operations

As discussed in Section 2.3, it is often the case that invariants conceptually depend on each other. To allow users capture such dependencies, the *satisfies(invariant:String): Boolean*, *satisfiesAll(invariants:Sequence(String))* and *satisfiesOne(invariants:Sequence(String))* have been added to EVL. Using these operations, an invariant can specify in its *guard* other invariants which need to be satisfied for it to be meaningful to evaluate.

### 3.2.2 Concepts reused from EOL

As EVL has been built atop EOL, it reuses the following constructs from the base-language:

*ExpressionOrStatementBlock.* There are cases where users needs to calculate a value (e.g. in the *message* of an *invariant*, in the *guard* of a *context* etc). When the value can be calculated declaratively, this is preferred. However, for cases in which calculating the value requires complex computations, users can use an EOL statement block and use a *ReturnStatement* to return the calculated value to the caller.

*StatementBlock.* A statement block is a sequence of EOL statements that can optionally include one or more *ReturnStatements* to return a calculated value to its caller.

## 3.3 Execution Semantics of EVL

The additional concepts EVL provides also affect its execution semantics. Currently, an EVL module can only be executed in batch-mode (all invariants against all instances). In the future we plan to investigate how the additional structures that EVL provides affect approaches to incremental consistency checking such as those presented in [9][10]. In this section we outline the execution semantics of the language in batch-mode.

*Phase 1.* Before any invariant is evaluated, the *pre* section of the module is executed.

*Phase 2.* For each *context*, the instances of the meta-class it defines are collected. For each instance, the *guard* of the *context* is evaluated. If the *guard* is satisfied then for each invariant contained in the context the invariant's *guard* is also evaluated. If the *guard* of the invariant is satisfied, the *body* of the invariant is evaluated. In case the *body* evaluates to *false*, the *message* part of the rule is evaluated and the produced message is added along with the instance, the invariant and the available *fixes* to the *ValidationTrace*.

The execution order of an EVL module follows a top-down depth-first scheme that respects the order in which the *contexts* and *invariants* appear in the module. However, the execution order can change in case one of the *satisfies*, *satisfiesOne*, *satisfiesAll* built-in operations are called. In this case, if the required invariants have not been evaluated yet for the instances on which the operations are invoked, the engine will evaluate them and then resume with the normal execution order. By using a caching mechanism, each invariant is executed for a specific instance at most once.



*Phase 3.* In this phase, the validation trace is examined for unsatisfied constraints and the user is presented with the message each one has produced. The user can then select one or more of the available *fixes* to be performed.

*Phase 4.* When the user has performed all the necessary *fixes*, the *post* section of the module is executed. There, the user can perform tasks such as serializing the validation trace or producing a summary of the validation process results.

### 3.4 Tool Support

As proof of concept, we have implemented a prototype EVL execution engine with respective development tools for Eclipse [11]. Moreover, to increase practicality for validating UML models, we have integrated the EVL execution engine with the open-source ArgoUML modelling tool [12].

**Eclipse Development Tools.** Using the Eclipse development tools, developers can compose EVL specifications using a dedicated editor that provides syntax highlighting and on-site reporting of syntax and run-time errors (Figure 2). With regard to modelling technologies, users can evaluate EVL specifications on EMF [13], MDR [14] and XML models. Following the evaluation process, messages generated by failed invariants are reported in the *Validation* view. For each one, the available fixes are accessible by right clicking on the respective error message, so that users can select and perform the appropriate ones. The EVL Eclipse Development tools have been released as part of the Epsilon development Tools available at the Epsilon GMT website [15].

**Integrating with ArgoUML.** As discussed above, using the EVL Eclipse Development Tools enables users to validate and repair models of diverse technologies. However, although EVL can produce meaningful messages (by contrast to pure OCL), there are cases where users may need to thoroughly inspect their models before deciding how to resolve an inconsistency. To provide better support for UML models, we have also integrated EVL with the open-source ArgoUML tool to enable users validate and fix their UML 1.5 models without leaving their modelling tool. Integrating with ArgoUML did not require *customizing* the EVL engine and therefore we anticipate that we can use the same technique to integrate EVL with additional Java-based modelling tools and particularly with the Eclipse Graphical Modelling Framework (GMF) [16].

## 4 Case Study

In this section we present a case study of comparing EVL and OCL in the context of a common scenario. The purpose of the case study is to present readers with the concrete syntax of the language and demonstrate the benefits delivered by the proposed extensions.

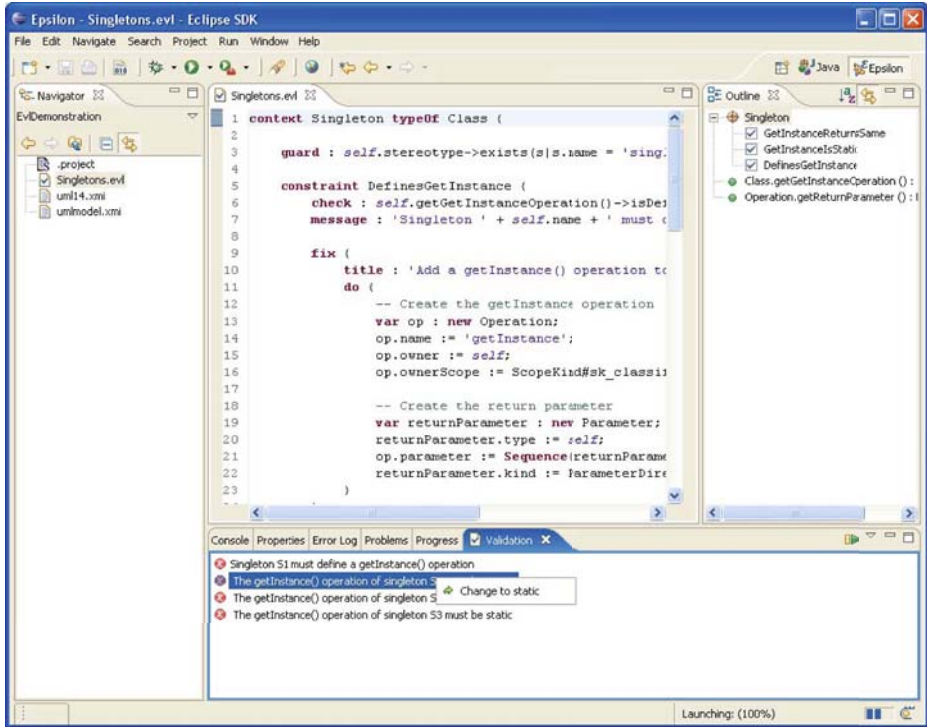


Fig. 2. Screenshot of the Eclipse-based EVL Development Tools

#### 4.1 Scenario: The Singleton Pattern

The *singleton* pattern is a widely-used object oriented pattern. A *singleton* is a class for which *exactly one instance is allowed* [17]. In UML, a singleton is typically represented as a class which is stereotyped with a `<<singleton>>` stereotype and which also defines a static operation named `getInstance()` that returns the unique instance.

To ensure that all singletons have been modelled correctly in a UML model we need to evaluate the following invariants on all classes that are stereotyped with the `<<singleton>>` stereotype:

- `DefinesGetInstance` : Each stereotyped class must define a `getInstance()` method
- `GetInstanceIsStatic` : The `getInstance()` method must be static
- `GetInstanceReturnsSame` : The return type of the `getInstance()` method must be the class itself

Obviously, invariants `GetInstanceIsStatic` and `GetInstanceReturnsSame` depend on `DefinesGetInstance` because if the singleton does not define a `getInstance()` operation, checking for the operation's scope and return type is meaningless. Moreover, in case an invariant fails, there are corrective actions (fixes) that users may want to perform semi-automatically: e.g. for `DefinesGetInstance`, such an action would be to add the

missing `getInstance()` operation, for `GetInstanceIsStatic` to change it to static and for `GetInstanceReturnsSame` to set the return type to the class itself. In the following sections we use OCL and EVL to express the three constraints and comment on each solution.

