Lecture Notes in Computer Science 5238

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

Egon Börger Michael Butler
Jonathan P. Bowen Paul Boca (Eds.)

Abstract State Machines, B and Z

First International Conference, ABZ 2008
London, UK, September 16-18, 2008
Proceedings

Springer

Volume Editors

Egon Börger
Università di Pisa
Dipartimento di Informatica
Pisa, Italy
E-mail: boerger@di.unipi.it

Michael Butler
University of Southampton
School of Electronics and Computer Science
Highfield, Southampton, UK
E-mail: mjb@ecs.soton.ac.uk

Jonathan P. Bowen
London South Bank University
Faculty of BCIM
London, UK
E-mail: jpbowen@gmail.com

Paul Boca
London South Bank University
Faculty of BCIM
London, UK
E-mail: paul.boca@googlemail.com

Library of Congress Control Number: 2008934655

CR Subject Classification (1998): D.2.1, D.2.2, D.2.4, D.3.1, H.2.3, F.3.1, F.4.2-3

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

ISSN 0302-9743
ISBN-10 3-540-87602-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87602-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12518180 06/3180 5 4 3 2 1 0

Preface

The ABZ 2008 conference was held in London during September 16–18, 2008. The conference aimed at contributing to the cross-fertilization of three rigorous methods that share a common conceptual foundation and are widely used in both academia and industry for the design and analysis of hardware and software systems, namely, abstract state machines, B, and Z. It followed on from the Dagstuhl seminar on Rigorous Methods for Software Construction and Analysis, which was organized in May 2006 by Jean-Raymond Abrial (ETH Zürich, Switzerland) and Uwe Glässer (Simon Fraser University – Burnaby, Canada), and brought together researchers from the ASM and the B community (see: *http://www.dagstuhl.de/06191*).

The conference simultaneously incorporated the 15th International ASM Workshop, the 17th International Conference of Z Users and the 8th International Conference on the B Method, which were present with separate Program Committees to select the papers published in separate tracks (see Chapters 2–4 of these proceedings). The conference covered a wide range of research spanning from theoretical and methodological foundations to tool support and practical applications. It was split into three main parts:

- A one-day common program of four invited lectures, see Chap. 1 of these proceedings, and the presentation of three papers selected among the submitted contributions
- Two days of contributed research papers and short presentations of work in progress, of industrial experience reports and of tool demonstrations, as documented in Chap. 2–5 of these proceedings
- Two tutorials on the ASM and B simulator tools *CoreAsm* and *Pro-B*

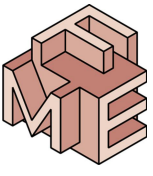
The conference was preceded by a UK EPSRC-funded Verified Software Repository Network (VSR-net) workshop on Monday, September 15, organized by Jim Woodcock and Paul Boca. This is reported in Chap. 6 of these proceedings, which includes an invited talk by Cliff Jones along with an overview of other technical talks at the VSR-net workshop covering progress on verification challenges.

We wish to thank the members of the three Program Committees and the numerous reviewers for their work, Springer for publishing the proceedings, and the sponsors for substantial financial support. In particular, the British Computer Society hosted the conference at their offices in central London, through the support of the BCS-FACS Specialist Group on Formal Aspects of Computing Science. Formal Methods Europe sponsored Wolfram Buettner's attendance, Nokia provided welcome financial sponsorship and two devices which were awarded as best-paper prizes, and Praxis High Integrity Systems sponsored the conference bags. The EPSRC VSR-net Network enabled the VSR-net workshop

to be held free for attendees. London South Bank University provided access to the Union Jack Club for convenient accommodation for conference delegates. The Easychair system was used for management of the submission and reviewing process.

Further information on the ABZ2008 conference may be found online at: *http://www.abz2008.org*

Egon Börger
Michael Butler
Jonathan P. Bowen
Paul Boca



Organization

Program Chairs

Paul Boca
Egon Börger
Jonathan P. Bowen
Michael Butler

Local Organization

Paul Boca

ASM Program Committee

Egon Börger (Chair)
Alessandra Cavarra
Andreas Friesen
Uwe Glaesser
Susanne Graf
Kristina Lundqvist
Andreas Prinz
Elvinia Riccobene
Klaus-Dieter Schewe
Anatol Slissenko
Jan Van den Bussche
Margus Veanes
Charles Wallace

B Program Committee

Christian Attiogbé
Richard Banach
Juan Bicarregui
Michael Butler (Chair)
Dominique Cansell
Daniel Dolle
Marc Frappier
Jacques Julliand
Regine Laleau
Michael Leuschel
Annabelle McIver

Dominique Mery
Louis Mussat
Marie-Laure Potet
Ken Robinson
Steve Schneider
Emil Sekerinski
Bill Stoddart
Elena Troubitsyna
Mark Utting

Z Program Committee

Jonathan P. Bowen (Chair)
John Derrick
Leo Freitas
Martin Henson
Mike Hinchey
Randolph Johnson
Yves Ledru
Steve Reeves
Mark Utting
Sergiy Vilkomir
Jim Woodcock

VSR Day Organizers

Paul Boca
Jim Woodcock (Chair)

External Reviewers

Roozbeh Farahbod
Frederic Gervais
Stefan Hallerstede
Thai Son Hoang
Angel Robert Lynas
Hassan Mountassir
Martin Ouimet
Abdolbaghi Rezazadeh
Patrizia Scandurra
Laurent Voisin

Table of Contents

Chapter 1. ABZ Invited Talks

Chapter 2. ASM Papers

Chapter 3. B Papers

Chapter 4. Z Papers

Chapter 5. ABZ Short Papers

Chapter 6. VSR Day

Complex Hardware Modules Can Now be Made Free of Functional Errors without Sacrificing Productivity

Wolfram Büttner

Prinzregentenstr. 74
81675 München, Germany
wolfram.buettner@email.de

Abstract. Complementary to the systems and software focus of the conference, this presentation will be about chips and the progress that has been made in their functional verification. Common ground will be high-level, still cycle-accurate, state-based models of hardware functionalities called Abstract RT. RT stands for register transfer descriptions of hardware such as VHDL or Verilog. An Abstract RT model is a formal specification which permits an automated formal comparison with its implementation, thus detecting any functional discrepancy between code and formal specification.

The first part of the presentation will sketch the big picture: Moore's Law still holds and permits building huge chips comprising up to hundreds of millions of gates. Under the constraints of shrinking budgets and development times, these so-called systems-on-chip (SoC) can no longer be developed from scratch but must largely be assembled from pre-designed, pre-verified design components such as processors, controllers, a plethora of peripherals and large amounts of memories. Therefore, getting a SoC right depends to a large extent on the quality of these design components – IP for short. At stake are critical errors making it into silicon. These may cost millions of Euros due to delayed market entry, additional engineering and re-production efforts. Hence, the lion's share of today's verification efforts goes into the functional verification of such IP.

Current functional verification practice for IP is based on simulation and employs a mix of handcrafted directed tests and automatically generated constrained random tests whose outcomes are compared with those obtained by running the tests through functionally equivalent C-models. Simulation traces are inspected by assertions and progress of verification is measured by coverage analysis. The underlying philosophy is largely black box testing: Test scenarios are developed from a specification and verification engineers intentionally avoid insight into details of a design. This practice has been around for almost 20 years and its high degree of automation has allowed putting much higher test loads on designs than what could be achieved with handcrafted tests only. It would be a miracle, though, if a basically black-box approach would reliably find the intricate errors that can hide in complex functionalities. While engineers have enjoyed the comfort of constrained random search over quite some time, they now start to feel the limitations of the approach: A confidence gap regarding detection of critical errors in deeply buried logic, the vast amount of hardware and software infrastructure required by the approach and the inability to adapt random tests driven by constraints to design changes other than by tweaking these constraints. Most importantly, the cost of verification of

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 1–3, 2008.

a complex hardware module or IP now by far exceeds the cost of designing it. Hence, the often quoted design gap, saying that chips keep growing much faster than the ability to fill them with functionality, is better described as a verification gap.

Therefore, it has been widely recognized in industry that effective verification of IP and hardware modules require that more design insight be made available to the verification process. There are first attempts to automatically extract some design insight from code and use it to link test scenarios to be analyzed with choosing effective tests – thus closing the loop that black-box testing lacks. However, the mainstream event for bringing more design insight into simulation-based verification has been the advent of standardized verification languages such as System Verilog Assertions (SVA). In practice, such a (designer) assertion captures local design intent of a designer and typically is written by him. The designer can simulate his assertions and – to some extent – formally verify them with a suitable model checker. While SVA brings about a big step forward, its current usage does not yet bridge the verification gap mentioned above, because the complex interaction of code fragments can not be captured by isolated designer assertions – think of instructions processed by pipelines which may conflict with newly inserted instructions.

This kind of interaction, however, is what makes functional verification so difficult. Real progress requires moving within the framework of SVA from partial models of IP capturing local design intent to models which capture full design intent –including interaction of functionalities. The key features of such a model are the assertions that it employs, its ability to compose such assertions and capabilities to check that the model indeed captures all behavior of the implementation of the IP. For any given IP, the assertions required to build the IP‘s model must formalize in an intuitive and compact fashion the operations that the IP can execute – such as read or write operations in its various modes or arithmetic instructions or many more operations. As any computation of the IP is the result of chaining suitable operations, respective assertions must provide information allowing a check that they connect seamlessly. Finally, algorithms that check that the IP‘s assertions indeed constitute a full model of the IP are required. At IP 07, I named such models “Abstract RT” and I believe this name fits well the intentions of this conference. Currently, the main application of Abstract RT for a given IP is an automated formal comparison with its implementation, thus detecting any functional discrepancy between code and formal specification. This equivalence check implements on a higher level what standard equivalence checking has been doing for many years to ensure correct design refinement from RT to gate level. Both comparisons yield best possible quality. But while the objects of the latter comparison are provided fully by the design process, the higher level comparison requires an additional activity, namely building an Abstract RT model.

The second part of the presentation will take a closer look at Abstract RT: A first example will show how to build the Abstract RT model of a small controller and provide an intuitive understanding of Abstract RT. This will be followed by a description of the various tasks required to build and verify Abstract RT for a given IP. These tasks fall into three categories:

Identifying operations from specification and code, formalizing these operations by suitable assertions and then ensuring compliance regarding code and specification. The first can be done with bounded property checking techniques, while the latter amounts to a manual review.

Grouping suitable operations in single processes each describing 'interesting' subfunctionalities. The central verification task here must ensure that any behavior of such a process is captured as a sequence of operations. Upon completion of certain checks, such a process is called complete in the terminology of OneSpin Solutions GmbH holding respective patents.

The full functionality of the design then is a collection of complete processes consisting of correct operations. Each of these processes has constraints regarding its legal inputs and makes assertions about its outputs. It remains to check whether the network of these constraints and assertions creates no additional behavior to that of the (isolated) processes.

Writing 'good' RT code is a creative and challenging activity and the same holds for writing Abstract RT. Methodology, therefore, is key for guiding practitioners to become good 'Abstract RT designers'.

The third part of the presentation considers the perspectives of Abstract RT: Proven Abstract RT is a compact, functionally equivalent model of the implementation of some IP. As has been mentioned above, chip designs largely emerge by connecting IP's. System exploration and verification is too slow if performed on the RT representation of the chip. Rather, one wishes to rapidly simulate on higher levels still capturing the chips full functionality. A chip model composed of Abstract RT models of the chips IP's could provide such a simulation model with the added benefit that 'what you simulate is what you provably get'.

Probably the largest impact of Abstract RT would be that of a platform interleaving specification, implementation and verification. Respective speculations will conclude the presentation.

Keywords: Design Process, Formal Verification, Abstract State Machine.

The High Road to Formal Validation:
Model Checking High-Level Versus Low-Level Specifications

Michael Leuschel

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

Abstract. In this paper we examine the difference between model checking high-level and low-level models. In particular, we compare the ProB model checker for the B-method and the Spin model checker for Promela. While Spin has a dramatically more efficient model checking engine, we show that in practice the performance can be disappointing compared to model checking high-level specifications with ProB. We investigate the reasons for this behaviour, examining expressivity, granularity and Spin's search algorithms. We also show that certain types of information (such as symmetry) can be more easily inferred and exploited in high-level models, leading to a considerable reduction in model checking time.

Keywords: B-Method, Tool Support, Model Checking, Symmetry Reduction, Spin [1].

1 Introduction

Model checking [11] is a technique for validating hardware and software systems based on exhaustively exploring the state space of the system. Model checking has been hugely successful and influential, culminating in the award of the Turing Prize to Clarke, Emerson and Sifakis.

Most model checking tools work on relatively low-level formalisms. E.g., the model checker smv [9, 29] works on a description language well suited for specifying hardware systems. The model checker Spin [5, 19, 21] accepts the Promela specification language, whose syntax and datatypes have been influence by the programming language C. Recently, however, there have also been model checkers which work on higher-level formalisms, such as ProB [24, 27] which accepts B [1]. Other tools working on high-level formalisms are, for example, fdr [16] for CSP and alloy [23] for a formalism of the same name (although they both are strictly speaking not model checkers).

[1] This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 4–23, 2008.

It is relatively clear that a higher level specification formalism enables a more convenient modelling. On the other hand, conventional wisdom would dictate that a lower-level formalism will lead to more efficient model checking. In this paper we try to establish that this is not necessarily the case; sometimes it can even be considerably more efficient to directly validate high-level models.

In this paper we concentrate on comparing two formalisms: the high-level B-method and the more low-level specification language Promela, as well as the associated model checkers ProB and Spin.

Indeed, Spin is one of the most widely used model checkers. Spin is an impressive feat of engineering and has received the ACM System Software Award in 2001. A lot of research papers and tools [3, 7, 10, 18, 31, 32, 35] translate other formalisms down to Promela and use Spin. Against Spin we pit the model checker ProB, which directly works on a high-level specification language,[2] but has a much less tuned model checking engine.

In Section 2 we first examine the granularity and expressivity of both formalisms, and explain what can go wrong when translating a high-level formalism down to Promela. In Section 3 we examine the problems that can arise when using Spin on systems with a very large state space (such as typically encountered when model checking software). In Section 4, we look at one piece of information (symmetry) that can be more easily extracted from high-level models, and its impact on the performance of model checking.

2 Granularity and Expressivity

B being at a much higher-level than Promela, it is clear that a single B expression can sometimes require a convoluted translation into Promela. As an example, take the following B operation to sort an array `a` of integers:

```
Sort = ANY p WHERE p:perm(dom(a)) &
                   !(i,j).(i:1..(size(a)-1) & j:2..size(a) & i<j
                            =>  a(p(i)) <= a(p(j))) THEN
          a := (p;a) END;
```

The sorting operation is specified as choosing any permutation `p` such that after applying the permutation the array is sorted. Note that `(p ; a)` corresponds to relational composition, and `(p;a)(i) = a(p(i))`. Promela does not have such a powerful non-deterministic construct. Basically, sorting will have to be encoded in Promela by actually implementing a sorting algorithm, which will take up considerably more space and time to write than the above B specification (see, e.g., the merge sort example accompanying the book [4]).

But even less extreme examples are sometimes surprisingly difficult to model in Promela. Take, e.g., the B statement `x::1..n`, which non-deterministically sets `x` to a value between 1 and n. In Promela, we have to encode this using an if-statement as follows:

[2] See [2] which argues that formal verification tools should tie directly to high-level design languages.

```
if
  :: x=1
  :: x=2
  (...)
  :: x=n
fi
```

This translation is not very concise, and moreover can only be performed if we statically know the value of n. In case n is a variable, we have to encode `x::1..n` in Promela as follows (see, Section 4.6.2 in [5]):

```
x = 1;
do
  :: (x<nn) -> x++
  :: ((x>=1)&&(x<=nn)) -> break
od
```

This translation is more concise (for larger n) than using an if-statement, but still adds unnecessary state and behaviours to the model. Figure 1 shows on the left the behaviour of the B model, starting from a state where the variable `x` has the value 2 and supposing that $n = 4$. On the right we see the behaviour of our Promela translation, which has $n = 4$ additional intermediate states.