#### 4.1.1 Using OCL to Express the Invariants

Listing 1.4 shows the aforementioned invariants implemented in OCL.

**Listing 1.4.** OCL Module for Validating Singletons

```

1  package Foundation::Core
2
3      context Class
4
5      def isSingleton :
6          let isSingleton : Boolean =
7              self.stereotype->exists(s|s.name = 'singleton')
8
9      def getInstanceOperation :
10         let getInstanceOperation : Operation =
11             self.feature->select(f|f.oclIsTypeOf(Operation)
12             and f.name = 'getInstance')->first().oclAsType(Operation)
13
14     inv DefinesGetInstanceOperation :
15         if isSingleton
16             then getInstanceOperation.isDefined
17             else true
18         endif
19
20     inv GetInstanceOperationIsStatic :
21         if isSingleton then
22             if getInstanceOperation.isDefined
23                 then getInstanceOperation.ownerScope = #classifier
24                 else false
25             endif
26         else
27             true
28         endif
29
30     inv GetOperationReturnsSame :
31         if isSingleton then
32             if getInstanceOperation.isDefined then
33                 if getInstanceOperation.returnParameter.isDefined
34                     then getInstanceOperation.returnParameter.type = self
35                     else false
36                 endif
37             else
38                 false
39             endif
40         else
41             true
42         endif
43
44     context Operation
45
46     def returnParameter :
47         let returnParameter : Parameter =
48             self.parameter->select(p|p.kind = #return)->first()
49
50 endpackage

```

By examining the OCL solution we note that all invariants first check that the class is a singleton (lines 15, 21 and 31) by using the *isSingleton* derived property defined in line 5. If the *isSingleton* returns *false*, the invariants return *true* since returning false would cause them to fail for all non-singleton classes. This reveals an additional shortcoming of OCL: if a constraint returns *true* it may mean two different things: either that the instance satisfies the constraint or that the constraint is not applicable to the instance at all. To our view, this overloading reduces understandability.

By further studying the solution of Listing 1.4 we observe that dependency between constraints is captured artificially using nested *if* expressions. For instance, both *GetInstanceIsStatic* and *GetInstanceReturnsSame* contain an *if* expression in lines 22 and 32 respectively, where they check the value of the *getInstanceOperation* defined in line 9, where they actually recalculate the result of the *DefinesGetInstanceOperation* invariant. As discussed in Section 2.3, this happens because OCL lacks constructs for capturing dependencies in a structured manner.

#### 4.1.2 Using EVL to Express the Invariants

Listing 1.5 provides a solution for this problem expressed in EVL.

Listing 1.5. EVL Module for Validating Singletons

```

1  context Singleton typeOf Class {
2
3  guard : self.stereotype->exists(s|s.name = 'singleton')
4
5  constraint DefinesGetInstance {
6  check : self.getGetInstanceOperation()->isDefined()
7  message : 'Singleton ' + self.name +
8  ' must define a getInstance() operation'
9  fix {
10  title : 'Add a getInstance() operation to singleton ' + self.name
11  do {
12  -- Create the getInstance operation
13  var op : new Operation;
14  op.name := 'getInstance';
15  op.owner := self;
16  op.ownerScope := ScopeKind#sk_classifier;
17
18  -- Create the return parameter
19  var returnParameter : new Parameter;
20  returnParameter.type := self;
21  op.parameter := Sequence{returnParameter};
22  returnParameter.kind := ParameterDirectionKind#pdk_return;
23  }
24  }
25  }
26
27  constraint GetInstanceIsStatic {
28  guard : self.satisfies('DefinesGetInstance')
29  check : self.getGetInstanceOperation().ownerScope =
30  ScopeKind#sk_classifier
31  message : ' The getInstance() operation of singleton '
32  + self.name + ' must be static'
33
34  fix {
35  title : 'Change to static'
36  do {
37  self.getGetInstanceOperation.ownerScope
38  := ScopeKind#sk_classifier;
39  }
40  }
41  }
42

```

```

43  constraint GetInstanceReturnsSame {
44
45    guard : self.satisfies('DefinesGetInstance')
46    check {
47      var returnParameter : Parameter;
48      returnParameter := self.getReturnParameter();
49      return (returnParameter->isDefined()
50              and returnParameter.type = self);
51    }
52    message : ' The getInstance() operation of singleton '
53              + self.name + ' must return ' + self.name
54
55    fix {
56      title : 'Change return type to ' + self.name
57      do {
58        var returnParameter : Parameter;
59        returnParameter := self.getReturnParameter();
60
61        -- If the operation does not have a return parameter
62        -- create one
63        if (not returnParameter.isDefined()){
64          returnParameter := Parameter.newInstance();
65          returnParameter.kind := ParameterDirectionKind#pdk_return;
66          returnParameter.behavioralFeature :=
67            self.getInstanceOperation();
68        }
69        -- Set the correct return type
70        returnParameter.type := self;
71      }
72    }
73  }
74 }
75
76 operation Class getGetInstanceOperation() : Operation {
77   return self.feature->
78     select(o:Operation|o.name = 'getInstance').first();
79 }
80
81 operation Operation getReturnParameter() : Parameter {
82   return self.parameter->
83     select(p:Parameter|p.kind =
84       ParameterDirectionKind#pdk_return).first();
85 }

```

The *Singleton* context defines that the invariants it contains will be evaluated on instances of the UML *Class* type. Moreover, its guard defines that they will be evaluated only on classes that are stereotyped with the *singleton* stereotype. Therefore, unlike the OCL solution of Listing 1.4, invariants contained in this context do not need to check individually that the instances on which they are evaluated are singletons.

Constraint *DefinesGetInstance* defines no guard which means that it will be evaluated for all the instances of the context. In its *check* part, the constraint examines if the class defines an operation named *getInstance()* by invoking the *getGetInstanceOperation()* operation. If this fails, it proposes a fix that adds the missing operation to the class.

Constraint *GetInstanceIsStatic* defines a guard which states that for the constraint to be evaluated on an instance, the instance must first satisfy the *DefinesGetInstance* constraint. If it doesn't, it is not evaluated at all. In its *check* part it examines that the *getInstance()* operation is static. Note that here the constraint needs not check that the *getInstance()* operation is defined again since this is assumed by the *DefinesGetInstance* constraint on which it depends. If the constraint fails for an instance, the fix part can be invoked to change the scope of the *getInstance()* operation to static.

Constraint *GetInstanceReturnsSame* checks that the return type of *getInstance()* is the singleton itself. Similarly to the *GetInstanceIsStatic* constraint, it defines that to be evaluated the *DefinesGetInstance* constraint must be satisfied. If it fails for a particular instance, the fix part can be invoked. In the fix part, if the operation defines a return parameter of incorrect type, its type is changed and if it does not define a return parameter at all, the parameter is created and added to the parameters of the operation.

It is worth noting here that by observing the two solutions the OCL one resembles the concept of defensive programming, where conditions are embedded in supplier code, while the EVL one is closer to the design by contract [5] approach where conditions are explicitly checked in guards.