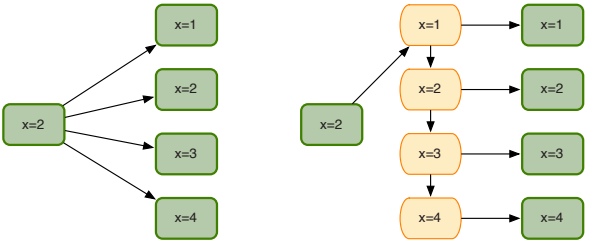


Fig. 1. B and Promela state space for non-deterministic choice `x::1..n` with $n = 4$

In [21] on page 462 another translation is suggested, which produces a more random distribution when animating. However, this translation has the disadvantage that the `do` loop may not terminate (which can cause a problem with verification; see below).

The situation depicted in Figure 1 is actually quite typical: by translating a specification from B to Promela we add new intermediate states and additional internal behaviour. As another example, take the B predicate `x:ran(f)`, where `f` is a total function.[3] In Promela, we could encode the function as an array, supposing for simplicity that the domain of the function is $0..n-1$ for some n. Evaluating the predicate `x:ran(f)` again requires a loop in Promela:

[3] This is actually taken from our small case study in Section 4.2.

```
byte i = 0;
do
  :: (i<n && f[i]!=x) -> i++
  :: (i<n && f[i]==x) -> break    /* not(x:ran(f)) */
  :: (i==n) -> break              /* x:ran(f) */
od;
```

Again, additional intermediate states appear in the Promela model. Note, that in the presence of concurrency, these additional states can quickly lead to a blow-up in the number of states compared to the high-level model. E.g., if we run four copies of the B machines from Figure 1 in parallel, we get $5^4 = 625$ states. If we run four copies of the Promela specification from Figure 1 in parallel, we get $9^4 = 6561$ states.

A similar picture arises for the `x:ran(f)` example. Supposing we have $n = 7$ and that we have 4 processes, the B model would have, for a given function f, $2^4 = 16$ states (before and after evaluating the predicate). The Promela model will have in the worst case (where the predicate `x:ran(f)` is false for all processes) $10^4 = 10,000$ states. Also, we have the problem that the new variable i is now part of the state. In the Promela model we should ensure that i is reset to some default value after being used; otherwise a further blow-up of the state space will ensue.

Luckily, Promela provides the `atomic` construct, which can be used to avoid the additional blow-up due to internal states. However, the `atomic` construct does have some restrictions (e.g., in the presence of channel communications). Also, care has to be taken when placing an `atomic` around `do` loops: if the loop does not necessarily terminate, then exhaustive verification with SPIN will become impossible (the "search depth too small" error will be systematically generated). If by accident the loop does not terminate at all, as in the following example, then SPIN itself will go into an infinite loop, without generating an error message or warning:

```
atomic{ do
        :: x<10 -> x++
        :: x>0 -> x--
    od }
```

In summary, translating high-level models into Promela is often far from trivial. Additional intermediate states and additional state variables are sometimes unavoidable. Great care has to be taken to make use of `atomic` (or even `dstep`) and resetting dead temporary variables to default values. However, restrictions of `atomic` make it sometimes very difficult to hide all of the intermediate states.

As far as an automatic translation is concerned, it is far from clear how B could be automatically translated into Promela;[4] it is actually already very challenging to write an interpreter in Prolog for B [24, 27].

[4] This process was actually attempted in the past — without success — within the EPSRC funded project ABCD at the University of Southampton.

A Small Empirical Study

[36] studies the elaboration of B-models for ProB and Promela models for SPINon ten different problems. With one exception (the Needham-Schroeder public key protocol), all B-models are markedly more compact than the corresponding Promela models. On average, the Promela models are 1.85 longer (counting the number of symbols). The time required to develop the Promela models was about 2-3 times higher than for the B models, and up to 18 times higher in extreme cases. No model took less time in Promela. Some models could not be fully completed in Promela. The study also found that in practice both model checkers ProB and Spin were comparable in model checking performance, despite ProB working on a much higher-level input language and being much slower when looking purely at the number of states that can be stored and processed. In the remainder of this paper, we will investigate the possible reasons for this surprising fact, which we have ourselves witnessed on repeated occasions, e.g., when teaching courses on model checking or conducting case studies.

3 Searching for Errors in Large State Spaces

In this section we will look at a simple problem, with simple datatypes, which can be easily translated from B to Promela, so that we have a one-to-one correspondence of the states of the models. In such a setting, it is obvious to assume that the Spin model checker for Promela will outperform the B model checker by several orders of magnitude. Indeed, Spin generates a specialised model checker in C which is then compiled, whereas ProB uses an interpreter written in Prolog. Furthermore, Spin has accrued many optimisations over the years, such as partial order reduction [22, 30] and bitstate hashing [20]. (ProB on its side does have symmetry reduction; but we will return to this issue in the next section.)

The simple Promela example in Fig. 2 can be used to illustrate this speed difference. The example starts with an array in descending order, and then allows permutation at some position i, with i cycling around the array. The goal is to reach a state where the array starts with `[1,2,3]`. On a MacBook Pro with a 2.33 GHz Core2 Duo, Spin (version 4.2.9 along with XSpin 4.28) takes 0.00 seconds to find a solution for the Promela model (plus about six seconds compilation time on XSpin and 40 seconds to replay the counter example which is 1244 steps long). ProB (version 1.2.7) takes 138.45 seconds for the same task on an equivalent B model. This, however, includes the time to display the counter example, which is in addition also only 51 steps long. Still, the model checking speed is dramatically in favour of Spin (and the difference increases further when using larger arrays).

However, it is our experience that this potential speed advantage of Spin often does not translate into better performance in practice in real-life scenarios. Indeed—contrary to what may be expected—we show in this section that Spin sometimes fares quite badly when used as a debugging tool, rather than as verification tool. Especially for software systems, verification of infinite

```
#define SZ 8
active proctype flipper () {
    byte arr[SZ];  byte i,t;
    do
      :: i==SZ -> break
      :: i<SZ -> arr[i] = SZ-i; i++
    od;
    i = 0;
    do
      :: i<SZ-1 -> i++
      :: i==SZ-1 -> i=0
      :: i<SZ-1 -> t = arr[i]; arr[i]=arr[i+1]; arr[i+1]=t; t=0; i++
      :: (arr[0]==1 && arr[1]==2 && arr[2]==3)
          -> assert(false) /* found solution */
    od
}
```

Fig. 2. Flipping adjacent entries in an array in Promela

state systems cannot be done by model checking (without abstraction). Here, model checking is most useful as a debugging tool: trying to find errors in a very large state space.

3.1 An Experiment

Let us model a simple ticket vending machine, each ticket costing five euros. Those tickets can either be paid using a credit card or with coins. If no more tickets are available the machine should no longer accept coins or credit cards. Figure 3 depicts a low-level Promela specification of this problem. For simplicity, the machine requires the user to insert the exact amount (i.e., no change is given) and we have not yet specified a button which allows a user to recover inserted money. In the model we have also encoded that before issuing a ticket (via `ticket--`), the number of available tickets is greater or equal than 1.

Figure 3 actually contains an error: the credit card number (for simplicity always 1) is not reset after a ticket has been issued. This can lead to an assertion violation. An equivalent specification in the high-level B formalism is depicted in Figure 4. The same error is reproduced there, leading to an invariant violation.

Both models can also deadlock, namely when all tickets have been issued. Both problems have been fixed in adapted models. In the Promela version the `cardnr` variable is now reset to 0 after being used and the following line has been added as the last branch of the `do` loop:

```
:: (ticket==0) -> ticket = 2 /* reload ticket machine */
```

Similarly, in the B model the `withdraw_ticket_from_card` has been corrected to reset the `cardnr` and the following operation has been added:

```
resupply = PRE ticket = 0 THEN ticket := 2 END;
```

```
active proctype user () {
 byte c10 = 0; byte c20 = 0; byte c50 = 0;
 byte c100 = 0; byte c200 = 0; byte cardnr = 0;
 byte ticket = 2;
 do
    :: (cardnr==0 && ticket>0) -> c10++
    :: (cardnr==0 && ticket>0) -> c20++
    :: (cardnr==0 && ticket>0) -> c50++
    :: (cardnr==0 && ticket>0) -> c100++
    :: (cardnr==0 && ticket>0) -> c200++
    :: (c10+c20+c50+c100+c200==0 && ticket>0) -> cardnr = 1
    :: ((c10+2*c20+5*c50+10*c100+20*c200)==500)
          -> assert(ticket>0);
             atomic{ticket--; c10=0; c20=0; c50=0; c100=0; c200=0}
    :: (cardnr>0) ->  assert(ticket>0); ticket--
         /* forgot to reset cardnr */
 od
}
```

Fig. 3. A nasty ticket vending machine in Promela

```
MACHINE NastyTicketVending
DEFINITIONS SET_PREF_MAXINT == 255
VARIABLES
  c10,c20,c50,c100,c200, cardnr, ticket
INVARIANT
  c10:NAT & c20:NAT & c50:NAT & c100:NAT & c200:NAT & cardnr:NAT & ticket:NAT
INITIALISATION
  c10,c20,c50,c100,c200, cardnr, ticket := 0,0,0,0,0, 0,2
OPERATIONS
  insert_10cents = PRE cardnr=0 & ticket>0 THEN c10 := c10 + 1 END;
  insert_20cents = PRE cardnr=0 & ticket>0 THEN c20 := c20 + 1 END;
  insert_50cents = PRE cardnr=0 & ticket>0 THEN c50 := c50 + 1 END;
  insert_100cents = PRE cardnr=0 & ticket>0 THEN c100 := c100 + 1 END;
  insert_200cents = PRE cardnr=0 & ticket>0 THEN c200 := c200 + 1 END;
  insert_card = PRE c10+c20+c50+c100+c200=0 & ticket>1 THEN cardnr := 1 END;
  withdraw_ticket_from_coins = PRE c10+2*c20+5*c50+10*c100+20*c200=50  THEN
    c10,c20,c50,c100,c200, cardnr, ticket := 0,0,0,0,0, 0,ticket-1
  END;
  withdraw_ticket_from_card = PRE cardnr>0 THEN
     ticket := ticket -1  /* forgot to reset cardnr */
  END
END
```

Fig. 4. A nasty ticket vending machine in B

Both models have also been enriched with an additional constraint, namely that not more than 70 coins should be inserted at any one time. The models do not yet ensure these constraints, hence the model checkers should again uncover assertion violations.

We now compare using SPIN (version 4.2.9 along with XSpin 4.28) on the Promela model against using PROB (version 1.2.7) on the B model. All tests were again run on a MacBook Pro with a 2.33 GHz Core2 Duo.

The original Promela specification from Fig. 3 actually also had an additional, unintentional error: it contained `assert(ticket>=0)` instead of `assert(ticket>0)` for the last branch of the `do` loop. This surprisingly meant that SPIN could not find an assertion violation, as bytes are by definition always positive ($0-1=255$ for bytes in Promela).

After fixing this issue, a series of experiments were run using SPIN. The results can be found in Table 1. The model checking times are those displayed by SPIN (user time), and do not include the time to generate and compile the `pan` file (which takes about 6 seconds with XSpin). The very first line shows the use of SPIN on the model from Fig. 3, when used in default settings. As one can see, it took SPIN 40 seconds before aborting with an "out of memory" error. No counter-example was found. After that we successively adapted various of the (many) SPIN's parameters. It can be seen that even with bitstate hashing enabled, no error was detected. However, after turning on breadth-first search, an assertion violation was finally found.

We now turned to the second model, where the two problems of the first model were corrected. We started off with the setting that was successful for the first model; but this time this setting proved incapable of detecting the new error in the second model. Only after reverting back to a depth-first search was an assertion violation detected. (Note that it took XSpin several minutes to replay the counter example containing over 65000 steps.)

In summary, for the first deadlocking model (Fig. 3) it took us about 1000 seconds of CPU time and an afternoon to realise that the initial version of the model was wrong. After that it took us about 800 seconds (adding up the various

Table 1. SPIN experiments on the nasty vending machine

Search Depth	Memory MB	Partial Order	Bitstate Hashing	Breadth First	Time (sec)	Result
Deadlocking model from Fig. 3						
10,000	128	yes	no	no	40.00	out of memory †
10,000	512	yes	no	no	580.26	out of memory †
10,000	512	yes	yes	no	89.59	†
100,000	512	yes	yes	no	91.51	†
1,000,000	512	yes	yes	no	97.04	†
100,000	512	yes	yes	yes	0.00	error found
Non-deadlocking model with ticket resupply						
100,000	512	yes	no	yes	64.26	out of memory
100,000	512	yes	yes	yes	47.23	out of memory
100,000	512	yes	yes	no	0.17	error found

† = search depth too small.

runs of SPIN in the upper half of Table 1) of CPU time and 45 minutes in total to locate an error in the model.[5] For the equivalent high-level B specification in Fig. 4, PROB took 0.2 seconds to find an invariant violation (with default settings). The counter-example consisted of 4 steps: one `insert_card`, followed by 3 `withdraw_ticket_from_card` events.[6]

For the non-deadlocking model, it took us in all about 111 seconds of CPU time and three attempts to uncover the error with SPIN. For the equivalent B model, PROB takes 24 seconds to find the invariant violation, again in default settings. Observe that the counter example consists of the minimally required 70 steps; SPIN's counter example consists of over 65000 steps.

3.2 Explanation of the Experimental Results

What can explain this poor performance of SPIN compared to PROB? The specification is quite simple, and the Promela and B models are very similar in size and complexity. On the technology side, SPIN compiles the Promela models to C and can crunch hundreds of thousands of states per second. PROB uses a Prolog interpreter to compute the state space of the B specification. SPIN uses partial order reduction, PROB does not (and symmetry does not apply here).

Let us first examine the characteristics of the models. The deadlocking model has a very large state space, where there is a systematic error in one of the operations of the model (as well as a deadlock when all tickets have been withdrawn). To detect the error, it is important to enable this operation and then exercise this operation repeatedly. It is not important to generate long traces of the system, but it is important to systematically execute combinations of the individual operations. This explains why depth-first behaves so badly on this model, as it will always try to exercise the first operation of the model first (i.e., inserting the 10 cents coin). Note that a very large state space is a typical situation in software verification (sometimes the state space is even infinite).

In the corrected non-deadlocking model the state space is again very large, but here the error occurs if the system runs long enough; it is not very critical in which order operations are performed, as long as the system is running long enough. This explains why for this model breadth-first was performing badly, as it was not generating traces of the system which were long enough to detect the error.

In order to detect both types of errors with a single model checking algorithm, PROB has been using a mixed depth-first and breadth-first search [27]. More precisely, at every step of the model checking, PROB randomly chooses between a depth-first and a breadth-first step. This behaviour is illustrated in Fig. 5,

[5] This shows that one should be very careful about experimental results for a tool with many parameters: if only the successful runs get published (i.e., experiment 6 in Table 1 for the deadlocking model) the reader can get a very misleading picture of the real-life performance of a tool, as all the time and expertise required to tune the parameters is ignored.

[6] Depending on the run, a deadlock can also be found. We return to this later.

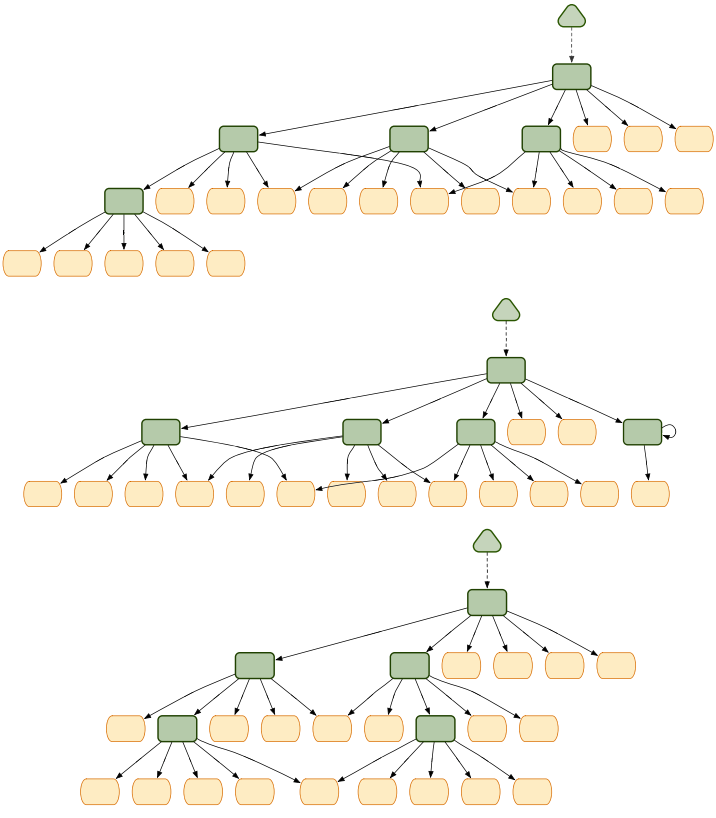


Fig. 5. Three different explorations of ProB after visiting 5 nodes of machine in Fig. 4

where three different possible runs of ProB are shown after exploring 5 nodes of the B model from Fig. 4.