Through this case study we have shown that the additional constructs provided by EVL can reduce repetition significantly and thus enable specification of more concise constraints. Moreover, in case a constraint is not satisfied for a particular instance, the user is provided with a meaningful context-aware message and with an automated facility for repairing the inconsistency.

## 5 Related Work

Although the academic community has shown little interest in extending OCL to make it more practical and usable for contemporary developers, the need for enhanced validation mechanisms similar to those presented in this paper is widely recognized by practitioners and thus implemented in tools. For instance, in the IBM Rational Software Architect, distinguishing between critical errors and warnings is achieved by adding appropriate stereotypes to constraints in UML models. In the ArgoUML modelling tool, validation functionality has been implemented using Java with an overall rationale quite similar to the one presented in this paper.

In the context of the openArchitectureWare (oAW) [18] framework, the *Check OCL*-based constraint language has been proposed for capturing errors and warnings in models. Moreover, in [19], a model transformation language (ATL) is used to check models by transforming them into *Problem* models. While both approaches are more powerful than plain OCL, they do not address the issues of defining constraint dependencies or repairing identified inconsistencies.

## 6 Conclusions and Further Work

In this paper we have identified a number of shortcomings of OCL in the context of capturing structural constraints for modelling languages and proposed a set of extensions to the OCL abstract and concrete syntaxes that address those shortcomings effectively. As proof of concept we have implemented the proposed extensions in the prototype Epsilon Validation Language with tool-support for Eclipse and ArgoUML. In the case study we used a concrete example and demonstrated solutions using both plain OCL and EVL and showed the benefits delivered by the additional constructs of EVL.

As discussed in Section 3 we do not consider EVL as a rival language to OCL but as a useful prototype through which we can experiment with new features and extensions to the language that can hopefully be considered for adoption in future versions of the standard.

There are two main directions for further research on the subject: to enrich EVL with additional constructs that can further simplify definition of constraints, and to assess how the proposed extensions affect work done in the context of incremental evaluation of constraints, which is essential for achieving scalability.

## Acknowledgements

The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

## References

1. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, <http://www.omg.org/cgi-bin/doc?ptc/03-10-04>
2. Object Management Group. UML 2.0 OCL Specification, <http://www.omg.org/docs/ptc/03-10-14.pdf>
3. 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* 7(3), 212–232 (2005)
4. Barnett, M., DeLine, R., Jacobs, B., Fährdrich, M., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.) *VSTTE 2005*. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
5. Meyer, B.: *Object-Oriented Software Construction*, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
6. Kolovos, D.S.: Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), <http://www.cs.york.ac.uk/~dkolovos/epsilon>
7. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) *ECMDA-FA 2006*. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
8. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: *Proc. Tool Support for OCL and Related Formalisms - Needs and Trends, Models/UML 2005*, Montego Bay, Jamaica (October 2005)
9. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Dubois, E., Pohl, K. (eds.) *CAiSE 2006*. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg (2006)
10. Egyed, A.: Instant consistency checking for the UML. In: *ICSE 2006: Proceeding of the 28th international conference on Software engineering*, Shanghai, China, pp. 381–390. ACM Press, New York (2006)
11. Eclipse Foundation, <http://www.eclipse.org>
12. ArgoUML, <http://argouml.tigris.org>
13. Eclipse.org. Eclipse Modelling Framework, <http://www.eclipse.org/emf>
14. Sun Microsystems. Meta Data Repository, <http://mdr.netbeans.org>
15. Epsilon component - Eclipse Generative Modeling Technology (GMT), <http://www.eclipse.org/gmt/epsilon>

16. Eclipse GMF - Graphical Modeling Framework, <http://www.eclipse.org/gmf>
17. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edn. Prentice Hall PTR, Englewood Cliffs (2004)
18. openArchitectureWare, <http://www.openarchitectureware.org/>
19. Jouault, F., Bezivin, J.: Using ATL for Checking Models. In: Proc. International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia (September 2005)



# Ten Commandments Ten Years On: Lessons for ASM, B, Z and VSR-net

Jonathan P. Bowen<sup>1</sup> and Michael G. Hinchey<sup>2</sup>

<sup>1</sup> Museophile Limited  
Reading, United Kingdom  
jpbowen@gmail.com  
[www.jpbowen.com](http://www.jpbowen.com)

<sup>2</sup> Loyola College in Maryland  
Computer Science Department, Baltimore, Maryland, USA  
mhinchey@loyola.edu  
[www.cs.loyola.edu](http://www.cs.loyola.edu)

**Abstract.** Just over a decade ago, a paper *Ten Commandments of Formal Methods* [16] suggested some guidelines to help ensure the success of a formal methods project. It proposed ten important requirements (or “commandments”) for formal developers to consider and follow, based on our knowledge of several industrial application success stories, most of which have been reported in more detail in two books [32,33]. The paper was surprisingly popular, is still widely referenced, and used as required reading in a number of formal methods courses. However, not all have agreed with some of our commandments, feeling that they may not be valid in the long-term. We re-examine the original commandments over ten years on, and consider their validity in the light of a further decade of industrial best practice and experiences, especially with respect to formal notations like ASM, B and Z. We also cover the activities of the UK Verified Software Repository Network (VSR-net) in the context of UK Grand Challenge 6 on Dependable Systems Evolution.

## 1 Introduction

*Painting is easy when you don't know how, but very difficult when you do.*  
— Edgar Degas (1834–1917)

In our everyday world, we are constantly bombarded with examples of problems with human design, be it using hardware or software. Sometimes the situation is not too critical, but in a real safety-critical setting, the resulting problems could be far more serious and the design engineer has a major responsibility to protect the lives of any humans involved [14].

There have been problems with the use of hardware since the dawn of civilization. However, the use of software is a newer and still less well understood phenomenon. This is partly due to the additional complexity introduced in systems by the use of software. The potential for mistakes is exponential and the mechanisms to prevent these are still in active development. In addition, the digital nature of software, in contrast to

the analogue nature of many hardware systems, means that traditional techniques of extrapolation and interpolation are impossible to extend to the software field. A single incorrect bit can produce catastrophic results.

Given the number of systems controlled by software that now directly affect our lives, whether in a critical or non-critical manner, we need techniques to reduce the number of errors in these systems in order to minimise the impact of problems that they may cause. This paper explores the use of one particular technique in the development of computer-based systems, namely formal methods [34]. These mathematically-based approaches have the potential to help eliminate errors early in the design process rather than try to remove them later on in the testing phase, or worse, even later. Of course both approaches (i.e., formal methods and testing) are important and are even complementary. The use of formal methods at the initial stages can help in improving the software at the later stages. It has been argued by Daniel Berry [4] that this is because the developer is in essence undertaking the development twice, and that we always do things better with the benefit of hindsight. Notwithstanding, many success stories (see [32,33], amongst others) illustrate the potential benefits of formal methods.

In this paper, we first briefly introduce formal methods, and specifically address ASM [13,28], the B-Method [2,50] and the Z notation [39,43], three leading approaches to aid in the formal development of software. We then consider a number of “commandments” [16,19,20] in relation to using these formal methods at various stages in the design process, especially with respect to developments over the last decade. We also present a “Grand Challenge” initiative in the United Kingdom inspired by Sir Tony Hoare [36,38], designed to encourage the improvement of formal approaches to software verification and development, and related activities concerning a planned Verified Software Repository [6,41,56]. Finally we draw some general conclusions.