The motivation behind ProB's heuristic is that many errors in software models fall into one of the following two categories:

- Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a model, this kind of error is very common.
- Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

One may ask whether the random component of ProB's algorithm can lead to large fluctuations in the model checking time. Figure 6 shows the result of a small experiment, where we have timed 16 runs for the above deadlocking machine from Fig. 4. The average runtime was 0.46 seconds, the standard deviation was

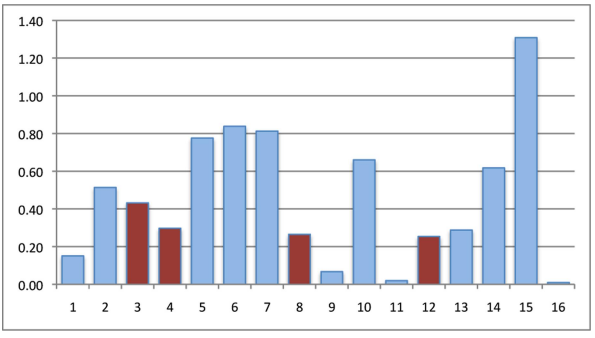


Fig. 6. 16 Runtimes for model checking Fig. 4; a deadlock was found in runs 3,4,8,12

0.36. As can be seen, in all cases an error was found reasonably quickly, the worst time being 1.31 seconds.

In summary, if the state space is very large, SPIN's depth-first search can perform very badly as it fails to systematically test combinations of the various operations of the system. Even partial order reduction and bitstate hashing do not help. Similarly, breadth-first can perform badly, failing to locate errors that require the system to run for very long. We have argued that PROB's combined depth-first breadth-first search with a random component does not have these pitfalls.

The aspect we have discussed in this section does not yet show a fundamental difference between model checking high-level and low-level models. Indeed, recently there has been considerable interest in directed model checking, using informed search strategies with heuristic functions such as A* or best-first-search, see, e.g. [37] leading to [14, 15] in the context of Promela. However, for a low level formalism, one is probably much more reluctant to adopt those techniques as they may noticeably slow down the verification when not needed. (Indeed, the above mentioned techniques have not yet found their way into the official distribution of SPIN.) For high-level models, the overhead of adopting a more intelligent search algorithm is less pronounced, as processing individual states takes up considerably more time. Hence, there is scope for considerably more refined search algorithms when model checking high-level models.[7]

4 Exploiting Symmetry in High-Level Models

In the previous section we encountered a scenario were complete verification was impossible (as the state space was too large), and model checking was used as a debugging tool. In this section we return to the verification scenario (although the points should also be valid for a debugging scenario), and show that even there it can be much more efficient to model check a high-level model than a low-level one. The reason is that in the high-level model certain properties, such

[7] Within the DFG-funded project GEPAVAS we are actually investigating adding heuristics when model checking B specifications.

as symmetries, can be detected much more easily than in a low-level model. Exploiting those properties in the model checker then leads to a considerably reduced state space.

For example in B, symmetries are induced by the use of deferred sets, as every element of such a set can be substituted for every other element [26]. The use of deferred sets is very common in B, and hence many B models exhibit symmetry which can be exploited by ProB [26, 28, 33, 34].

4.1 First Experiment: Scheduler

For a first experiment we have used the scheduler1.ref machine from [25] (also used in [26] and contained in Appendix B). Here `PROC` is a deferred set (of process identities) and the translation can be found in Appendix A. Comparing different tools and formalisms is always a difficult task. We have obtained the help of Promela/Spin experts to construct a faithful Promela counterpart of the B model. We have taken extreme care to translate this B specification into the best Promela code possible (with the help of Alice Miller and Alastair Donaldson) and have ensured that both models exhibit the same number of states (in the absence of symmetry reduction and partial order reduction). Also, the example is not artificially chosen or constructed so as to make our tool behave better. It was chosen because it was a relatively simple B model, that could still be hand translated with reasonable effort into an equivalent Promela model (it also has the property that symmetry in the B model can be translated into symmetry of process identifiers in the Promela model; something which will no longer be true for more complicated models, e.g., from [28] or the model we present later in Section 4.2).

Still, this is just one example and we do not claim that the phenomena uncovered here is universal. Indeed, as we have already seen in Section 3, if one takes a B model with no deferred sets (and hence no symmetry exploitable by ProB), then complete verification with Spin will be substantially faster than our tool (provided the B model can be translated into Promela). But our intuition is that, for some application domains, working at a higher level of abstraction (B vs. Promela) can be beneficial both for the modelling experience of the user (less effort to get the model right) and for the model checking effort.

In the experiments below, we have used the same setup as before in Section 3. This time we have incorporated the ProB and Spin results in the single Table 2. To exploit the symmetry in ProB we have used the technique from [33] based on the nauty graph canonicalisation package. In addition to timings reported by the two tools, we have also used a stopwatch to measure the user experience (these timings were rounded to the nearest half second). For Spin, default settings were used, except where indicated by the following symbols: $c1$ means compression (c1) was used, $\odot$ meaning bitstate hashing (DBITSTATE) was used, $1GB$ means that the allocated memory was increased to 1 GB of RAM, $>$ signifies that the search depth was increased to 100,000, $\gg$ that the search depth was increased to 1,000,000, and $\ggg$ that search depth was increased to 10,000,000. The ProB time includes time to check the invariant. Only deadlock and invalid end state checking is performed in Spin.

Table 2. Experimental results for scheduler1

Card	Tool	States	Time	Stopwatch
2	ProB	17	0.04 s	< 0.5 s
	ProB + nauty	10	0.03 s	< 0.5 s
	Spin (default)	17	0.02 s	6 s
4	ProB	321	1.08 s	1.5 s
	ProB+ nauty	26	0.15 s	< 0.5 s
	Spin (default)	321	0.00 s	6 s
8	ProB+ nauty	82	1.18 s	1.5 s
	Spin (default)	† 483980	2.50 s	6.5 s
	Spin ($> c1$)	† 545369	5.84 s	11 s
	Spin ($\gg$)	595457	3.75 s	6 s
	Spin ($\gg c1$)	595457	7.47 s	11 s
12	ProB+ nauty	170	4.90 s	5.5 s
	Spin ($\gg c1$)	†1.7847e+06	17.92 s	22 s
	Spin ($\ggg c1$ 1GB)	$\propto$ † 1.1877e+07	135.60 s	140 s
	Spin ($\ggg c1$ 1GB $\odot$)	† 4.0181e+07	295.71 s	302 s

† = search depth too small, $\propto$ = out of memory.

One can observe that for 2 and 4 processes ProB with symmetry reduction is actually quite competitive compared to Spin with partial order reduction, despite the much higher-level input language. Furthermore, if we look at the total time taken to display the result to the user measured with a stopwatch, ProB is faster (for Spin there is the overhead to generate and compile the C code). For 8 processes, ProB is about three times faster than Spin. Note that it took us considerable time to adjust the settings until Spin was able to fully check the model for 8 processes.[8] For 12 processes we were unable to exhaustively check the model with Spin, even when enabling bitstate hashing. Attempts to further increase the search depth led to "command terminated abnormally." ProB checked the model for 12 processes in 5.5 seconds.

Of course, in addition to partial order reduction, one could also try and use symmetry reduction for Spin, e.g., by using the SymmSpin tool [6] or TopSpin tool [13]. To use TopSpin a minor change to the Promela model is required, after which the model with 8 processes can be verified in 0.09 s with combined partial order reduction and symmetry (not counting the compilation overhead). However, compared to ProB's approach to symmetry, we can make the following observations:

1. In Promela the user has to declare the symmetry: if he or she makes a mistake the verification procedure will be unsound; (there is, however, the work [12] which automatically detects some structural symmetries in Promela). In B symmetries can be inferred automatically very easily.

[8] The development of the model itself also took considerable time (and several email exchanges with Alice Miller and Alastair Donaldson); the first versions exhibited much worse performance when used with Spin.

2. Symmetry is much more natural and prevalent in B and ProB can take advantage of partial symmetries (see, e.g., the generic dining philosophers example in [28]) and one can have multiple symmetric types (for Spin typically only a single scalarset, namely the process identifiers, is supported). In TopSpin all processes must be started in an atomic block; in B constants can be used to assign different tasks or behaviours to different deferred set elements; the partial symmetries can still be exploited.
3. Using and installing the symmetry packages for Spin is not always straightforward (the packages patch the C output of Spin). TopSpin cannot as of now be downloaded.

To further illustrate point 2, we now show a model with multiple deferred sets. To the best of our knowledge, this model cannot be put into a form so that TopSpin can exploit the symmetries.

4.2 Second Experiment: Multiple Symmetric Datatypes

Figure 7 contains a small B model of a server farm, with two deferred sets modelling the users and the servers. Individual servers can connect and disconnect from the system, and the system routes user requests to an available server via the `UserRequest` operation, trying to maintain the same server for the same user on later occasions (unless a timeout occurs). The time to develop and model check the model with ProB was about 10 minutes.

```
MACHINE ServerFarm
SETS USERS;SERVER
VARIABLES active, serving
INVARIANT active <: SERVER & serving: active >+> USERS
INITIALISATION active,serving := {},{}
OPERATIONS
  ConnectServer(s) = PRE s:SERVER & s/: active THEN
      active := active \/ {s} END;
  DisconnectServer(s) = PRE s:SERVER & s:active THEN
    active := active - {s} || serving := {s} <<| serving END;
  s <-- UserRequest(u) =  PRE u:USERS THEN
    IF u:ran(serving) THEN
       s := serving~(u)
    ELSE
       ANY us WHERE us:SERVER & us : active & us /: dom(serving) THEN
           s:= us || serving(us) := u END
    END
  END;
  UserTimeout(u) = PRE u:USERS & u:ran(serving) THEN
     serving := serving |>> {u} END
END
```

Fig. 7. A server farm model in B

```
chan connect = [0] of { byte } ;
chan disconnect = [0] of { byte } ;
chan request = [0] of { byte } ;
#define SERVERS 8
#define USERS 8

active[SERVERS] proctype server () { /* pids from 0.. SERVERS-1 */
   do
     :: connect!_pid -> disconnect!_pid
   od
}
active[USERS] proctype user () { /* pids start at SERVERS */
   do
     :: request!_pid
   od
}
active proctype mserver () {
    bit sactive[SERVERS];
    byte serving[SERVERS];
    byte x = 0;
    do
      :: atomic{connect?x -> assert(sactive[x]==0) ->
          assert(serving[x]==0) -> sactive[x]=1 ->x=0}
      :: atomic{disconnect?x -> assert(sactive[x]==1) ->
          sactive[x]=0 -> serving[x]=0-> x=0}
      :: atomic{request?x;
        byte i = 0;
        do
          :: (i<SERVERS && serving[i]!=x) -> i++
          :: (i<SERVERS && serving[i]==x) -> break
          :: (i==SERVERS) -> break
        od;
        if
          :: (i==SERVERS) -> i=0 -> do
             :: (i<SERVERS && (sactive[i]==0 || serving[i]!=0)) -> i++
             :: (i<SERVERS && sactive[i]!=0 & serving[i]==0)
                    -> serving[i] = x -> break
             :: (i==SERVERS) -> printf("no server available") -> break
            od;
          :: (i<SERVERS) -> printf("already connected")
        fi;
        x = 0;i=0 /* reset x,i to avoid state explosion */
        }
    od
 }
```

Fig. 8. A server farm model in Promela

Table 3. SPIN on the server farm from Fig. 8

Card	Search Depth	Memory MB	Partial Order	Bitstate Hashing	Breadth First	Time (sec)	Result
6	100,000	512	yes	no	no	1.74	†
	1,000,000	512	yes	no	no	2.26	ok
7	1,000,000	512	yes	no	no	22.71	†
	10,000,000	512	yes	no	no	32.21	ok
8	10,000,000	512	yes	no	no	82.05	†
	100,000,000	512	yes	no	no	-	error
	10,000,000	512	yes	yes	no	279.37	†
	100,000,000	512	yes	yes	no	-	error

† = search depth too small; err = "command terminated abnormally".

Figure 8 contains a Promela version of the same problem. The Promela model was actually simplified: the second do loop always takes the first available server rather than non-deterministically choosing one.[9]

It took about one hour and a half until we had a model which could be checked for cardinality 6 (local variables had to be reset to 0; errors had crept up in the loops to find server machines, etc.). In Table 3 we show the results of our experiments with SPIN. Observe that for cardinality of 8 we did not manage to verify the model using SPIN. Complete model checking with PROB for the same cardinality takes 3.48 seconds with nauty and 0.77 seconds with the hash marker method from [28]. For a cardinality of 9, PROB takes 21.04 seconds with nauty and 1.16 seconds with the hash marker method.

It may be possible to further improve the Promela model (but note that we have already put about 10 times the effort into the Promela model than into the B model). In summary, the symmetry that can be inferred in the high-level model again leads to a dramatic reduction in model checking time.

5 Conclusion

SPIN is an extremely useful and very efficient model checking tool. Still, over the years, we have accumulated a certain amount of anecdotal evidence which shows that using a model checker for high-level models can quite often give much better results in practice. In this paper we have investigated the reasons for this counterintuitive behaviour.

In Section 2 we have studied the granularity and expressivity of Promela versus B, and have shown that the Promela counterpart of a B model may have a large number of additional internal states. If those internal states are

[9] One solution would be not to force the loop to chose the first available server. However, to avoid deadlocks, one should then also allow decrementing `i`. But then Spin will never be able to exhaustively check the model, unless we remove the `atomic` surrounding the loop. I.e., the proper solution would be to adapt the data structure, e.g., remembering also how many servers are still available.

not hidden using Promela's `atomic` construct, an explosion of the state space can ensue. Seasoned Promela users will not fall into this trap, but especially newcomers and students are likely to encounter this problem.

Another reason, which we have examined in Section 3, is that SPIN's fast but naive depth-first search fares very badly in the context of debugging systems with a very large (or infinite) state space. The mixed depth-first and breadth-first strategy of PROB can give much better results in such a setting.

Finally, in Section 4, we have shown that by exploiting symmetries in a high-level model, the model checking time can be dramatically reduced. For two examples, the PROB model checker performs verification in substantially less time than SPIN with partial order reduction and bitstate hashing.

Looking to the future, we believe there is a big potential for applying more intelligent model checking techniques to high-level formalisms. In particular, we believe that the potential for techniques such as heuristics-directed or parallel model checking is much more pronounced for a high-level formalism such as B than for a low-level formalism such as Promela.

In conclusion, due to the inherent exponential blow-up of the state space, it is often not that relevant whether a model checking tool can treat 100,000 or 10,000,000 states; it can be much more important how cleverly the tool treats those states and whether it can limit the exponential blow-up through techniques like symmetry reduction.