## 2 Formal Methods

*The ability to simplify means to eliminate the unnecessary so that the necessary may speak.*

— Hans Hofmann (1880–1966)

The term “formal methods” [17] as we know it now has been in use since the late 1970s; the term was used in logic in a different context a century previously. Although the exact origins of the term seem to have been lost in the midst of time, unlike the term “software engineering”, which was coined through NATO conferences in the late 1960s; see an online archive from one of the original editors [47]. A book with the title “Formal Methods” by the Dutch logician and philosopher Evert W. Beth was published in 1970 (although Beth died in 1964).

Perhaps the first real example of the use of formal methods is the proof of a program presented by Alan Turing in the late 1940s [45]. C. A. R. Hoare’s classic 1969 paper on the axiomatic basis of computer programming was certainly an important turning point in the recognition of the importance of a formal basis for software [35].

Formal methods grew out of the structured programming paradigm as it was realised that a mathematical (and specifically logical) basis to software development could

provide a key underpinning to ensure correctness of a program with respect to a specification. Suitable mature tool support is needed to help ensure scalability and industrial applicability.

Important aspects of formal methods are a mathematical specification of the product and the ability to prove the implemented program meets this specification, even if a full proof is not carried out in practice due to engineering and cost limitations. This process is known as verification. Another useful use of proof is validation, where a challenge property of a specification is demonstrated to increase the confidence that it is indeed the specification that is desired. If a required validation property does not in fact hold, it illustrates a probable misunderstanding in the specification itself, which should be rectified in discussion with the customer.

Over the years, there has been much misunderstanding about formal methods and the way that they can usefully be applied [16]. Certainly it is very easy to use formal methods inappropriately and it is very important that the project manager has a full understanding and appreciation of how and where formal methods can be helpfully used in the development process. The impact of using formal methods inappropriately seems particularly high because it is possible to devote huge amounts of time to proofs, which may or may not be worthwhile in practice, depending on the circumstances.

For example, GEC Alsthom spent decades of person years developing a few thousand lines of program code fully formally, with tool support in the guise of the B Tool, for the Paris subway RER line A to the (then) new Euro Disney site [23]. This controlled the automated train protection (ATP) system, controlling the braking, etc., and crucially allowing trains to be more closely spaced while still maintaining safety. Around half the development time was spent on the verification process. This may sound like much effort for little reward. However, the alternative would have been to build a new tunnel, an obviously far more expensive endeavour. In such circumstances, what may appear to be excessive cost initially can actually be a very cost-effective alternative in practice.

### 3 Levels of Abstraction

*Abstraction is the way to the heart — it is not the heart itself.*

— John Piper (1903–1992)

Formal methods are not widely used in software development in general. However, they have achieved niche use in high-integrity systems where safety or security is paramount [114], or the cost of failure would be huge (i.e., in business-critical applications). Here we consider the impact of using formal methods in software development. Most software projects start with a set of informal requirements agreed with the customer, ideally in advance, although requirements are notorious for changing even during the lifetime of a software project. These could be relatively simple. In Table 1 below [19], by way of example, we consider a hypothetical project where the initial requirements are only 25 lines, which may be the result of discussions between a supplier and a customer. It is even possible to formalise this high level of abstraction, for example using “lightweight” formal methods [27] or a domain-specific formalism such as Duration Calculus for real-time systems [58].

**Table 1.** Levels of complexity [19]

25 lines of informal requirements
250 lines of specification (e.g., Z)
2,500 lines of design description (e.g., B)
25,000 lines of high-level program code
250,000 machine instructions of object code
2,500,000 CMOS transistors in hardware

The contractor could take these rather short and informal requirements and produce a specification document, detailing what the system is intended to do. This could be informal, with a natural language description and diagrams, perhaps with some tool support. Alternatively it is possible to formalise for the first time at this level of abstraction. A typical approach would be to use the Z notation [39], based on logic and set theory with the addition of schema boxes for structuring. This could be 250 lines long, i.e., an order of magnitude larger than the original requirements. Unless the various levels in a development process introduce significant additional complexity and information, there is little point in including such a step.

Now it is helpful to describe the design of *how* the desired software is to operate in more detail, including algorithms, data structures, etc. Again, much more detail could be introduced, making the design description 2,500 lines long, for example. Industry often uses UML (The Unified Modeling Language) [8] at this level of abstraction. Using a more formal approach, a tool like Atelier B could be used to develop a high-level specification and transform to a lower-level algorithmic description close to a compilable programming language.

The program produced from the design stage, whether produced formally or informally, could be an order of magnitude larger than the design documentation (i.e., 25,000 lines of code in our running example). This may be the first time a fully formal piece of text is produced on many software projects. In fact, in this sense, all software development involves formal methods. It is just that often the formality is not introduced until the last possible stage, as an executable high-level program.

By this time, this executable formal description is many times larger than its counterpart at the specification stage. Thus, if only because of scale, it is much more difficult to reason about and manipulate. In addition, most programming languages are not designed for formal manipulation, although the use of assertions in programs, originally developed for proof rather than testing purposes [35], is increasingly common.

Once an executable program is realized, the process of generating lower level descriptions becomes increasingly automated. A compiler can produce object code suitable for direct execution by a processor. This may have an order of magnitude more instructions in it than the high-level program equivalent (i.e., 250,000 machine instructions in our example). This could be implemented in hardware of yet more complexity, perhaps 2,500,000 CMOS transistors for example.

Formal methods can be introduced and used at any of these levels of abstraction. However, the impact is likely to be much more cost-effective at the higher levels such as the requirements or specification. At these stages, the description is still relatively

small and can be changed easily at little cost. At each development stage, the cost of correcting errors becomes very much more expensive.

It is highly desirable to correct errors early on before proceeding to further stages of development. However, this can easily give the appearance of delaying a project because it is superficially apparent to a naïve observer (such as a manager inexperienced in the use of formal methods) that there has been little progress with respect to the generation of executable code until relatively late in a project that uses formality at the initial stages.

## 4 Ten Commandments

*There is no mistake so great as the mistake of not going on.*

— William Blake (1757–1827)

In this section we briefly cover some developments in the past decade relevant to the original “ten commandments” of formal methods presented in [16], especially in relation to ASM, B and Z. Further, more extensive, information can be found in [19].

### I. Thou Shalt Choose an Appropriate Notation

Choosing a formal method can be difficult for an industrial practitioner. Around 95 different approaches are listed on the main Virtual Library Formal Methods web page [<http://vl.fmnet.info>]. Of these, ASM, B and Z are three of the leading approaches.

ASM has been demonstrated to be useful in combination with testing, as has been done at Microsoft with Component Object Models [3], and in the Falko project at Siemens, where apparently testing was the only viable means of verification [13].

B has particularly good mechanized tool support for software development compared to most formal methods. Atelier B [<http://www.atelierb.societe.com>] is a tool used by industry. B4free [<http://www.b4free.com>] is a free version of this tool that is useful for teaching. For the future, The European Rodin project is developing a new B-based development environment of industrial quality [<http://rodin.cs.ncl.ac.uk>].

Z is an excellent general purpose formal specification language with an ISO standard [39]. However its tool support has been weaker than could be desired. The Z/EVES tool [48] provides good theorem proving support (with type-checking as well), but is quite difficult to use and is no longer supported. The Community Z Tools (CZT) project [<http://czt.sourceforge.net>] is a collaborative project to develop open source tools for Z, based around the SourceForge website facilities.

Increasingly all formal methods are providing XML support for interoperability, including ASM, B and Z. Indeed, integrated formal methods continues to be a topic of interest for research and application [7]. This trend is likely to continue. Although most efforts for individual notations are rather independent at the moment, there is some coordination. For example, an *International Workshop on Web Languages and Formal Methods* (WLFM 2005) was held in conjunction with the FM’05 conference [<http://www.w3c.rl.ac.uk/WLFM2005>].