Acknowledgements. We would like to thank Alastair Donaldson, Alice Miller, Daniel Plagge, Harald Wiegard, and Dennis Winter for insightful comments and contributions to this paper.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Arvind, N.D., Katelman, M.: Getting formal verification into design flow. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 12–32. Springer, Heidelberg (2008)
3. Basin, D.A., Friedrich, S., Gawkowski, M., Posegga, J.: Bytecode model checking: An experimental analysis. In: Bosnacki and Leue [8], pp. 42–59
4. Ben-Ari, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, Reading (2006)
5. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, Heidelberg (2008)
6. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric Spin. STTT 4(1), 92–106 (2002)
7. Bosnacki, D., Dams, D., Holenderski, L., Sidorova, N.: Model checking SDL with Spin. In: Graf, S., Schwartzbach, M.I. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 363–377. Springer, Heidelberg (2000)
8. Bošnački, D., Leue, S. (eds.): SPIN 2002. LNCS, vol. 2318. Springer, Heidelberg (2002)
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (1992)

10. Chen, J., Cui, H.: Translation from adapted uml to promela for corba-based applications. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 234–251. Springer, Heidelberg (2004)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 481–496. Springer, Heidelberg (2005)
13. Donaldson, A.F., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 541–556. Springer, Heidelberg (2006)
14. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Partial-order reduction and trail improvement in directed model checking. STTT 6(4), 277–301 (2004)
15. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with hsf-spin. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
16. Formal Systems (Europe) Ltd. Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)
17. Godefroid, P. (ed.): SPIN 2005. LNCS, vol. 3639. Springer, Heidelberg (2005)
18. Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its promela translation. In: APSEC, pp. 283–290. IEEE Computer Society, Los Alamitos (2005)
19. Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
20. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods in System Design 13(3), 289–307 (1998)
21. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
22. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) FORTE. IFIP Conference Proceedings, vol. 6, pp. 197–211. Chapman & Hall, Boca Raton (1994)
23. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11, 256–290 (2002)
24. 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)
25. 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)
26. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 79–93. Springer, Heidelberg (2006)
27. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203 (2008)
28. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proceedings International Symmetry Conference, pp. 71–85 (January 2007)
29. McMillan, K.L.: Symbolic Model Checking. PhD thesis, Boston (1993)
30. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)

31. Prigent, A., Cassez, F., Dhaussy, P., Roux, O.: Extending the translation from sdl to promela. In: Bosnacki and Leue [8], pp. 79–94
32. Rothmaier, G., Kneiphoff, T., Krumm, H.: Using Spin and Eclipse for optimized high-level modeling and analysis of computer network attack models. In: Godefroid [17], pp. 236–250
33. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: Proceedings Symposium TASE 2008, Nanjing, China, June 2008, pp. 15–22. IEEE, Los Alamitos (2008)
34. Turner, E., Leuschel, M., Spermann, C., Butler, M.: Symmetry reduced model checking for B. In: Proceedings Symposium TASE 2007, Shanghai, China, June 2007, pp. 25–34. IEEE, Los Alamitos (2007)
35. Wachter, B.D., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with $_{d}$sl and Spin. Formal Asp. Comput. 17(2), 177–200 (2005)
36. Wiegard, H.: A comparison of the model checker ProB with Spin. Master's thesis, Institut für Informatik, Universität Düsseldorf (to appear, 2008)
37. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: DAC, pp. 599–604 (1998)

A Scheduler.Prom

The following is a manual translation (and slight simplification) of the scheduler1.ref machine into Promela (with 2 processes).

```
chan readyq = [2] of { byte } ; bool activef=0;
proctype user () {
    bool created=0; bool idle=0; bool ready=0; bool act=0;
    label1:
    do
      :: atomic{(created==0) -> created = 1; idle = 1}
      :: atomic{(created==1 && idle==1) -> created = 0; idle=0}
      :: atomic{idle==1 ->  idle=0; ready=1; label2:readyq!_pid }
      :: atomic{(readyq?[eval(_pid)] &&  ready==1 && activef==0) ->
                    readyq?eval(_pid);ready = 0 ->
                    activef=1 ->  act = 1 }
      :: atomic{act==1 -> idle = 1; act = 0; activef = 0}
    od;
 }
 /* initialize flags and start the processes */
init {  atomic{  run user();  run user(); }; printf("init\n")}
```

B Scheduler1.Ref

```
REFINEMENT scheduler1_improved
REFINES scheduler0
VARIABLES proc, readyq, activep, activef, idleset
INVARIANT proc : POW(PROC) &  /* created */
          readyq : seq(PROC) & activep : POW(PROC) &
          activef : BOOL &  idleset : POW(PROC)
```

```
INITIALISATION
    proc:={}  ||  readyq:={} ||
    activep:={} || activef := FALSE || idleset := {}
OPERATIONS
 new(p) = PRE p : PROC - proc THEN
        idleset := idleset \/ {p} || proc := proc \/ {p} END;
 del(p) = PRE p : PROC & p : idleset THEN
        proc := proc-{p} || idleset := idleset - {p} END;
 ready(p) = PRE  p : idleset THEN
        readyq:=readyq<-p || idleset := idleset - {p} END;
 enter(p) =  PRE p : PROC & readyq/=<> &
                 p = first(readyq) & activef=FALSE THEN
        activep:={p} || readyq := tail(readyq) ||
        activef:=TRUE END;
 leave(p) = PRE p : PROC &  activef=TRUE & p : activep THEN
        idleset := idleset \/ {p} || activef := FALSE ||
        activep := {} END
END
```

Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach

Egon Börger[1] and Bernhard Thalheim[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] Chair for Information Systems Engineering, Department of Computer Science, University of Kiel D-24098 Kiel
thalheim@is.informatik.uni-kiel.de

Abstract. We survey the use of the Abstract State Machines (ASM) method for a rigorous foundation of modeling and validating web services, workflows, interaction patterns and business processes. We show in particular that one can tailor business process definitions in application-domain yet rigorous terms in such a way that the resulting ASM models can be used as basis for binding contracts between domain experts and IT technologists. The method combines the expressive power and accuracy of rule-based modeling with the intuition provided by visual graph-based descriptions. We illustrate this by an ASM-based semantical framework for the OMG standard for BPMN (Business Process Modeling Notation). The framework supports *true concurrency*, *heterogeneous state* and *modularity* (compositional design and verification techniques). As validation example we report some experiments, carried out with a special-purpose ASM simulator, to evaluate various definitions proposed in the literature for the critical OR-join construct of BPMN.[1]

1 Introduction

Over the last five years the Abstract State Machines (ASM) method has been used successfully in various projects concerning modeling techniques for web services, workflow patterns, interaction patterns and business processes.

An execution semantics for (an early version of) the Business Process Execution Language for Web Services (BPEL) has been provided in terms of ASMs in [19,23] and has been reused in [17,18]. In [4] one finds a survey of recent applications of the ASM method to design, analyze and validate execution models for *service behavior mediation* [3], *service discovery* [2,20] and *service composition* techniques [22], three typical themes concerning Service Oriented Architectures

[1] The work of the first author is supported by a Research Award from the Alexander von Humboldt Foundation (*Humboldt Forschungspreis*), hosted by the Chair for Information Systems Engineering of the second author at the Computer Science Department of the University of Kiel/Germany.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 24–38, 2008.

(SOAs). In [22] multi-party communication is viewed as an orchestration problem ("finding a mediator to steer the interactions"). A systematic analysis, in terms of ASMs, of complex communication structures built from basic service interaction patterns has been carried out in [5]. The workflow patterns collected in [27,24], which are widely considered in the literature as paradigms for business process control structures, have been shown in [10] to be instances of eight (four sequential and four parallel) basic ASM workflow schemes.

Recently, we have adopted the ASM method for a systematic study of business process modeling techniques [14,13,12]. As authoritative reference for basic concepts and definitions we have chosen the OMG standard for BPMN [15], which has been defined to reduce the fragmentation of business process modeling notations and tools. In the following we describe the salient methodological features of this work and report our experience in applying the ASM framework [11] to provide a transparent accurate high-level definition of the execution semantics of the current BPMN standard (version 1.0 of 2006).

The paper is organized as follows. In Sect. 2 we explain how ASM models for business processes can serve as *ground model* and thus as basis for a precise software contract, allowing one to address the *correctness question* for a business process description with respect to the part of the real-world it is supposed to capture. In Sect. 3 we list the main methodological principles which guided us in defining a succinct modularized ASM that constitutes an abstract interpreter for the entire BPMN standard. In Sect. 4 we formulate the ASM rule pattern that underlies our feature-based description of specific workflow behaviors. In Sect. 5 we show how this scheme can be instantiated, choosing as example BPMN gateways. In Sect. 6 we illustrate by a discussion of the critical BPMN OR-join construct how one can put to use an appropriate combination of local and global state components in ASMs. We report here some results of an experimental validation, performed with a special-purpose ASM interpreter [25] that integrates with current graphical visualization tools, of different definitions proposed for the OR-Join in the literature. In Sect. 7 we point to some directly related work and research problems.

2 Building ASM Ground Models for Business Processes

To guarantee that software does what the customer expects it to do involves first of all to accurately describe those expectations and then to transform their description in a controlled way to machine code. This is a general problem for any kind of software, but it is particularly pressing in the case of software-driven business process management, given the large conceptual and methodological gap between the business domain, where the informal requirements originate, and the software domain, where code for execution by machines is produced. The two methodologically different tasks involved in solving the problem have been identified in [9] as *construction of ground models*, to fully capture the informal requirements in an experimentally validatable form, and their mathematically verifiable *stepwise detailing* (technically called refinement) to compilable code.

Ground models represent accurate "blueprints" of the piece of "real world" (here a business process) one wants to implement, a system reference documentation that binds all parties involved for the entire development and maintenance process. The need to check the accuracy of a ground model, which is about a not formalizable relation between a document and some part of the world, implies that the model is described in application domain terms one can reliably relate to the intuitive understanding by domain experts of the involved world phenomena. Ground models are vital to reach a firm understanding that is shared by the parties involved, so that a ground model has to serve as a solid basis for the communication between the (in our case three) parties: business analysts and operators, who work on the business process design and management side, information technology specialists, who are responsible for a faithful implementation of the designed processes, and users (suppliers and customers). We refer for a detailed dicussion of such issues to [9] and limit ourselves here to illustrate the idea by three examples of a direct (i.e. coding-free, abstract) mathematical representation of business process concepts in ASM ground models. The examples define some basic elements we adopted for the abstract BPMN interpreter in [13].

Example 1. Most business process model notations are based on flowcharting techniques, where business processes are represented by *diagrams* at whose nodes activities are executed and whose arcs are used to contain the information on the desired execution order (so-called control information). We therefore base our BPMN model on an underlying graph structure, at whose nodes ASM rules are executed which express the associated activity and the intended control flow.

Furthermore, usually the *control* flow is formulated using the so-called *token* concept, a program counter generalization known from the theory of Petri nets. The idea is that for an activity at a target node of incoming arcs to become executable, some (maybe all) arcs must be *Enabled* by a certain number of tokens being available at the arcs; when executing the activity, these tokens are CONSUMEd and possibly new tokens are PRODUCEd on the outgoing arcs. This can be directly expressed using an abstract dynamic function *token* associating (multiple occurrences of) tokens—elements of an abstract set *Token*—to arcs[2]:

$$token : Arc \to Multiset(Token)$$

The use of an abstract predicate *Enabled* and abstract token handling machines CONSUME and PRODUCE allows us to adapt the token model to different instantiations by a concrete token model. For example, a frequent understanding of *Enabled* is that of an atomic quantity formula, stating that the number of tokens currently associated to an arc *in*coming into a given node is at least a quantity $inQty(in)$ required at this arc.

$$Enabled(in) = (|\ token(in)\ | \geq inQty(in))$$

[2] In programming language terms one can understand $f(a_1, \ldots, a_n)$ for a dynamic function f as array variable.

With such a definition one can also specify further the abstract control related operations, namely to CONSUME ($inQty(in)$ many occurrences of) a token t on in and to PRODUCE ($outQty(out)$ many occurrences of) t on an arc outgoing from the given node.

$$\text{CONSUME}(t, in) = \text{DELETE}(t, inQty(in), token(in))$$
$$\text{PRODUCE}(t, out) = \text{INSERT}(t, outQty(out), token(out))$$

We express the *data* and *events*, which are relevant for an execution of the activity associated to a node and belong to the underlying database engine respectively to the environment, by appropriate ASM *locations*: so-called controlled (read-and-write) locations for the data and monitored (only read) locations for the events. Any kind of whatever complex value an application needs for data or events is allowed to be stored in an ASM location, directly, corresponding to the given level of abstraction, avoiding any encoding a later refinement to implementable data structures may require.

This approach allows us to combine the visual appeal of graph-based notations with the expressive power and simplicity of abstract-state and rule-based modeling: we can paraphrase the informal explanations in the BPMN standard document of "how the graphical elements will interact with each other, including conditional interactions based on attributes that create behavioral variations of the elements" [15, p.2] by corresponding ASM rules, which address issues the graphical notation does not clarify. More generally speaking, (asynchronous) ASMs can be operationally understood as extension of (locally synchronous and globally asynchronous [21]) Finite State Machines to FSMs working over abstract data. Therefore a domain expert, when using graphical design tools for FSM-like notations, can reason about the graphical elements in terms of ASMs whenever there is some need for an exact reference model to discuss semantically relevant issues.

Example 2. In business process design it is usual to distinguish between a *business process* (the static diagram) and its *instances* (with specific token marking and underlying data values). This distinction is directly reflected in the ASM model in terms of instantiations of the underlying parameters, which appear abstractly but explicitly in the ASM model. For example a token is usually characterized by the process ID of the process instance pi to which it belongs (via its creation at the start of the process instance), which allows one to distinguish tokens belonging to different instances of one process p. It suffices to write $token_{pi}$ to represent the current token marking in the process diagram instance of the process instance pi a token belongs to. In this way $token_{pi}(arc)$ denotes the *token* view of process instance pi at arc, namely the multiset of tokens currently residing on arc and belonging to process instance pi. BPEL uses this for a separation of each process instance by a separate XML document.

Correspondingly one has to further detail the above predicate *Enabled* by the stipulation that only tokens belonging to one same process instance have to be considered:

$$Enabled(in) = (|\ token_{pi}(in)\ | \geq inQty(in)\ \textbf{forsome}\ pi)$$

The reader will notice that the use of abstract INSERT and DELETE operations in defining the macros PRODUCE and CONSUME for tokens, instead of directly updating $token(a, t)$, comes handy: it makes the macros usable in a concurrent context, where multiple agents, belonging to multiple process instances, may want to simultaneously operate on the *token*s on an arc. Note that it is also consistent with the special case that in a transition with both DELETE(in, t) and INSERT(out, t) one may have $in = out$, so that the two operations are not considered as inconsistent, but their cumulative effect is considered.

Thus the ASM model of the given business process represents the scheme of the process, statically defined by its rules; its instances are the scheme instantiations obtained by substituting the parameters by concrete values belonging to the given process instance pi. In accordance with common practice, one can and usually does suppress notationally the process instance parameter pi, as we did when explaining the function *token* above, as long as it is clear from the context or does not play a particular role.

Example 3. This example is about the need for global data or control structures in various business process constructs, e.g. synchronization elements owned by cooperating process agents or more generally speaking data shared by local processes. They can be directly expressed in terms of global locations of possibly asynchronous ASMs, thus avoiding detours one has to invent in frameworks where transitions can express only local effects. For an illustration we refer again to the definition of the BPMN standard in [15]. It uses a predominantly local view for task execution and step control, although some constructs such as splits and joins, multi-instance processes, gotos (called links) , and sub-processes are bound by context integrity constraints.

For example splits can be intimately related by such integrity constraints to joins and thus their execution is not free of side (read: not local) effects. For an illustration see the discussion of the OR-join gateway construct of BPMN in Sect. 6.

Another example are data dependencies among different processes, whose description in [15] seems to be relegated to using associations, but really need global or shared locations to appropriately represent their role for the control flow.

3 Separation of Different Concerns

For design and analysis of business processes it turned out to be crucial that the ASM method supports to first explicitly separate and then smoothly combine the realization of different concerns, based upon appropriate abstractions supporting this form of modularization. We list some of the main separation principles, which we have used with advantage for the definition of the execution semantics for BPMN by ASMs.

Separation Principle 1. This principle is about the *separation of behavior from scheduling*. To cope with the distributed character of cooperating business processes, one needs descriptions that are compatible with various strategies to realize the described processes on different platforms for parallel and distributed

computing. This requires the underlying model of computation to support most general scheduling schemes, including *true concurrency*.

In general, in a given state of execution of a business process, more than one rule could be executable, even at one node. We call a node *Enabled* in a state (not to be confused with the omonymous *Enabled*ness predicate for arcs) if at least one of its associated rules is *Fireable* at this node in this state.[3]

We separate the description of workflow behavior from the description of the underlying scheduling strategy in the following way. We define specific business process transition rules, belonging to a set say *WorkflowTransition* of such rules, to describe the behavioral meaning of any workflow construct associated to a node. Separately, we define an abstract scheduling mechanism, to choose at each moment an enabled node and at the chosen node a fireable transition, by two not furthermore specified selection functions, say $select_{Node}$ and $select_{WorkflowTransition}$ defined over the sets *Node* of nodes respectively *WorkflowTransition*. These functions determine how to choose an enabled node and a fireable workflow transition at such a node for its execution. We then can combine behavior and scheduling by a rule scheme WorkflowTransitionInterpreter, which expresses how scheduling (together with the underlying control flow) determines when a particular node and rule (or an agent responsible for applying the rule) will be chosen for an execution step.