ASM offers a significant advantage over both Z and B in that it is not tied to a first-order set theoretic language and theorem provers, but may be used with any mathematical proof framework. In addition, the ability to utilize operational abstract pseudo-code

to address system dynamics, while still utilizing declarative mathematical axioms for environmental assumptions, offers significant advantages over the declarative nature of languages such as Z. This flexibility in specification and proving properties comes at the cost of difficulties in implementation and mechanization, however.

## II. Thou Shalt Formalize But Not Over-Formalize

Formal methods continue to have different levels of use. ASM and Z can be used at quite a light high-level with little tool support apart from type-checking if desired. Refinement towards an implementation is also possible [9][24], although the theory is more advanced than current practical tool support. For formal development, B provides an excellent framework and selection of tools for actual software development. Properties can be proved and refinement calculus applied in a stepwise manner, with many of the proofs performed automatically.

Theorem provers and model checkers allow full machine checking but can be difficult to use and limited in scale. However, the Alloy Analyzer [<http://alloy.mit.edu>], with its Z-like notation, allows a relatively lightweight approach to analyzing models. It can usefully be used for checking potentially tricky aspects of real systems at relatively low cost.

Animation of formal specifications can be beneficial. The ProB tool [<http://www.stups.uni-duesseldorf.de/ProB>] and the newer ProZ tool allow this approach for B and Z, respectively.

A significant number of open source tools for ASM are in existence and several have been used in successful industrial projects ranging from hardware control systems to protocol verification. These include the component-based ASM tool environment CoreAsm [<http://www.coreasm.org>], downloadable from SourceForge, as well as ASM-based hardware and software design and analysis tools that support real-time features. AsmL (Abstract State Machine Language) [<http://research.microsoft.com/fse/asm/>] from Microsoft is an executable specification language based on ASM, integrated within the Microsoft Word editor. AsmL has also been integrated into the C# programming environment of Microsoft's .NET. The Spec Explorer tool can be used to explore models written in AsmL [53].

## III. Thou Shalt Estimate Costs

The cost of using formal methods is difficult to estimate because of the different quality of people involved, as with all software development. The specialist nature of formal methods makes such estimation even more difficult. A study at NASA has demonstrated that increased effort at the requirements phase of a project results in lower cost overruns later [19], something that has also been experienced when using ASM [12].

A tool such as those for B can automate 90–95% of the proof obligations associated with the refinement from the specification to code using heuristics within the tool. An experienced and expert B user can help ensure that this figure is nearer 95% than 90%. Since the remaining 5–10% of proof obligations must be proved with manual intervention, the human effort involved could vary considerably as a result. In addition, the more

naïve user will find the proofs that do need manual direction, through the addition of suitable lemmas for example, considerably more difficult than the expert. Because of this, it is well worth obtaining more expert formal methods experts at a higher cost per day to reduce overall costs and development time.

#### IV. Thou Shalt Have a Formal Methods Guru on Call

Technology transfer is an important part of ensuring success in the use of formal methods. For example, with Z, there have been courses both from academia and industry, a reasonable choice of textbooks (around 15 [<http://vl.zuser.org/pub/faq.txt>]), tools for type-checking and proofs, web resources [<http://vl.zuser.org>], a discussion forum (electronic mailing list and newsgroup), a user group with regular meetings [<http://www.zuser.org>] and an ISO standard [39]. Similar user groups and regular meetings exist for B and ASM.

More generally, Formal Methods Europe [<http://www.fmeurope.org>] founded the Formal Techniques Industry Association (ForTIA) [<http://www.fortia.org>] in 2003 to specifically support technology transfer of formal methods to industry. The European Research Consortium for Informatics and Mathematics (ERCIM) Working Group on Formal Methods for Industrial Critical Systems(FMICS) has been promoting the industrial use of formal methods for the last decade.

#### V. Thou Shalt Not Abandon Thy Traditional Development Methods

Most of industry does not use formal methods. Instead, over the last decade, UML (The Unified Modeling Language) [8] has increasingly dominated the favoured tool-supported approach of industry for software development. A major flaw in the eye of the formal methods community, and benefit in the eyes of some practitioners, is that UML has an ill-defined semantics, and is indeed a loosely associated collection of tools branded collectively as UML.

The precise UML group (pUML) [<http://www.cs.york.ac.uk/puml/>] has looked at formalizing parts of UML. In addition, some efforts have been made to produce formal methods tools that integrity with UML. For example, the UML-B approach with the U2B translation tools enables integration of UML and B [51]. For increased use of formal methods in the future, it will probably be necessary to consider the use of UML in association with the formal methods itself. There have also been efforts to provide precise semantics for UML activity diagrams, state machines and sequence diagrams, which have subsequently been incorporated into ASM execution tools; e.g., [22] uses AsmGofer to build a simulator for UML state diagrams, while [10] gives an ASM semantics for UML activity diagrams and extended in [11].

Object-orientation is also very popular and indeed object-oriented ideas were originated by O.-J. Dahl who was to go on to be prominent in the field of formal methods [46]. Object-oriented extensions of formal approaches are possible. For example, the most popular extension for Z is Object-Z [25]. However, tool support is less extensive and its use is less widespread too.

An object-oriented proof tool, similar to those for B, called Perfect Developer has been developed by the UK firm Escher Technologies [<http://www.eschertech.com/products>]. It targets to Java and C++ program implementations. A commendable feature



of the tool is that it has been applied to itself, sadly still a rare occurrence among formal methods approaches. It can automatically prove around 95% of the approximately 130,000 verification conditions that are generated [19]. The remaining conditions have not been proved due to lack of resources to do this.

## VI. Thou Shalt Document Sufficiently

Documenting successes and failures in the use of formal methods is important. Two books have attempted to give examples of applications and industrial use of formal methods [32][33], including B and Z. Documenting real successes can be difficult because companies do not wish to reveal information that might be useful to their competition. Equally, documenting failures can be difficult to encourage because of possible bad publicity for the company concerned.

Much documentation concerning formal methods presents finished specifications. One of the real benefits of a formal approach is the knowledge gained during the production of a formal specification and reasoning about it for validation purposes. A book recently reissued in a second edition [26] attempts to present a number of formal specification approaches applied to a single case study. A unique aspect of the book is that the material is presented by going through a series of questions together with answers to develop the eventual formal specification from the initial informal English requirements. This helps to give the reader an insight into the various approaches presented. There are three Z-related examples: “vanilla” Z in Chapter 1, SAZ (a combination of the SSADM diagram and text approach with Z) in Chapter 2, and UML+Z (adding Z to UML) in Chapter 5. There are also three B-based approaches: using B itself in Chapter 3, an approach transforming UML into B in Chapter 4, and using the event-based Event-B in Chapter 9. The ASM approach is covered in Chapter 6. A comparison of all the approaches covered is given in Chapter 19.

There are relatively few textbooks for formal methods. For the Z notation, there have been around 15 textbooks as previously mentioned (e.g., [43]), for B less than half this number (e.g., [50]) and ASM still fewer (e.g., [13]). This compares with the Java programming language, for example, where around two orders of magnitude of books exist.

Online, there are websites for ASM, B and Z linked from the Virtual Library formal methods pages [<http://vl.fmnet.info>], which have been developed over the past decade. More recently, there are also entries linked from the main Wikipedia category on formal methods [[http://en.wikipedia.org/wiki/Category:Formal\\_methods](http://en.wikipedia.org/wiki/Category:Formal_methods)]. However, documentation on the practical application of formal methods is still relatively scant online. A Verified Software Repository is planned that may provide more real and accessible examples for the future [64][56]. One of the goals of the ASM Research Center is to provide a repository of design and analysis work, see [<http://www.asmcenter.org>].

## VII. Thou Shalt Not Compromise Thy Quality Standards

With regard to standards, a decade ago some standards mentioned or even mandated the use of formal methods in the development of the most critical systems. The UK Ministry of Defense 00-55 and 00-56 standards including mandated use of formal methods for



safety-related software. 00-55 was updated in 1997 and Issue 3 of 00-56 for hardware and software was made available in 2005.

In 2002, an ISO standard for the Z notation was accepted and published [39] after its acceptance in 2001. However this was issued after a long gestation period during much of the 1990s. There are many hurdles for the acceptance of an international standard through ISO and agreement from several countries as required, with the possibility of updates being needed due to feedback from members of ISO.

A de facto definition of Z has been available since the early 1990s in the form of a book [52] (dubbed “ZRM” for the Z Reference Manual, now freely available online in a slightly updated form since 2001). However, this did not specify the semantics of Z and much of the time in the production of the Z standard was spent in the discussion and production of an acceptable semantics [30]. While this was beneficial in clarifying some aspects of Z that were potentially unsound (especially concerning schemas, used for structuring specifications), it caused considerable delays in the production of the standard. In addition, further features were added to the style of Z presented in the standard.

With hindsight, it may have been advisable to adopt the ZRM book as the standard with minimal changes. It would have then been available considerably earlier, possibly with more benefit for producers of tools for Z. With the progress of time there have been some benefits, such as the inclusion of additional Z symbols in UNICODE and the consideration of an XML-based markup language for Z, for use on the web.

There are currently no explicit plans for the production of ISO standards for ASM and B. The Abstract State Machines book [<http://www.di.unipi.it/AsmBook>] [13] does however act as a de facto standard for core ASM concepts and the B-Book [2] takes a similar role for B. It may be that this is all that is required in practice, especially with the increased ease of access to material using the web. The more recent development of Event-B, an event-based version of B, is likely to need a similar widely accepted de facto standard [44].

### VIII. Thou Shalt Not be Dogmatic

There should be some flexibility in the choice of notations and tools when applying formal methods. While B is tied by definition of the method to mechanical tool support, Z and ASM are not. It is important to listen to industry’s problems and let this guide the development and use of tools, rather than being driven from an entirely academic viewpoint. Good industrial-strength tool support is often the most critical aspect for commercial use and formal methods still do not have a very good record on this.

Z/EVES has been one of the best Z theorem provers available over the last decade. However, although this was developed by the Canadian company ORA, it is officially owned by the Canadian government and is no longer generally available under license except to organizations with existing agreements. More recently, the international Community Z Tools (CZT) initiative has been developing a suite of open source tools for the support of Z. These are available on a SourceForge website [<http://czt.sourceforge.net>] and it is possible to contribute to the project. However CZT is essentially a volunteer

and academic effort, which can be treated with some suspicion in industrial, where good reliable support for software tool is crucial and the cost is much less of an issue than in academia.

Fortunately, the situation is somewhat better with respect to B. Atelier-B is a commercially available tool for B and includes code generation facilities. The B4free tool [<http://www.b4free.com>] is a freely available version suitable for academic use, but lacking code generation. The European RODIN Project on Rigorous Open Development Environment for Complex Systems [<http://rodin.cs.ncl.ac.uk>], running from 2004 to 2007, is developing excellent tool support with an impressive user interface for a newer variant of B known as B# (“B sharp”, like the C# programming language). Support for changing the specification and handling the implications of this as automatically as possible is an aspect that is being considered serious on this project. This is commendable since it is something that is lacking in many existing formal methods tools, but it is crucial in actual software development where software changes and maintenance are a major part of the overall cost in practice. The RODIN B# tool is available from a SourceForge website [<http://rodin-b-sharp.sourceforge.net>].

Tool support for ASM is less well developed than for B, and this will be a critical aspect for industrial take-up of ASM. There are however tools based on theorem provers for verification and validation of ASMs; theorem provers used for these tools include KIV, Isabelle and PVS.

## IX. Thou Shalt Test, Test, and Test Again

The availability of a formal specification (e.g., in the framework of ASM, B or Z), means that there is a very precise description against which the final software can be tested. ASM has been found to be particularly useful with respect to testing, as mentioned earlier, but the relationship of specification and testing has also been explored in the context of Z as well.

If the formal specification can be animated, it can be used as a “test oracle”, and this approach has been used successfully in the context of ASM. Even if the formal specification is not easily animatable (as with Z in general), it is possible to use a formal specification to semi-automatically determine worthwhile test sets (e.g., to cover all the various preconditions in an operation). Indeed, this can reduce the cost of testing considerably, perhaps even more than the cost of producing the formal specification itself, since this typically requires considerable manual effort.

Formalizing test criteria themselves can also be beneficial since there are normally defined informally in practice and there can be misunderstanding as to what is required for testing with respect to various criteria. Indeed, there are sometimes subtle variants that may cause confusion for those that are not expert in the field.

MC/DC is one such criteria, mandated by the US FAA (Federal Aircraft Authority) for critical aircraft software. Even stricter criteria are possible for greater assurance. For example, RC/DC, based on MC/DC, has been suggested [54,42]. The formal definition of the RC/DC criterion can be formulated in terms of the existing MC/DC criterion (formalised using the Z notation) as follows [54]:

$RC\_DC$
$MC\_DC$
$\forall DecReinforced; c : cond \mid c \in argdec \bullet$ $(\mathbf{let} \ target0 == choice(keep0fix \ c, keep0 \ c);$ $\ \ \ \ target1 == choice(keep1fix \ c, keep1 \ c) \bullet$ $\ \ \ \ (target0 \neq \emptyset \Rightarrow (testset \times testset) \cap target0 \neq \emptyset) \wedge$ $\ \ \ \ (target1 \neq \emptyset \Rightarrow (testset \times testset) \cap target1 \neq \emptyset))$

The predicate part of the Z schema above adds an additional constraint to be satisfied.

In the United Kingdom, a government-funded EPSRC Network on formal methods and testing (FORTEST) [15], with academic and industrial partners, ran between 2001 and 2005 [<http://www.fmnet.info/fortest>]. An associated book is due to appear [31] and the TAICPART conference series (Testing: Academic & Industrial Conference — Practice And Research Techniques) continues to address these issues [29].

## X. Thou Shalt Reuse

Meaningful reuse is possible in general as soon as a sufficient level of formality has been introduced into the description language being used. Normally in a software project, this is only at the programming stage (see Table 1 earlier). In a project employing formal specification (e.g., ASM, B or Z), this could be at a higher level of abstraction. Since the description is considerably less complex at these levels, there is the potential for it to be easier and cheaper as well.

However, formal methods still do not have a good record of reuse in practice and this is something that tools developers should consider seriously. The RODIN tool, as mentioned earlier, is probably the leading formal methods tool under development that is attempting to address this problem.

## 5 A Grand Challenge and the Verified Software Repository

For the future, realistic examples of formal specification and verification could help to set goals for the improvement of formal methods. In the United Kingdom, there is a Hilbert-style “Grand Challenge” initiative considering seven difficult and unsolved problems in computer science [38]. Grand Challenge 6 [<http://www.fmnet.info/gc6>] concerns *Dependable Systems Evolution*, inspired by Tony Hoare’s concept of a verifying compiler in particular [36] and verified software in general [37]. A workshop was held in conjunction with the FM’05 conference in 2005 [21].