WorkflowTransitionInterpreter =
let $node = select_{Node}(\{n \mid n \in Node$ **and** $Enabled(n)\})$
let $rule = select_{WorkflowTransition}(\{r \mid r \in WorkflowTransition$ **and** $Fireable(r, node)\})$
rule

Separation Principle 2. The second principle is about the *separation of orthogonal constructs*. To make ASM workflow interpreters easily extensible and to pave the way for modular and possibly changing workflow specifications, we adopted a *feature-based* approach, where the meaning of workflow concepts is defined elementwise, construct by construct. For each control flow construct associated to a *node* we provide a dedicated rule (or set of rules) WorkflowTransition(*node*), belonging to the set *WorkflowTransition* in the WorkflowTransitionInterpreter scheme of the previous example, which abstractly describe the operational interpretation of the construct. We illustrate this in Sect. 5 by the ASM rules defining the execution behavior of BPMN gateways, which can be separated from the behavioral description of BPMN event and activity nodes.

Another example taken from BPMN is the separation of atomic tasks from non-atomic subprocesses and from activities with an iterative structure. BPMN distinguishes seven kinds of tasks:

$$TaskType = \{Service, User, Receive, Send, Script, Manual, Reference, None\}$$

[3] We treat the fireability of a rule (by an agent) as an abstract concept, because its exact interpretation may vary in different applications. For business process diagrams it clearly depends on the *Enabled*ness of the incoming arcs related to a rule at the given node, but typically also on further to be specified aspects, like certain events to happen, on the (degree of) availability of needed resources, etc.

These task types are based on whether a message has been sent or received or whether the task is executed or calls another process. The execution semantics for task nodes is given by one ASM rule (scheme) [13], which uses as interface abstract machines to SEND or RECEIVE messages and to CALL or EXECute processes.

A third example from the BPMN standard is the separation of cyclic from acyclic processes, whe we use for the discussion of the OR-join gateway in Sect. 6.

Separation Principle 3. The third principle is the *separation of different model dimensions* like control, events, data and resources. Such a separation is typical for business process notations, but the focus of most of these notations on control (read: execution order, possibly influenced also by events) results often in leaving the underlying data or resource features either completely undefined or only partly and incompletely specified. The notion of abstract state coming with ASMs supports to not simply neglect data or resources when speaking about control, but to tailor their specification to the needed degree of detail, hiding what is considered as irrelevant at the intended level of abstraction but showing explicitly what is needed. We illustrate this in Sect. 4 by the four components for data, control, events and resources in WORKFLOWTRANSITION, which constitute four model dimensions that come together in the ASM scheme for workflow interpreter rules. These four components are extensively used in the BPMN standard, although the focus is on the control flow, which is represented by the control flow arcs, relegating interprocess communication (via message flow arcs between processes) and data conditions and operations to minor concerns. For an enhanced specification of interprocess communication see the orchestration of processes in [28].

Separation Principle 4. The fourth principle, whose adoption helps to reduce the description size of abstract models, is the *separation of rule schemes and concrete rules*, where the concrete rules may also be specialized rule schemes. It exploits the powerful abstraction mechanisms ASMs offer for both data and operations, whether static or dynamic. We illustrate this by the COMPLEXGATETRANSITION in Sect. 5.1, a scheme from which one can easily define the behavior of the other BPMN gateways by instantiating some of the abstractions (see Sect. 5.2).

Separation Principle 5. The fifth example is about the *separation of design, experimental validation and mathematical verification* of models and their properties. In Sect. 6 we illustrate an application of this principle by an analysis of the OR-join gateway, where for a good understanding of the problem one better separates the definition of the construct from the computation or verification of its synchronization behavior. Once a ground model is defined, one can verify properties for diagrams, separating the cases with or without cyclces and in the former case showing which cycles in a diagram are alive and which ones may result in a deadlock. In addition, the specialized ASM workflow simulator [25] allows one to trace and to experimentally validate the behaviour of cyclic diagrams.

The principle goes together with the separation of different levels of detail at which the verification of properties of interest can take place, ranging from proof sketches over traditional or formalized mathematical proofs to tool supported

proof checking or interactive or automated theorem proving, all of which can and have been used for ASM models (see [11, Ch.8,9] for details).

Separation Principle 6. This pinciple is about the *separation of responsibilities, rights and roles of users* of BPMN diagrams. To represent different roles of users BPMN diagrams can be split into so-called pools, between which messages can be exchanged. Furthermore user actions can be separated by so-called swimlanes. Such a separation of user actions depending on the user's role within a diagram is supported in a natural way by the ASM concept of rule executing agents: one can associate different and even independent agents to sets of user rules; moreover these agents could be supervised by a user superagent coming with his own supervising rules, which leads to more general interaction patterns than what is foreseen by the BPMN standard (see [5]).

In the next section we show how from a combination of the separation principles formulated above one can derive an orthogonal high-level interpretation of the basic concepts of BPMN.

4 The Scheme for Workflow Interpreter Rules

For every workflow or BPMN construct associated to a *node*, its behavioral meaning can be expressed by a guarded transition rule WorkflowTransition(*node*) ∈ *WorkflowTransition* of the general form defined below. Every such rule states upon which events and under which further conditions—typically on the control flow, the underlying data and the availability of resources—the rule can fire to execute the following actions:

- perform specific operations on the underlying data ('how to change the internal state') and control ('where to proceed'),
- possibly trigger new events (besides consuming the triggering ones),
- operate on the resource space to handle (take possession of or release) resources.

In the scheme, the events and conditions in question remain abstract, the same as the operations that are performed. This allows one to instantiate them by further detailing the guards (expressions) respectively the submachines for the description of concrete workflow transitions.[4]

```
WorkflowTransition(node) =
  if EventCond(node) and CtlCond(node)
    and DataCond(node) and ResourceCond(node) then
      DataOp(node)
      CtlOp(node)
      EventOp(node)
      ResourceOp(node)
```

[4] We remind the reader that by the synchronous parallelism of single-agent ASMs, in each step all applicable rules are executed simultaneously, starting from the same state to produce together the next state.

WorkflowTransition(*node*) represents an abstract state machine, in fact a scheme (sometimes also called a pattern) for a set of concrete machines that can be obtained by further specifying the guards and the submachines for each given *node*. In the next section we illustrate such an instantiation process to define the behavior of BPMN gateways by ASM rules taken from the high-level BPMN interpreter defined in [13].

5 Instantiating WorkflowTransition for BPMN Gateways

In this section we instantiate WorkflowTransition for BPMN gateways, nodes standing for one of the three types of BPMN flow objects. The other two types are event and activity nodes, whose behavior can be described by similar instantiations, see [13] for the details. We start with the rule for so-called complex gateway nodes, from which the behavior of the other BPMN gateway constructs can be defined as special cases.

5.1 ComplexGateTransition

Gateways are used to describe the convergence (also called merging) and/or divergence (also called splitting) of control flow, in the sense that tokens can 'be merged together on input and/or split apart on output' [15, p.68]. For both control flow operations one has to determine the set of incoming respectively outgoing arcs they are applied to at the given node. The particular choices depend on the *node*, so that we represent them by two abstract selection functions, namely to

- $select_{Consume}$ the incoming arcs where tokens are consumed,
- $select_{Produce}$ the outgoing arcs where tokens are produced.

Both selection functions come with constraints: $select_{Consume}$ is required to select upon each invocation a non-empty set of enabled incoming arcs, whose *firingToken*s are to be consumed in one transition.[5] $select_{Produce}$ is constrained to select upon each invocation a non-empty subset of outgoing arcs o satisfying an associated $OutCond(o)$. On these arcs *complxGateToken*s are produced, whose particular form may depend on the *firingToken*s. We skip that in addition, as (part of) DataOp(*node*), multiple assignments may be 'performed when the Gate is selected' [15, Table 9.30 p.86] (read: when the associated rule is fired).

[5] A function $firingToken(A)$ is used to express a structural relation between the consumed incoming and the produced outgoing tokens, as described in [15, p.35]. It is assumed to select for each element a of an ordered set A of incoming arcs some of its $token(a)$ to be Consumed. For the sake of exposition we make the usual assumption that $inQty(in) = 1$, so that we can use the following sequence notation: $firingToken([a_1, \ldots, a_n]) = [t_1, \ldots, t_n]$ denotes that t_i is the token selected to be fired on arc a_i.

ComplexGateTransition($node$) =
  let
    $I = select_{Consume}(node)$
    $O = select_{Produce}(node)$
  in WorkflowTransition($node, I, O$)
where
  $CtlCond(node, I) = (I \neq \emptyset$ **and forall** $in \in I$ $Enabled(in))$
  CtlOp($node, I, O$) =
    if $O \neq \emptyset$ **and** $O \subseteq \{o \in outArc(node) \mid OutCond(o)\}$ **then**
      ProduceAll($\{(complxGateToken(firingToken(I), o), o) \mid o \in O\}$)
      ConsumeAll($\{(t_i, in_i) \mid 1 \leq i \leq n\}$) **where**
        $[t_1, \ldots, t_n] = firingToken(I), [in_1, \ldots, in_n] = I$

5.2 Instantiating ComplexGateTransition

The BPMN standard defines and names also special gateways, which can all be obtained by specializing the selection functions in ComplexGateTransition. To describe these instantiations here more clearly, we assume without loss of generality that these special gateways never have both multiple incoming and multiple outgoing arcs. Thus the so-called split gateways have one incoming and multiple outgoing arcs, whereas the so-called join gateways have multiple incoming and one outgoing arc.

For AND-split and AND-join gateway nodes, $select_{Produce}$ and $select_{Consume}$ are required to yield all outgoing resp. all incoming arcs.

For OR-split nodes two cases are distinguished: $select_{Produce}$ chooses exactly one (exclusive case, called XOR-split) or at least one outgoing arc (called inclusive OR or simply OR-split). For the exclusive case a further distinction is made depending on whether the decision is 'data-based' or 'event-based', meaning that $OutCond(o)$ is a $DataCond(o)$ or an $EventCond(o)$. For both cases it is required to select the first $out \in outArc(node)$, in the given order of gates, satisfying $GateCond(out)$.

Similarly also for OR-join nodes two versions are distinguished, an exclusive and data-based one—the event-based XOR is forbidden by the standard to act only as a Merge—and an event-based inclusive one. In the latter case $select_{Consume}$ is required to yield a subset of the incoming arcs with associated tokens 'that have been produced upstream' [15, p.80], but no indication is given how to determine this subset, which is a synchronization problem. We discuss this very much disputed issue further in the next section.

6 OR-Join Gateway: Global versus Local Description Elements

The OR-join concept is present in many workflow and business process modeling languages and is used with different understandings advocated in the literature, in different commercial workflow systems and by different users. Part of this

situation stems from the fact that in dealing with the OR-join concept, often two things are mixed up that should be kept separate, namely a) how the intended meaning of the concept is defined (question of semantics) and b) how properties of interest for the construct (most importantly its fireability in a given state) can be computed, validated (at run time) or verified (at design time) (question of computation, validation and verification methods).

It could be objected that an algorithm to compute the fireability of the OR-join rules defines the crucial synchronization property and thus the semantics of the OR-join. Speaking in general terms this is true, but then the question is whether there is agreement on which algorithm to use and whether the algorithm is understandable enough to serve as a behavioral specification the business process expert can work with. However, looking at the literature there seems to be no agreement on which algorithm should be used and the complexity of the proposed ones makes them unfit to serve as satisfactory semantical specification for the workflow practitioner.

The semantical issue disputed in the literature is the specification of the $select_{Consume}$ functions, which incorporate the critical synchronization conditions. $select_{Consume}(node)$ plays the role of an interface for triggering for a set of to-be-synchronized incoming arcs the execution of the rule at the given *node*. Unfortunately, most proposals for an OR-join semantics in one way or the other depend on the framework used for the definition. This is particularly evident in the case of Petri-net-based definitions, where, to circumvent the restrictions imposed by the local nature of what a Petri net transition can express, either the diagrams are restricted (to the possible dislike of a business process practitioner) or ad hoc extensions of Petri nets are introduced that are hard to motivate in application domain terms (see for example [33,31,32]). A side problem is that the BPMN standard document seems to foresee that the function is dynamic (run-time determined), since the following is required:

> Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process ... Some of the incoming Sequence Flow will not have signals and the pattern of which Sequence Flow will have signals may change for different instantiations of the Process. [15, p.80]

We refer to [12] for a detailed discussion of OR-join variations and ways to define and compute the underlying synchronization functions $select_{Consume}$. We restrict our attention here to report some experiments Ove Soerensen has made with various alternatives we considered to come up with a practically acceptable definition that could serve for the standard, in particular in connection with diagrams that may contain cycles. For this purpose Soerensen has built a specialized ASM workflow simulator [25] that is interfaced with standard graph representation tools, so that the token flow and the unfolding of a diagram cycle triggered by applying ASM OR-join rules can be visualized.

One alternative we considered is to a) pass at runtime every potential synchronization request from where it is originated (a split gateway node) to each

downstream arc that enters a join gateway node and to b) delete this request each time the synchronization possibility disappears due to branching. Assume for the moment that the given diagram contains no cycles and assume without loss of generality that there is a unique start node. Then it suffices to operate the following refinement on our BMPN model.

- **Split gate transition refinement.** When due to an incoming token t at a split *node* a new token $t.o$ is produced on an arc o outgoing *node*, a computation path starts at o that may need to be synchronized with other computation paths started simultaneously at this *node*, so that also a synchronization copy is produced and placed on each downstream arc that enters a join node, i.e. an arc entering a join node to which a path leads from o. We denote the set of these join *arc*s by $AllJoinArc(o)$. Simultaneously the synchronization copy of t is deleted from all such arcs that are reachable from *node*.
- **Join gate transition refinement.** We consume the synchronization tokens that, once the to-be-synchronized tokens have been fired, have served their purpose, and produce new synchronization tokens for the tokens the join produces. To $CtlCond(node, I)$ we add the synchronization condition that I is a synchronization family at *node*, which means a set of incoming arcs with non-empty *syncToken* sets such that all other incoming arcs (i.e. those not in I) have empty *syncToken* set (read: are arcs where no token is still announced for synchronization so that no token will arrive any more (from upstream) to enable such an arc).

It is not difficult to formulate this idea as a precise refinement (in the sense of [8]) of our ASMs for BPMN split and join rules (see [12]). To extend this approach to the case of diagrams with cycles (more generally subprocesses), one can refine the *AllJoinArc* function to yield only arcs of join nodes up to and including the next subprocess entry node; inside a subprocess *AllJoinArc* is further restricted to only yield join nodes that are inside the subprocess.[6] The production of synchronization tokens by the transition rule for join gate nodes that enter a subprocess is postponed to the exit node rule(s) of the subprocess.

There are obviously various other possibilities, with all of which one can experiment using the work that will be reported in [25].

7 Related and Future Work

There are two specific papers we know on the definition of a formal semantics of a subset of BPMN. In [16] a Petri net model is developed for a core subset of BPMN which however; it is stated there that due to the well-known lack of high-level concepts in Petri nets, this Petri net model "does not fully deal with: (i) parallel multi-instance activities; (ii) exception handling in the context of subprocesses that are executed multiple times concurrently; and (iii) OR-join gateways. "

[6] In Soerensen's tool this is realized by spanning a new diagram copy of the subprocess.

In [30] it is shown "how a subset of the BPMN can be given a process semantics in Communicating Sequential Processes", starting with a formalization of the BPMN syntax using the Z notation and offering the possibility to use the CSP-based model checker for an analysis of model-checkable properties of business processes written in the formalized subset of BPMN. The execution semantics for BPMN defined in [13] covers every standard construct and is defined in the form of **if** *Event* **and** *Condition* **then** *Action* rules of Event-Condition-Action systems, which are familiar to most analysts and professionals trained in process-oriented thinking. Since ASMs assign a precise mathematical meaning to abstract (pseudo) code, for the verification and validation of properties of ASMs one can adopt every appropriate accurate method, without being restricted to, but allowing one to use, appropriate mechanical (theorem proving or model checking) techniques.

In [29] an inclusion of process interaction and resource usage concerns is advocated for the forthcoming extension BPMN 2.0 of BPMN. It could be worth to investigate how the ASM models defined in [5] for the interaction patterns in [1] can be included into the current ASM model for BPMN, extending the current communication means in BPMN—event handling, message exchange between pools and data exchange between processes—to richer forms of interaction between multiple processes. Also a rigorous analysis of scheduling and concurrency mechanisms would be interesting, in particular in connection with concerns about resources and workload balancing that play a crucial role for efficient implementations.

The feature-based definition of workflow concepts in this paper is an adaptation of the method used in a similar fashion in [26] for an instructionwise definition, verification and validation of interpreters for Java and the JVM. This method has been developed independently for the definition and validation of software product lines [7], see [6] for the relation between the two methods.

References

1. Barros, A., Dumas, M., Hofstede, A.: Service interaction patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 302–318. Springer, Heidelberg (2005)
2. Altenhofen, M., Börger, E., Friesen, A., Lemcke, J.: A high-level specification for virtual providers. International Journal of Business Process Integration and Management 1(4), 267–278 (2006)
3. Altenhofen, M., Börger, E., Lemcke, J.: A high-level specification for mediators (virtual providers). In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 116–129. Springer, Heidelberg (2006)
4. Altenhofen, M., Friesen, A., Lemcke, J.: ASMs in service oriented architectures. Journal of Universal Computer Science (2008)
5. Barros, A., Börger, E.: A compositional framework for service interaction patterns and communication flows. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 5–35. Springer, Heidelberg (2005)

6. Batory, D., Börger, E.: Modularizing theorems for software product lines: The Jbook case study. In: Hartmann, S., Kern-Isberner, G. (eds.) FoIKS 2008. LNCS, vol. 4932, pp. 1–4. Springer, Heidelberg (2008)
7. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. In: ACM TOSEM. ASM (October 1992)
8. Börger, E.: The ASM refinement method. Formal Aspects of Computing 15, 237–257 (2003)
9. Börger, E.: Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. Formal Aspects of Computing 19, 225–241 (2007)
10. Börger, E.: Modeling workflow patterns from first principles. In: Storey, V.C., Parent, C., Schewe, K.-D., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 1–20. Springer, Heidelberg (2007)
11. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
12. Börger, E., Thalheim, B.: Experiments with the behavior of or-joins in business process models. J. Universal Computer Science (submitted, 2008)
13. Börger, E., Thalheim, B.: A high-level BPMN interpreter (submitted)
14. Börger, E., Thalheim, B.: A method for verifiable and validatable business process modeling. In: Advances in Software Engineering. LNCS, Springer, Heidelberg (2008)
15. BPMI.org. Business Process Modeling Notation Specification. dtc/2006-02-01 (2006), http://www.omg.org/technology/documents/spec_catalog.htm
16. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri nets. Technical Report 7115, Queensland University of Technology, Brisbane (2007)
17. Fahland, D.: Ein Ansatz einer Formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Master's thesis, Humboldt-Universität zu Berlin (June 2004)
18. Fahland, D., Reisig, W.: ASM semantics for BPEL: the negative control flow. In: Beauquier, D., Börger, E., Slissenko, A. (eds.) Proc. ASM 2005, Université de Paris, vol. 12, pp. 131–152 (2005)
19. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and validation of the Business Process Execution Language for web services. In: Zimmermann, W., Thalheim, B. (eds.) ASM 2004. LNCS, vol. 3052, pp. 78–94. Springer, Heidelberg (2004)
20. Friesen, A., Börger, E.: A high-level specification for semantic web service discovery services. In: ICWE 2006: Workshop Proceedings of the Sixth International Conference on Web Engineering (2006)
21. Lavagno, L., Sangiovanni-Vincentelli, A., Sentovitch, E.M.: Models of computation for system design. In: Börger, E. (ed.) Architecture Design and Validation Methods, pp. 243–295. Springer, Heidelberg (2000)
22. Lemcke, J., Friesen, A.: Composing web-service-like Abstract State Machines (ASMs). In: Workshop on Web Service Composition and Adaptation (WSCA 2007); IEEE International Conference on Web Service (ICWS 2007) (2007)
23. Farahbod, U.G.R., Vajihollahi, M.: An Abstract Machine Architecture for Web Service Based Business Process Management. Int. J. Business Process Integration and Management 1(4), 279–291 (2006)
24. Russel, N., ter Hofstede, A., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. BPM-06-22 (July 2006), http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/

25. Sörensen, O.: Diplomarbeit. Master's thesis, University of Kiel, forthcoming (2008), www.is.informatik.uni-kiel/~thalheim/ASM/MetaProgrammingASM
26. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001)
27. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)
28. Weske, M.: Business Process Management. Springer, Heidelberg (2007)
29. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A., Russel, N.: On the suitability of BPMN for business process modelling. In: The 4th Int. Conf. on Business Process Management (2006)
30. Wong, P.Y.H., Gibbons, J.: A process semantics fo BPMN. Oxford University Computing Lab (preprint, July 2007), http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmn_extended.pdf
31. Wynn, M., van der Aalst, W., ter Hofstede, A., Edmond, D.: Verifying workflows with cancellation regions and OR-joins: an approach based on reset nets and reachability analysis. In: Dustdar, S., Fiadeiro, J.L., Seth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 389–394. Springer, Heidelberg (2006); Previous versions edited as BPM-06-16 and BPM-06-12
32. Wynn, M., Verbeek, H.M.W., van der Aalst, W., ter Hofstede, A., Edmond, D.: Reduction rules for reset workflow nets. Technical Report BPM-06-25, BPMcenter.org (2006)
33. Wynn, M., Verbeek, H.M.W., van der Aalst, W., ter Hofstede, A., Edmond, D.: Reduction rules for YAWL workflow nets with cancellation regions and OR-joins. Technical Report BPM-06-24, BPMcenter.org (2006)

Refinement of State-Based Systems: ASMs and Big Commuting Diagrams (Abstract)

Gerhard Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de

Effective and efficient support for the incremental development of verified implementations from abstract requirements has always been of central importance for the successful application of formal methods in practice.

Effective means first, that a modelling language is available that allows an adequate problem specification. Second, a refinement theory must be available that preserves the relevant properties of the abstract specification.

Efficient means, that the refinement theory reduces the problem to the essential proof obligations necessary, and that the theorem prover provides powerful deduction support.

The talk discusses the topic from the experience we have gained from formalizing various refinement theories [1], [2] with the interactive theorem prover KIV [3], as well as from the correctness proofs for various case studies involving refinement.

Predominantly, we have uses Abstract State Machines (ASM, [4], [5]) and ASM Refinement [6], [7], [8], [9] in our case studies, but occasionally we also used data refinement [10], [11] and refinement preserving noninterference [12].

ASM refinement was originally developed for the verification of 12 refinements [13], [14], [15] of the Prolog compiler given in [16]. We will relate the definition of ASM refinement to various other refinement definition for state-based systems that have been defined: data refinement [17] and its various instances (the contract approach [18] used in Z, the behavioral approach [19] used in Object-Z or the variation used in Event-B [20]), refinement of IO automata [21] and refinement in the Abadi-Lamport [22] setting.

One important lesson learned from this case study was that the use of "'big commuting diagrams"' can often simplify proofs enormously. Such big diagrams relate several (m) abstract steps to several (n) concrete steps (m:n diagrams). They generalize forward (also called downward) simulations from data refinement which relates one abstract to one concrete step. Various generalization of data refinement (e.g. stuttering forward simulation from IO refinement [21], weak forward simulation [23] and coupled refinement [24]) can be shown to be special cases [8]. A recent result is that generalized forward simulations alone are a complete proof system [2] together with using choice functions as proposed in [25] and [5]. A similar result, that also considers fairness, has been derived earlier for the Abadi-Lamport setting by Wim Hesselink [26].

Big commuting diagrams are natural in compiler verification: some source code instructions are either implemented by some assembler instructions or optimized to a shorter sequence of instructions. We found that such diagrams can be used in other

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 39–41, 2008.

application areas too: one example is security protocol verification [27], [28], [29], where one abstract action (like data transfer) is implemented by running an n-step protocol (1:n diagram). An additional difficulty compared to compiler verification is that protocol runs are interleaved with other protocol runs and actions of an attacker. Our current research [10], [30] focusses on the verification of lock-free algorithms. These implements an atomic actions on a data type (such as a push or pop on a stack) by programs working on pointer structures. Like security protocols the programs are interleaved and we report on experiments using big commuting diagrams to simplify the verification task.

References

1. Boiten, E., Derrick, J., Schellhorn, G.: Relational concurrent refinement part ii: Internal operations and ouputs. In: FAC (2008)
2. Schellhorn, G.: Completeness of asm refinement. In: Proceedings of REFINE 2008. ENTCS (to appear, 2008)
3. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications. Systems and Implementation Techniques, vol. II, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
4. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford Univ. Press, Oxford (1995)
5. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
6. 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`
7. Börger, E.: The ASM Refinement Method. Formal Aspects of Computing 15(1–2), 237–257 (2003)
8. 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)
9. Schellhorn, G.: ASM Refinement Preserving Invariants. In: Proceedings of the 14th International ASM Workshop, ASM 2007, Grimstad, Norway (2008); JUCS (to appear), `http://ikt.hia.no/asm07/`
10. Derrick, J.: Mechanizing a refinement proof for a lock-free concurrent stack. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 78–95. Springer, Heidelberg (2008)
11. 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 20(1) (January 2008)
12. Schellhorn, G., Reif, W., Schairer, A., Karger, P., Austel, V., Toll, D.: Verification of a Formal Security Model for Multiapplicative Smart Cards. special issue of the Journal of Computer Security 10(4), 339–367 (2002)
13. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. Journal of Universal Computer Science (J.UCS) 3(4), 377–413 (1997), `http://www.jucs.org`
14. 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. Applications, vol. III, pp. 165–194. Kluwer Academic Publishers, Dordrecht (1998)

15. Schellhorn, G.: Verification of Abstract State Machines. PhD thesis, Universität Ulm, Fakultät für Informatik (1999), http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/
16. 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)
17. Jifeng, H., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
18. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science (1996)
19. Bolton, C., Davies, J., Woodcock, J.: On the refinement and simulation of data types and processes. In: Araki, K., Galloway, A., Taguchi, K. (eds.) Proceedings of the International conference of Integrated Formal Methods (IFM), pp. 273–292. Springer, Heidelberg (1999)
20. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae 21 (2006)
21. Lynch, N., Vaandrager, F.: Forward and Backward Simulations – Part I: Untimed systems. Information and Computation 121(2), 214–233 (1995); also: Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, MIT
22. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 2, 253–284 (1991); Also appeared as SRC Research Report 29
23. Derrick, J., Boiten, E.A., Bowman, H., Steen, M.: Weak Refinement in Z. In: Bowen, J., Hinchey, M. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 369–388. Springer, Heidelberg (1997)
24. Derrick, J., Wehrheim, H.: Using Coupled Simulations in Non-atomic Refinement. In: Bert, D., Bowen, J., King, S., Walden, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 127–147. Springer, Heidelberg (2003)
25. Stärk, R.F., Nanchen, S.: A Complete Logic for Abstract State Machines. Journal of Universal Computer Science (J.UCS) 7 (11), 981–1006 (2001)
26. Hesselink, W.H.: Universal extensions to simulate specifications. Information and Computation 206, 106–128 (2008)
27. Schellhorn, G., Grandy, H., Haneberg, D., Moebius, N., Reif, W.: A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In: Abrial, J.-R., Glässer, U. (eds.) Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, Springer, Heidelberg (2008)
28. Banach, R., Schellhorn, G.: On the refinement of atomic actions. In: Proceedings of REFINE 2007. ENTCS (2007)
29. Schellhorn, G., Banach, R.: A concept-driven construction of the mondex protocol using three refinements. In: Proceedings of ABZ conference 2008, vol. 5238. Springer, Heidelberg (2008)
30. Bäumler, S., Schellhorn, G., Balser, M., Reif, W.: Proving linearizability with temporal logic (submitted, draft available from the authors)

Model Based Refinement and the Tools of Tomorrow

Richard Banach

School of Computer Science,
University of Manchester,
Manchester, M13 9PL, UK
banach@cs.man.ac.uk

Abstract. The ingredients of typical model based development via refinement are re-examined, and some well known frameworks are reviewed in that light, drawing out commonalities and differences. It is observed that alterations in semantics take place *de facto* due to applications pressures and for other reasons. This leads to a perspective on tools for such methods in which the proof obligations become programmable and/or configurable, permitting easier co-operation between techniques and interaction with an Evidential Tool Bus. This is of intrinsic interest, and also relevant to the Verification Grand Challenge.

Keywords: Model Based Development, Refinement, Configurable Proof Obligations, Tools, Verification Grand Challenge.

1 Introduction

Refinement, as a model based methodology for developing systems from abstract specifications, has been around for a long time [1]. In this period, many variations on the basic idea have arisen, to the extent that an initiate can be bewildered by the apparently huge choice available. As well as mainstream refinement methodologies such as ASM, B, Z, etc., which have enjoyed significant applications use, there are a myriad other related theories in the literature, too numerous to cite comprehensively. And at a detailed theoretical level, they are all slightly different.

From a developer's point of view, this variety can only be detrimental to the wider adoption of formal techniques in the real world applications arena — in the real world, developers have a host of things to worry about, quite removed from evaluating the detailed technical differences between diverse formal techniques in order to make the best choice regarding which one to use. In any event, such choice is often made on quite pragmatic grounds, such as the ready access to one or more experts, and crucially these days, availability of appropriate tool support. Anecdotally, the choice of one or another formalism appears to make little difference to the outcome of a real world project using such techniques — success seems to be much more connected with proper requirements capture, and with organising the development task in a way that is sympathetic to both the formal technique and to the developers' pre-existing development practices.

In this paper we examine closely what goes into a typical notion of model based refinement by examining a number of cases. As a result, we can extract the detailed similarities and differences, and use this to inform a view on how different techniques ought to relate to one another. This in turn forms a perspective on how different techniques

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 42–56, 2008.

can meet a contemporary environment in which verification techniques and their tools can increasingly address mainstream industrial scale problems — determining how to address the spectrum of technical differences between techniques, in the face of a wider world prone to see them as intrinsically divisive, remains a significant challenge. In this paper, we contend that techniques in this field can be viewed as comprising a number of features, amongst which, the commonly occuring ones ought to be emphasised, and the more specific ones deserve to be viewed more flexibly. This line is developed to the point that a landscape can be imagined, in which different techniques, and their tools, can ultimately talk to one another.

The rest of the paper is as follows. In Section 2 we cover the common features of model based formalisms. In Section 3 we show how these generalities are reflected in a number of specific well known approaches. Section 4 reflects on the evidence accumulated in the previous one and draws some appropriate conclusions. Section 5 takes the preceding material and debates the implications for tools. It is suggested that increased programmability can substantially help to bridge the gaps between techniques, and the way that programmability features in some recent tools is discussed. These thoughts are also in sympathy with the SRI 'Evidential Tool Bus' idea [2], and can contribute positively towards the current Verification Grand Challenge [3,4,5]. Section 6 concludes.

2 Model Based Refinement Methods: Generalities

A typical model based formal refinement method, whose aim is to formalise how an abstract model may be refined to a more concrete one, consists of a number of elements which interact in ways which are sometimes subtle. In this section we bring some of these facets into the light; the discussion may be compared to a similar one in [6].

Formal language. All formal refinement techniques need to be quite specific about the language in which the elements of the technique are formalised. This precision is needed for proper theoretical reasoning, and to enable mechanical tools with well defined behaviour to be created for carrying out activities associated with the method. There are inevitably predicates of one kind or another to describe the properties of the abstract and concrete models, but technical means are also needed to express state change within the technique. Compared with the predicates used for properties, there is much more variety in the linguistic means used for expressing state change, although each has a firm connection with the predicates used for the modelling of properties.

Granularity and naming. All formal refinement techniques depend on relating concrete steps (or collections of steps) to the abstract steps (or collections of steps) which they refine. Very frequently, a single concrete step is made to correspond to a single abstract one, but occasionally more general schemes (in which sequences of abstract and concrete steps figure) are considered. The $(1, 1)$ scheme is certainly convenient to deal with theoretically, and it is often captured by demanding that the names of operations or steps that are intended to correspond at abstract and concrete levels are the same. However, in many applications contexts, such a simple naming scheme is far removed from reality, and if naively hardwired into the structure of a tool, makes the tool much less conveniently usable in practice.