In 2005, the three-year United Kingdom EPSRC Network VSR-net started, based around the idea of a Verified Software Repository to document verified software attempts and case studies [<http://www.fmnet.info/vsr-net>]. This has been partly inspired by the existing QED Pro Quo repository of tools in the US [<http://www.qpq.org>].

Initially the security-critical Mondex Electronic Purse example (developed by Logica for the NatWest bank) has been studied [56,57]. This was originally specified using Z with hand proofs for verification in the late 1990s. At the time although Z proof

tools such as Z/EVES were available, it would have taken days to run the proofs using the hardware of the time, making this intractable in practice. Now the proofs have been replayed using Z/EVES and on current hardware they only take hours to complete, making this a reasonable proposition. In addition, mechanization has revealed a number of minor problems in the hand proofs, something that is to be expected in any proof of industrial scale. Fortunately there are no major issues.

The case study has been attempted using a number of other formal approaches and tools as well. In general these have discovered the same problems independently, which is a pleasing aspect of the experiment. The alternative approaches used besides Z have included RAISE, Perfect Developer (a B-like proof tool), the KIV theorem prover using the ASM for the formalization [49].

## 6 Conclusion

*Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*

— Antoine de Saint-Exupéry (1900–1944)

Formal methods continue to be important in safety and security critical applications, where they have found their niche. They are useful for both hardware (especially using model-checking approaches) and software. ASM, B and Z are mainly suitable for aiding software development. Z is the most mature approach, with an ISO standard. It is general purpose and is especially applicable for formal specification, but still lacks good tool support. B is better for machine-support program development from a specification to program code, with good industrial-level support, and more planned. ASM is the least mature approach in terms of industrial usage, but has been applied to real problems, and has potential for the future if tool and other support can be improved.

For all approaches, tool support is critical for success, and this is why B looks like a favoured practical approach at present out of ASM, B and Z. Open source tools seem to be increasing, which certainly has benefits, especially in the academic world. However, industry requires reliable support even for open source software, and is willing to pay for it. Formal methods developers should take note of this since a professional approach is very important in an industrial context.

The formal methods community is still relatively small. While it has not declined in the last decade, it certainly has not increased dramatically either. Whether it will do so in the future is debatable, although even large software companies like Microsoft now recognize the importance of formal methods in the right context. For example, the problem of third party device drivers causing system crashes has been aided by taking a more formal approach and an ASM-based test oracle approach has also been adopted in parts of the company. This is the sort of environment where it is possible that there could be a breakthrough in the use of formal methods, but it will require significant technology transfer skills with convincing support (especially through tools) for this to happen.

Interaction between different parts of the formal methods community is also important. The Dagstuhl seminar at which this paper was presented explicitly aimed to help with interaction between the ASM, B and to a lesser extent the Eiffel communities.

For the future, an ABZ Conference is planned for 2008 to continue such interaction between the groups of researchers and practitioners interested in ASM, B and Z [<http://www.abz2008.org>].

## Acknowledgements

The ideas in this paper are a development from two papers that have appeared in *IEEE Computer* over a ten-year period [16,19]. The latter paper was based on [18,20]. Thank you to the organizers of the Dagstuhl Seminar 06191 on *Rigorous Methods for Software Construction and Analysis*, for an excellent meeting and the opportunity to present this paper. We are grateful to the anonymous reviewers for several pointers and insightful comments on an earlier draft. A special thank you to Prof. Egon Börger of the University of Pisa for his support and academic conviviality over the years. He has been instrumental in bringing together the ASM, B and Z communities.

## References

1. Abdallah, A.E., Bowen, J.P., Nissanke, N.: Formal methods for safety critical systems. In: Diab, H.B., Zomaya, A.Y. (eds.) *Dependable Computing Systems: Paradigms, Performance Issues, and Applications, Part I: Models and Paradigms*. Wiley Series on Parallel and Distributed Computing, vol. 9. John Wiley & Sons, Chichester (2005)
2. Abrial, J.-R.: *The B-Bool: Assigning programs to meanings*. Cambridge University Press, Cambridge (1996)
3. Barnett, M., Schulte, W.: Spying on components: A runtime verification technique. In: *Workshop on Specification and Verification of Component-Based Systems*, Technical Report TR 01-09a, Iowa State University, USA, pp. 7–13 (2001)
4. Berry, D.M.: Formal methods: The very idea — Some thoughts about why they work when they work. *Science of Computer Programming* 42(1), 11–27 (2002)
5. Beth, E.W.: *Formal Methods: An Introduction to Symbolic Logic and to the Study of Effective Operations in Arithmetic and Logic*. Synthese Library. Springer, Heidelberg (1970)
6. Bicarregui, J., Hoare, C.A.R., Woodcock, J.C.P.: The Verified Software Repository: A step towards the verifying compiler. *Formal Aspects of Computing* 18(2), 143–151 (2006)
7. Boiten, E.A., Derrick, J., Smith, G.P. (eds.): *IFM 2004*. LNCS, vol. 2999. Springer, Heidelberg (2004)
8. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*, 2nd edn. Addison-Wesley Object Technology Series (2005)
9. Börger, E.: The ASM refinement method. *Formal Aspects of Computing* 15(1–2), 237–257 (2003)
10. Börger, E., Cavarra, A., Riccobene, E.: An ASM semantics for UML Activity Diagrams. In: Rus, T. (ed.) *AMAST 2000*. LNCS, vol. 1816, pp. 293–308. Springer, Heidelberg (2000)
11. Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: [28], pp. 223–241
12. Börger, E., Päppinghaus, P., Schmid, J.: Report on a practical application of ASMs in software design. In: [28], pp. 361–366
13. Börger, E., Stärck, R.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Heidelberg (2003), <http://www.di.unipi.it/AsmBook/>
14. Bowen, J.P.: The ethics of safety-critical systems. *Communications of the ACM* 43(4), 91–97 (2000)