Concrete-abstract fidelity. All formal refinement techniques demand that the concrete steps relate in a suitable manner to abstract ones. Almost universally, a retrieve relation (also referred to as a refinement mapping, abstraction relation, gluing relation, etc.) is used to express this relationship. It is demanded that the retrieve relation holds between the before-states of a concrete step (or sequence of steps) and the abstract step (or sequence of steps) which simulates it; likewise it must hold for the after-states of the simulating pair. In other words (sequences of) concrete steps must be faithful to (sequences of) abstract steps. (A special case, simple refinement, arises when the retrieve relation is an identity.)

Concrete-abstract fidelity is the one feature that can be found in essentially the same form across the whole family of model based formalisms. It is also the case that this fidelity —usually expressed using a proof obligation (PO), the *fidelity PO*— is often derived as a *sufficient condition* for a more abstract formulation of refinement, concerning the overall behaviour of 'whole programs'. These sufficient conditions normally form the focus of the theory of model based refinement techniques, since they offer what is usually the only route to proving refinement in practical cases.

Notions of correctness. One of the responsibilities of a formal refinement technique is to dictate *when* there should be concrete steps that correspond to the existence of abstract ones. This (at least implicitly) is connected with the potential for refinement techniques to be used in a black-box manner. Thus if an abstract model has been drawn up which deals adequately with the requirements of the problem, then any refinement should *guarantee* that the behaviour expressed in the abstract model should be reflected appropriately in more concrete models, and ultimately in the implementation, so that the requirements coverage persists through to code.

There is much variation among refinement techniques on how this is handled, particularly when we take matters of interpretation into account. Although the mainstream techniques we discuss below are reasonably consistent about the issue, some variation is to be found, and more variety can be found among refinement variants in the literature. The formal content of these perspectives gets captured in suitable POs, and often, the policy adopted has some impact on the fidelity PO too. A similar impact can be felt in initialisation (and finalisation) POs.

Interpretation. The preceding referred (rather obliquely perhaps) to elements of model based refinement theories that are expressed in the POs of the theory, i.e. *via logic*. However, this does not determine how the logical elements relate to phenomena in the real world. If transitions are to be described by logical formulae (involving before and after states, say), then those formulae can potentially take the value false as well as true. And while determining how the logical formulae correspond to the real world is usually fairly straightforward in the true case, determining the correspondence in the false case can be more subtle. These matters of logical-to-real-world correspondence constitute the *interpretation* aspects of a formal development technique.

Trace inclusion. Trace inclusion, i.e. the criterion that every execution sequence of the system (i.e. the concrete model) is as permitted by the specification (i.e. the abstract model), is of immense importance in the real world. When an implemented system

behaves unexpectedly, the principal *post hoc* method of investigation amounts to determining how the preceding behaviour failed to satisfy the trace inclusion criterion. This importance is further underlined by the role that trace inclusion plays in model checking. The 'whole program' starting point of the derivation of many sufficient conditions for refinement is also rooted in trace inclusion. Two forms of trace inclusion are of interest. *Weak trace inclusion* merely states that for every concrete trace there is a simulating abstract one. *Strong trace inclusion* goes beyond that and states that if *Asteps* simulates *Csteps* and we extend *Csteps* to $Csteps \fatsemi C_{nxt}$, then *Asteps* can be extended to $Asteps \fatsemi A_{nxt}$ which also simulates. With weak trace inclusion, we might have to abandon *Asteps* and find some unrelated $Asteps_{different}$ to recover simulation of $Csteps \fatsemi C_{nxt}$.

Composition. It is a given that large systems are built up out of smaller components, so the interaction of this aspect with the details of a refinement development methodology are of some interest, at least for practical applications. Even more so than for notions of correctness, there is considerable variation among refinement techniques on how compositionality is handled — the small number of techniques we review in more detail below already exhibit quite a diversity of approaches to the issue.

3 ASM, B, Event-B, Z

In this section, we briefly review how the various elements of model based methods outlined above are reflected in a number of specific and well-known formalisms. For reasons of space, we restrict to the ASM, B (together with the more recent Event-B) and Z methodologies. We also stick to a forward simulation perspective throughout. It turns out to be convenient to work in reverse alphabetical order.

3.1 Z

Since Z itself [7] is simply a formal mathematical language, one cannot speak definitively of *the* Z refinement. We target our remarks on the formulations in [8,9].

Formal language: Z uses the well known schema calculus, in which a schema consists of named and typed components which are constrained by a formula built up using the usual logical primitives. This is an all-purpose machinery; 'delta' schemas enable before-after relations that specify transitions to be defined; other schemas define retrieve relations, etc. The schema calculus itself enables schemas to be combined so as to express statements such as the POs of a given refinement theory.

Granularity and naming: Most of the refinement formulations in [8,9] stick to a $(1,1)$ framework. Purely theoretical discussions often strengthen this to identity on 'indexes' (i.e. names) of operations at abstract and concrete levels, though there is no insistence on such a close tieup in [10,11].

Concrete-abstract fidelity: In the above context for Z refinement, the fidelity PO comes out as follows, which refers to the contract interpretation without I/O (while the behavioural interpretation drops the 'pre AOp'):

$$\forall\, AState;\ CState;\ CState' \bullet \mathrm{pre}\, AOp \wedge R \wedge COp \Rightarrow \exists\, AState' \bullet R' \wedge AOp \quad (1)$$

where *AState*, *CState* are (abstract and concrete) state schemas (primes denote after-states), *AOp*, *COp* are corresponding operations, *R* is the retrieve relation, and 'pre *AOp*', the precondition, in fact denotes the domain of *AOp*.

Notions of correctness: In Z, an induction on execution steps is used in the $(1,1)$ framework to derive trace inclusion. To work smoothly, totality (on the state space) of the relations expressing operations is assumed. To cope with partial operations, a $\perp$ element is added to the state space, and *totalisations* of one kind or another, of the relations representing the operations, are applied. The consequences of totalisation (such as (1)), got by eliminating mention of the added parts from a standard forward simulation implication, constitute the POs of, and embody the notion of correctness for, the totalisation technique under consideration. These turn out to be the same for both contract and behavioural approaches, aside from the difference in (1) noted above.

Interpretation: The two main totalisations used, express the *contract* and *behavioural* interpretations. In the former, an operation may be invoked at any time, and *the consequences of calling it outside its precondition are unpredictable* (within the limits of the model of the syntax being used), including $\perp$, nontermination. In the latter, $\perp$ is guaranteed outside the precondition (usually called the guard in this context, but still defined as the domain of the relevant partial relation), which is typically interpreted by saying the operation *will not execute* if the guard is false.

Trace inclusion: Trace inclusion has been cited as the underlying derivation technique for the POs, and since an inductive approach is used, it is strong trace inclusion. However, the 'fictitious' transitions of operations introduced by totalisation are treated on an equal footing to the original 'honest' ones, so many spurious traces, not corresponding to real world behaviour, can be generated. For instance a simulation of a concrete trace may hit a state (whether abstract or concrete) that is outside the 'natural' domain of the *next* partial operation. Then, in the contract interpretation, the trace can continue in a very unrestricted manner, despite the different way that one would view the constituent steps from a real world perspective. Things look a bit better in the behavioural interpretation, since such a trace is thereafter confined to $\perp$.

Composition: One prominent composition mechanism to be found in Z is *promotion*. In promotion, a component which is specified in a self-contained way is replicated via an indexing function to form a family inside a larger system; this interacts cleanly with refinement [8,9]. However, the schema calculus in general is not monotonic with respect to refinement without additional caveats [12].

3.2 B

The original B Method was described with great clarity in [13], and there are a number of textbook treatments eg. [14,15,16].

Formal language: Original B was based on predicates for subsets of states, written in a conventional first order language, and on weakest precondition predicate transformers (wppts) for the operations. The use of predicate transformers obviates the need for explicitly adjoining $\perp$ elements to the state spaces.

Granularity and naming: Original B adheres to a strict $(1, 1)$ framework; 'strict' in the sense that tools for original B demand identical names for operations and their refinements. Abstract models of complex operations can be assembled out of smaller pieces using such mechanisms as INCLUDES, USES, SEES. However once the complete abstract model has been assembled, refinement proceeds monolithically towards code. The last step of refinement to code, is accomplished by a code generator which plugs together suitably designed modules that implement the lowest level B constructs.

Concrete-abstract fidelity: This is handled via the predicate transformers. Adapting the notation of [13] for ease of comparison with (1), the relevant PO can be written:

$$AInv \wedge CInv \wedge \mathsf{trm}\, AOp \Rightarrow [\ COp\ ] \neg [\ AOp\ ] \neg CInv \tag{2}$$

In this, *AInv* and $\mathsf{trm}\ AOp$ are the abstract invariant and termination condition (the latter being the predicate of the precondition), while *CInv* is the concrete invariant, which in original B, involves both abstract and concrete variables and thus acts also as a retrieve relation; all of these are predicates. $[\ AOp\ ]$ and $[\ COp\ ]$ are the wppts for the abstract and concrete operations, so (2) says that applying the concrete and 'doubly negated' abstract wppts to the after-state retrieve relation yields a predicate (on the before-states) that is implied by the before-state quantities to the left of the implication.

Notions of correctness: In original B, precondition (trm) and guard (fis) are distinct concepts (unlike Z), albeit connected by the implication $\neg\ \mathsf{trm} \Rightarrow \mathsf{fis}$, due to the details of the axiomatic way that these two concepts are defined. Moreover, $\mathsf{trm} \wedge \neg\ \mathsf{fis}$ can hold for an operation, permitting *miracles*, a phenomenon absent from formalisms defined in a purely relational manner. In original B, trm is a conjunct of any operation's definition, so outside trm, nothing is assumed, and when interpreted relationally, it leads to something like a 'totalisation' (though different from the Z ones). During refinement, the precondition is weakened and the guard is strengthened, the former of which superficially sounds similar to Z, though it is again different technically.

Interpretation: The interpretation of operation steps for which trm and fis both hold is the conventional unproblematic one. Other steps fire the imagination. If trm is false the step *aborts*, i.e. it can start, but not complete normally; modelled relationally by an unconstrained outcome, a bit like contract Z. If fis is false the step does not start normally, but can complete; a miracle indeed, usually interpreted by saying that the step will not take place if fis is false.

Trace inclusion: In original B, trace inclusion is not addressed directly, but as a consequence of monotonicity. Refinement is monotonic across the B constructors, including sequential composition. This yields a notion of weak trace inclusion, since the trm and fis of a composition are an *output* of a composition calculation, not an input, and in particular, cannot be assumed to be the trm and fis of the first component, as one would want if one were extending a simulation by considering the next step. And even though the sufficient condition for fidelity (2) is a strengthening of the natural B refinement condition, it does not lead to an unproblematic strong trace inclusion, since in a relational model, we have the additional transitions generated by the 'totalisation', and miracles do not give rise to actual transitions.

Composition: In original B, the interaction of refinement and composition is not a real issue. The earlier INCLUDES, USES, SEES mechanisms are certainly composition mechanisms, but they just act at the top level. Only the finally assembled complete abstract model is refined, avoiding the possibility of Z-like nonmonotonicity problems. The IMPORTS mechanism allows the combination of independent developments.

3.3 Event-B

Event-B [17,18,19] emerged as a focusing of original B onto a subset that allows for both more convenient practical development, and also an avoidance of the more counterintuitive aspects of the original B formalism, such as miracles.

Formal language: Event-B is rooted in a traditional relational framework, derived by restricting original B operations (henceforth called events) to have a trm which is true, and controlling event availability purely via the guard, which is the domain of the event transition relation, as in Z. Distinguishing between guard and event in the syntax enables event transitions to be defined via convenient notations (such as assignment) which are more widely defined than the desired guard. Formally, the more exotic possibilities afforded by predicate transformers are no longer needed.

Granularity and naming: Event-B relaxes the strict $(1,1)$ conventions of original B. As in original B, the syntax of the refinement mechanism is embedded in the syntax of the refining machine, so an abstraction can be refined in more than one way, but not *vice versa*. However, a refining event now names its abstract event, so an abstract event can have several refinements within the same refining machine. New events in a refining machine are *implicitly* understood to refine an abstract skip, something which needed to be stated explicitly in original B, cluttering incremental development.

Concrete-abstract fidelity: The absence of the more exotic aspects of predicate transformers gives the Event-B fidelity PO a quite conventional appearance:

$$\forall\, u, v, v' \bullet AInv \wedge CInv \wedge G_{CEv} \wedge CEv \Rightarrow \exists\, u' \bullet AEv \wedge CInv' \tag{3}$$

This says that assuming the abstract invariant and the concrete invariant (which is again a joint invariant i.e. retrieve relation) and the concrete guard and concrete transition relation for the before-states, yields the existence of an abstract event which re-establishes the joint invariant in the after-states.

Notions of correctness: The absence of preconditions distinct from guards simplifies matters considerably. The previous 'weakening of the precondition' during refinement of an operation, is now taken over by 'disjunction of concrete guard with guards of all new events is weaker than the abstract guard'. This is a quite different criterion, which nevertheless guarantees that if something can happen at the abstract level, a 'suitable' thing is enabled at the concrete level. This is also combined with guard strengthening in the refinement of individual events, and a well foundedness property to prevent new events from being always enabled relative to old events. Totalisations are no longer present in any form, which has an impact on trace inclusion (see below).

Interpretation: The absence of preconditions distinct from guards simplifies interpretational matters considerably. There is a firm commitment to the idea that events which are not enabled do not execute, avoiding the need to engage with miracles and with spurious transitions generated by totalisation.

Trace inclusion: In the Event-B context, trace inclusion wins massively. Since for a refined event, the concrete guard implies the abstract one, the implication has the same orientation as the implication in (3), so the two work in harmony to enable any concrete step joined to an appropriate abstract before-state, to be unproblematically simulated, a phenomenon not present in formalisms mentioned earlier — simulated moreover, by a 'real' abstract event, not a fictitious one introduced via totalisation. New events do not disturb this, since they are by definition refinements of skip, which can always effortlessly simulate them. So we have genuine, uncluttered, strong trace inclusion.

Composition: Event-B takes a more pro-active approach to composition than original B. Event-B's top-down and incremental approach means that system models start out small and steadily get bigger. This allows composition to be instituted via *decomposition*. As a system model starts to get big, its events can be partitioned into subsystems, each of which contains *abstractions* of the events not present. These abstractions can capture how events in different subsystems need to interact, allowing for independent refinement, and avoiding the non-monotonicity problems mentioned earlier.

3.4 ASM

The Abstract State Machine approach developed in a desire to create an operationally based rigorous development framework at the highest level of abstraction possible. A definitive account is given in [6].

Formal language: Among all the methodologies we survey, ASM is the one that de-emphasises the formality of the language used for modelling the most — in a laudable desire to not dissuade users by forcing them to digest a large amount of technical detail at the outset. System states are general first order structures. These get updated by applying ASM rules, which modify the FO structures held in one or more *locations*. In a departure from the other formalisms reviewed, *all* rules with a true guard are applied simultaneously during an update.

Granularity and naming: The ASM approach tries as hard as it can to break the shackles of imposing, up front, any particular scheme of correspondence between abstract and concrete steps during refinement. Since a retrieve relation has to be periodically re-established, a practical technique that breaks a pair of simulating runs into (m, n) diagrams of m abstract steps and n concrete ones (for arbitrary finite $m + n > 0$), without any preconceptions about which steps occur, is minimally demanding.

Concrete-abstract fidelity: In striving to be as unrestrictive as possible, ASM does not prescribe specific low level formats for establishing refinement. However, one technique, generalised forward simulation, established by Schellhorn [20] (see also [21]), has become identified as a *de facto* standard for ASM refinement. This demands that the (m, n) diagrams mentioned above are shown to be simulating by having a 'working'

retrieve relation $\approx$, which implies the actual retrieve relation $\equiv$, which itself is referred to as an *equivalence*. The $\approx$ relation is then used in implications of the form (1)-(3), except that several abstract or concrete steps (or none) can be involved at a time. As many (m,n) diagram simulations as needed to guarantee coverage of all cases that arise must then be established.