15. Bowen, J.P., Bogdanov, K., Clark, J., Harman, M., Hierons, R., Krause, P.: FORTEST: Formal methods and testing. In: Proc. 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), Oxford, UK, August 26–29, pp. 91–101. IEEE Computer Society Press, Los Alamitos (2002)
16. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods. *IEEE Computer* 28(4), 56–63 (1995)
17. Bowen, J.P., Hinchey, M.G.: Formal methods. In: Tucker Jr., A.B. (ed.) *Computer Science Handbook*, 2nd edn. Section XI, Software Engineering, ch. 106, pp. 106-1–106-25. Chapman & Hall / CRC, ACM (2004)
18. Bowen, J.P., Hinchey, M.G.: Ten commandments revisited: A ten-year perspective on the industrial application of formal methods. In: Margaria, T., Massink, M. (eds.) *FMICS 2005: Proceedings of the Tenth International Workshop on Formal Methods for Industrial Critical Systems*, Lisbon, Portugal, September 5-6, pp. 8–16. ACM Press, New York (2005)
19. Bowen, J.P., Hinchey, M.G.: Ten commandments of formal methods... ten years later. *IEEE Computer* 39(1), 40–48 (2006)
20. Bowen, J.P., Hinchey, M.G.: Ten commandments ten years on: An assessment of formal methods usage. In: Eleftherakis, G. (ed.) *SEEFM 2005: 2nd South-East European Workshop on Formal Methods, Formal Methods: Challenges in the Business World*, Ohrid, Macedonia, South-East European Research Centre, November 18–19, pp. 1–16 (2005)
21. Bowen, J.P., Woodcock, J.C.P. (eds.): Grand Challenge 6 Workshop on Dependable Systems Evolution. Workshop in association with the FM 2005 Formal Methods Conference, University of Newcastle upon Tyne, United Kingdom, July 18 (2005), <http://www.fmnet.info/gc6/fm05/proceedings.pdf>
22. Cavarra, A.: Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method. PhD thesis, University of Catania, Sicily, Italy (2000)
23. Dehbonie, B., Mejia, F.: Formal development of safety-critical software systems in railway signalling. In: [32], ch. 10, pp. 227–252 (1995)
24. Derrick, J.: A single complete refinement rule for Z. *Journal of Logic and Computation* 10(5), 663–675 (2000)
25. Duke, R., Rose, G.: *Formal Object-Oriented Specification using Object-Z. Cornerstones of Computing Series*. MacMillan Press, Basingstoke (2000)
26. Frappier, M., Habrias, H. (eds.): *Software Specification Methods: An Overview Using a Case Study*. ISTE (2006)
27. George, V., Vaughan, R.: Application of lightweight formal methods in Requirement Engineering1. *CrossTalk: The Journal of Defense Software Engineering* (January 2003), <http://www.stsc.hill.af.mil/crosstalk/2003/01/George.html>
28. Gurevich, Y., Kutter, P., Odersky, M., Thiele, L. (eds.): *ASM 2000. LNCS*, vol. 1912. Springer, Heidelberg (2000)
29. Harman, M., McMinn, P. (eds.): *Proceedings of Testing: Academic & Industrial Conference — Practice And Research Techniques (TAICPART)*, Windsor, United Kingdom, August 29–31, pp. 29–31. IEEE Computer Society Press, Los Alamitos (2006)
30. Henson, M.C., Reeves, S., Bowen, J.P.: Z logic and its consequences. *CAI: Computing and Informatics* 22(4), 381–415 (2003)
31. Hierons, R.M., Bowen, J.P., Harman, M. (eds.): *FORTEST 2008. LNCS*, vol. 4949. Springer, Heidelberg (2008)
32. Hinchey, M.G., Bowen, J.P. (eds.): *Applications of Formal Methods. Prentice Hall International Series in Computer Science* (1995)
33. Hinchey, M.G., Bowen, J.P.: *Industrial-Strength Formal Methods in Practice. FACIT series*. Springer, Heidelberg (1999)
34. Hinchey, M.G., Bowen, J.P., Rouff, C.: Introduction to formal methods. In: Rouff, C., Hinchey, M.G., Rash, J., Truszkowski, W., Gordon-Spears, D. (eds.) *Agent Technology from a Formal Perspective. NASA Monographs in Systems and Software Engineering*, vol. 2, pp. 25–64. Springer, Heidelberg (2006)



35. Hoare, C.A.R.: An axiomatic basic for computer programming. *Communications of the ACM* 12(10), 576–583 (1969)
36. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. *Journal of the ACM* 50(1), 63–69 (2003)
37. Hoare, S.T.: The ideal of verified software. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 5–16. Springer, Heidelberg (2006)
38. Hoare, C.A.R., Milner, R.: Grand challenges for computing research. *The Computer Journal* 48(1), 49–52 (2005)
39. ISO. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, ISO/IEC 13568 (2002)
40. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
41. Jones, C.B., O’Hearn, P.W., Woodcock, J.C.P.: Verified software: A grand challenge. *IEEE Computer* 39(4), 93–95 (2006)
42. Kapoor, K., Bowen, J.P.: A formal analysis of MCDC and RCDC test criteria. *Software Testing, Verification and Reliability* 15(1), 21–40 (2005)
43. Lightfoot, D.: *Formal Specification Using Z*, 2nd edn. Grassroots Series. Palgrave (2001)
44. Métayer, C., Abrial, J.-R., Voisin, L.: Event-B Language. Project IST-511599 RODIN (Rigorous Open Development Environment for Complex Systems), Deliverable 3.2, Public Document, May 31 (2005), <http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf>
45. Morris, L., Jones, C.B.: An early program proof by Alan Turing. *IEEE Annals of the History of Computing* 6(2), 129–143 (1984)
46. Owe, O., Krogdahl, S., Lyche, T. (eds.): *From Object-Orientation to Formal Methods*. LNCS, vol. 2635. Springer, Heidelberg (2004)
47. Randell, B.: Memories of the NATO software engineering conferences. *IEEE Annals of the History of Computing* 20(1), 51–54 (1998), <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/>
48. Saaltink, M.: The Z/EVES system. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) *ZUM 1997*. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
49. Schellhorn, G., Grandy, H., Haneberg, D., Möbius, N., Reif, W.: A systematic verification approach for Mondex electronic purses using ASMs. In: Abrial, J.-R., Glässer, U. (eds.) *Börger Festschrift*. LNCS, vol. 5115. Springer, Heidelberg (2009)
50. Schneider, S.: *The B-Method: An Introduction*. Cornerstones of Computing Series. MacMillan Press, Basingstoke (2001)
51. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology* 15(1), 92–122 (2006)
52. Spivey, J.M.: *The Z Notation: A Reference Manual*, 2nd edn. Prentice Hall International Series in Computer Science (1992), <http://spivey.orient.ox.ac.uk/~mike/zrm/>
53. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: [31]
54. Vilkomir, S.A., Bowen, J.P.: From MC/DC to RC/DC: Formalization and Analysis of Control-Flow Testing Criteria. *Formal Aspects of Computing* 18(1), 42–62 (2006)
55. Weiser, M.: Program slicing. *IEEE Transactions on Software Engineering* 10, 352–357 (1984)
56. Woodcock, J.C.P.: First steps in the verified software Grand Challenge. *IEEE Computer* 39(10), 57–64 (2006)
57. Woodcock, J., Freitas, L.: Z/Eves and the mondex electronic purse. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) *ICTAC 2006*. LNCS, vol. 4281, pp. 15–34. Springer, Heidelberg (2006)
58. Chaochen, Z., Hansen, M.R.: *Duration Calculus: A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. Springer, Heidelberg (2004)

# Author Index

- Boulmé, Sylvain 1  
Bowen, Jonathan P. 219  
Butler, Michael 78
- Cansell, Dominique 17, 78
- Evans, Neil 130
- Farahbod, Roozbeh 147, 170
- Gargantini, Angelo 33  
Gervasi, Vincenzo 147, 170  
Glässer, Uwe 147  
Glausch, Andreas 50  
Grandy, Holger 93  
Grant, Neil 130
- Haneberg, Dominik 93  
Hinchey, Michael G. 219
- Ifill, Wilson 130
- Kirchberg, Markus 65  
Kolovos, Dimitrios S. 204
- Leuschel, Michael 78
- Ma, Geroge 147  
Méry, Dominique 17  
Moebius, Nina 93  
Müller, Peter 187
- Paige, Richard F. 204  
Polack, Fiona A.C. 204  
Potet, Marie-Laure 1
- Reif, Wolfgang 93  
Reisig, Wolfgang 50  
Riccobene, Elvinia 33  
Ruskiewicz, Joseph N. 187
- Scandurra, Patrizia 33  
Schellhorn, Gerhard 93  
Schewe, Klaus-Dieter 65  
Schmidt, Peggy 111  
Schneider, Steve 130
- Thalheim, Bernhard 111  
Treharne, Helen 130
- Zhao, Jane 65