Notions of correctness: It has already been mentioned that $\equiv$ is referred to as an equivalence. While almost all retrieve relations used in practice are in fact partial or total equivalences [22], knowing this *a priori* has some useful consequences. It leads to a simple relationship between the guards of the run fragments in simulating (m,n) diagrams, subsuming guard strengthening, and eliminating many potential complications. Refinement is defined directly via a trace-inclusion-like criterion (periodic reestablshment of $\equiv$), and for $(0,n)$ and $(m,0)$ diagrams, there is a well foundedness property to prevent permanent lack of progress in one or other system in a refinement. The analogue of 'precondition weakening' (though we emphasise that there is no separate notion of precondition in ASM) is subsumed by the notion of 'complete refinement' which demands that the abstract model refines the concrete one (as well as *vice versa*), thus ensuring that any time an abstract run is available, so is a suitable concrete one, yielding persistence of coverage of requirements down a refinement chain. Of course not all refinements need to be complete, permitting convenient underspecification at higher levels, in a similar manner to Event-B.

Interpretation: Since states and transitions are defined directly, there are no subtle issues of interpretation associated with them. Also, ASM rule firing is a hardwiring of the 'transitions which are not enabled do not execute' convention into the formalism.

Trace inclusion: The (m,n) diagram strategy of ASM modifies the notion of trace inclusion that one can sustain. The ASM (m,n) notion, at the heart of the ASM *correct refinement* criterion, can be viewed as a generalisation of the Event-B $(1,1)$ strategy.

Composition: With the major focus being on identifying the ground model, and on its subsequent refinement (rather as in original B), the composition of independent refinements is not prominent in [6,21]. On the other hand, if $\equiv$ *really is* an equivalence (or as we would need to have it between two state spaces which are different, a *regular* relation a.k.a. a *difunctional* relation), there is a beneficial effect on any prospective composition of refinements. Many of the issues noted in [12] arise, because incompatible criteria about abstract sets (of states, say) which are unproblematic due to the abstract sets' disjointness, can become problematic due eg. to precondition weakening when the sets' concrete retrieve images become non-disjoint via a non-regular retrieve relation. A regular retrieve relation does much to prevent this, facilitating composition of refinements.

4 Configurable Semantics

The preceding sections very briefly surveyed a few well known refinement paradigms. Although it might not be as apparent as when one examines more of the details in each case, it is easy to be struck by how so many of the issues we have highlighted,

turn out merely to be *design decisions* that happen to have been taken, about some particular feature, in the context of one or other formalism. Although some such design decisions are interrelated, one can very easily imagine, that in many cases, a given design decision about some aspect of a refinement methodology, could just as easily have been implemented in the context of a methodology different from the one in which we happen to find it. Here are a few examples.

- Regarding Z, one could easily imagine its notion(s) of correctness being substituted by the ones from Event-B or ASM. Its notion of trace inclusion would then be replaced by one not requiring the use of 'fictitious' transitions generated by totalisation.
- For B, one could easily imagine adding $\perp$ elements to state spaces etc. in order to obtain a different relational semantics, with fewer 'fictitious' transitions.
- For Event-B and ASM one could imagine bringing in some aspects of the Z modelling, though it appears that little would be gained by doing so.

Of course such ideas are not new. In many cases, for mature methodologies, alternatives of one kind or another have been investigated, whether in the normal research literature or as student research projects — making an even moderately comprehensive list of the cases covered would swell the size of this paper unacceptably.

Semantic modifications of the kind hinted at can serve a more serious purpose than mere curiosity. In ProB [23], a model checker and animator for the B-Method first implemented for original B, the original B preconditions are re-interpreted as (i.e. given the semantics of) additional guards. The reason for this is that preconditions are *weakened* during refinement, whereas guards are *strengthened*. As already noted in Section 3.3, the orientation of the latter implication is the same as that in the fidelity PO, so the two collaborate in establishing trace inclusion. Precondition weakening is in conflict with this, so the ProB adaptation is necessary to ensure that the theoretical constructions at the heart of model checking remain valid.

Commenting from a real world developer's perspective, the fewer the extraneous and counterintuitive elements that a formalism contains, the more appealing it becomes for real world use. For example, if an applications sphere features operations that are intrinsically partial, then that is all that there ought to be to the matter, and consequently, the approach of totalising such operations becomes an artificial distraction, potentially even a misleading one if the fictitious transitions could be mistaken for real ones.

Such techniques as totalisation can be seen as making the task of *setting up* the semantics of a formal framework simpler. However, the real world developer's priorities are more focused on accurate modelling of the application scenario, and this can motivate a modification of the semantics, albeit at the price of additional formal complexity. In the Météor Project [24], the semantics of original B was modified to explicitly check well-definedness conditions for applications of (partial) functions, using techniques going back to Owe [25], in recognition of this application need. Event-B, a more recent development, has such checks built in *ab initio*, and its semantics fits model checking needs much better too, as already noted.

The above thoughts, assembled with the wisdom of hindsight, drive one to the conclusion that the semantics of formal development notations would be better designed

in a more *flexible*, or *configurable* way. The idea that a single pre-ordained semantic framework can cover all cases in all needed application scenarios is hard to sustain.

Such a viewpoint has consequences of course, both theoretical and practical. Theoretically, one would have to structure the theory of a particular formalism so that contrasting design decisions could be adopted straightforwardly, in a way that avoided confusing the reader, and so that the consequences of adopting alternatives could easily be imagined. Moreover, doing this would not constitute a huge overhead since theoretical work is relatively cheap. Practically though, it is a different matter. Practically, formalisms, such as the ones we have discussed, are embodied in tools; and creating a good tool requires a considerable investment. We discuss the wider consequences of our perspective for tools in the next section.

A final thought on the topic of semantic flexibility. One cannot help notice from the above brief discussion, that the places where semantic modifications have been imposed on a technique in order to satisfy application development methodology needs, have all occurred in the 'notions of correctness' and 'interpretation' areas. Notably free from interference has been the 'concrete-abstract fidelity' area. This indicates a strong consensus among approaches that simulation (in one form or another) is *the* key criterion that techniques must establish. Other issues from Section 2, such as 'formal language', 'granularity and naming' and 'trace inclusion', 'composition', can be seen as either enablers for setting up a framework, or derivable consequences of the design decisions taken. This in turn suggests a scheme for organising theories in this field: one sets up the linguistic machinery, one sets up concrete-abstract simulation, one chooses additional correctness and accompanying concepts, and then one derives whatever additional properties of interest follow from the preceding choices. And when comparing or combining one formalism with another, it is the *intersection* of features rather than their *union* that is of greatest importance.

5 Issues for Tools

The considerations of the preceding sections have implications for tool design, as already noted. Up to now, most tools in this arena have been based on a commitment to a particular set of design decisions about various semantic issues, and these decisions, howsoever arrived at, have been hardwired into the structure of the tool, making tools somewhat monolithic. This has the advantage that with each tool, one knows exactly what one is getting. However, it also has the disadvantage that it isolates tools from each other, and makes tool interoperability difficult or impossible.

These days, it is more and more recognised that to best address the risks inherent in the whole process of a system development, it is desirable to utilise a range of techniques and to interconnect them. A consequence of the isolation between tools is that it is difficult to simultaneously capitalise on the strengths of more than one. It also means that when an advance is made in one tool, other tools have to duplicate the work involved before similar ideas can be used in the other contexts. One way of addressing this difficulty is to not only make the various theoretical frameworks flexible and configurable, as recommended earlier, but to also make the tools that support them more

configurable and *programmable*. We now discuss three approaches to this as exemplified within three different tool environments.

The **Rodin Toolset** [18] for supporting the Event-B methodology, is built on Eclipse [26], a flexible platform for software development which manages dependencies between development artifacts and supports a GUI for displaying them. The semantic content of a methodolgy supported by an Eclipse-based tool is captured via a collection of Eclipse plugins. Rodin is thus a collection of plugins for introducing Event-B machines and contexts, editing them, checking them, generating POs, supporting PO proof, and general housekeeping. Other plugins exist for LaTeX printing, ProB support, and support for additional development activities to aid Event-B development is planned or can easily be envisaged. Since the source of Rodin is in the public domain, one can integrate such additional activities by simply writing more plugins of one's own. If one wished to actually *alter* specific semantic elements of Event-B for any reason, one might well have to *replace* an existing plugin by a different one, since the standard semantics of Event-B is hardwired into the plugins, if not into Eclipse. This, although possible, is not trivial, since writing Eclipse plugins, especially ones that would have to collaborate closely with other existing ones, is not an easy task. Counter to this relative inflexibility, we note that a certain limited amount of semantic flexibility has been built into Rodin *ab initio*, since one can configure certain attributes of events, eg. whether they are *ordinary*, *convergent*, etc. This influences the verification conditions that are generated.

The **Frog tool** [27,28] is an experimental tool, originally designed for mechanically supporting retrenchment [29], whose inbuilt flexibility addresses our concerns very well. In Frog, much of what is hardwired in typical existing proof-driven development tools is programmable. Thus there is an intermediate language (Frog-CCL) for declaring the *structure* of the clauses that comprise the usual syntactic constructs that constitute a typical formal development framework. Paradigmatically, one has machine definitions, relationships between machines and the like. In Frog, the mathematical ingredients of all the constructs are specified using Z schemas, thus exploiting Z's essence as a general purpose formal mathematical notation. Since relationships between constructs, such as refinements, are themselves syntactic constructs, the precise nature of what constitutes a refinement (in terms of the POs that characterise it), can be precisely specified and configured using Frog-CCL scripts. Designing a complete formal development methodology in Frog is thus a matter of writing several Frog-CCL scripts, rather than a major development task. At least that is so in principle. Due to limited time during Simon Fraser's doctorate, certain things are still hardwired in Frog, such as: the use of Z as mathematical language, the use of the Isabelle theorem prover [30], and a strict $(1,1)$ naming convention for operations. Evidently, more flexibility could easily be contemplated for these aspects.

Of course the maximum flexibility for adapting the semantic and/or any other aspects of a methodology whilst still within a tool environment, is to work with a fairly general purpose theorem prover. There are essentially no constraints when one takes this approach, since, regardless of what features are taken as constituting the foundations of a given formal development methodology (and there is considerable variation on what is regarded as fundamental among different methodologies), the verification that a

particular development is correct with respect to that particular methodology, always reduces to constructing proofs (of a fairly conventional kind) of a number of posited properties of the development, the verification conditions. The flexibility of the general purpose theorem prover approach has been demonstrated with great success in deploying the **KIV Theorem Prover** [31] to address system development in the ASM methodology (and others). The web site [32] gives full details of the mechanical verification of a number of substantial developments, carried out under mechanical formalisations of a variety of detailed refinement formalisms. The approach has enjoyed particular success in the context of the mechanical verification of Mondex [33,34]. The generality of KIV enabled previously investigated refinement strategies to be quickly adapted to the details of Mondex, and the whole of the verification, done in competition with several international groups, to be accomplished in record time.

6 Conclusions

In this paper, we have examined some key features of a small number of well known refinement methodologies, and commented on their similarities and differences. We noted that many of their features were not especially specific to the methodologies in which they were found, and that we could just as easily transplant them into others. We also observed that applications considerations can influence and adapt such methodologies, irrespective of first principles, belying the view that their semantics are sacrosanct.

The same considerations impact tool support, but more deeply, given the investment needed to create a good tool. Accordingly, we turned our attention to strategies for achieving greater tool flexibility: from Rodin's plugins, to Frog's scripting approach, to theorem proving using eg. KIV. While the last of these undoubtedly offers the greatest flexibility, it also requires the greatest expertise, and for more everyday development environments, some tool-imposed discipline is probably necessary. The question is how to achieve an adequate level of tool supervision without compromising openness, interoperability and flexibility. In the author's view, the Frog approach offers great promise for quick adaptability of the semantic details of a formal methodology, without demanding a huge investment in reprogramming the tool. It is easy to imagine that in a tool such as Frog, for industrial application, the programmable semantic aspects can be made editable only by senior personnel, and the majority of the development team see a tool which behaves as though its semantics was conventionally hardwired. In any event, all the approaches outlined above certainly offer promise, and further experimentation is to be expected in the near future.

All of the above is certainly in harmony with the call for an Evidential Tool Bus (ETB) [2], over which tools could communicate. In the ETB, tools are no longer envisaged as monolithic entities, isolated from each other, but rather as members of a community, each responsible for a subset of, or for a particular approach to, the overall verification task. Tools on the bus could make use of the (partial) evidence for correctness established by other tools on the bus, to enhance what they themselves would be able to achieve — they in turn publishing their own results on the bus for successor tools to benefit from. Thus the community could achieve, by cooperation, far more, far more cheaply, than any one tool could achieve on its own.

The preceding is also in harmony with the currently active Verification Grand Challenge [3,4,5]. This has many aims, from promoting formal techniques in the mainstream (on the basis of their by now well established capacity to deliver, to standard, on time, on budget, and overall more cheaply than by the use of conventional techniques), to establishing cadres of formally verified applications in a repository (as further evidence to encourage their uptake, and perhaps to provide thereby collections of reusable formally verified components), to encouraging the harmonisation and cooperation of formal techniques. This last aim is squarely aligned with our motivations for carrying out the analysis of refinement techniques given in this paper.

References

1. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. C.U.P (1998)
2. Rushby, J.: Harnessing Disruptive Innovation in Formal Verification. In: Proc. IEEE SEFM 2006, pp. 21–28. IEEE Computer Society Press, Los Alamitos (2006)
3. Jones, C., O'Hearne, P., Woodcock, J.: Verified Software: A Grand Challenge. IEEE Computer 39(4), 93–95 (2006)
4. Woodcock, J.: First Steps in the The Verified Software Grand Challenge. IEEE Computer 39(10), 57–64 (2006)
5. Woodcock, J., Banach, R.: The Verification Grand Challenge. Communications of the Computer Scociety of India (May 2007)
6. Börger, E., Stärk, R.: Abstract State Machines. A Method for High Level System Design and Analysis. Springer, Heidelberg (2003)
7. ISO/IEC 13568: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics: International Standard (2002), `http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip`
8. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. PHI (1996)
9. Derrick, J., Boiten, E.: Refinement in Z and Object-Z. FACIT. Springer, Heidelberg (2001)
10. Spivey, J.: The Z Notation: A Reference Manual, 2nd edn. PHI (1992)
11. Cooper, D., Stepney, S., Woodcock, J.: Derivation of Z Refinement Proof Rules. Technical Report YCS-2002-347, University of York (2002)
12. Groves, L.: Practical Data Refinement for the Z Schema Calculus. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 393–413. Springer, Heidelberg (2005)
13. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. C.U.P (1996)
14. Lano, K., Haughton, H.: Specification in B. Imperial College Press (1996)
15. Habrias, H.: Specification Formelle avec B. Hermes Sciences Publications (2001)
16. Schneider, S.: The B-Method. Palgrave (2001)
17. Abrial, J.R.: Event-B (to be published)
18. Rodin. European Project Rodin (Rigorous Open Development for Complex Systems) IST-511599, `http://rodin.cs.ncl.ac.uk/`
19. The Rodin Platform, `http://sourceforge.net/projects/rodin-b-sharp/`
20. Schellhorn, G.: Verification of ASM Refinements Using Generalised Forward Simulation. J.UCS 7(11), 952–979 (2001)
21. Börger, E.: The ASM Rrefinement Method. Form. Asp. Comp. 15, 237–257 (2003)
22. Banach, R.: On Regularity in Software Design. Sci. Comp. Prog. 24, 221–248 (1995)

23. 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)
24. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: 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)
25. Owe, O.: Partial Logics Reconsidered: A Conservative Approach. F.A.C.S. 3, 1–16 (1993)
26. The Eclipse Project, http://www.eclipse.org/
27. Fraser, S., Banach, R.: Configurable Proof Obligations in the Frog Toolkit. In: Proc. IEEE SEFM 2007, pp. 361–370. IEEE Computer Society Press, Los Alamitos (2007)
28. Fraser, S.: Mechanized Support for Retrenchment. PhD thesis, School of Computer Science, University of Manchester (2008)
29. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. 67, 301–329 (2007)
30. The Isabelle Theorem prover, http://www.cl.cam.ac.uk/research/hvg/Isabelle/
31. The Karlsruhe Interactive Verifier, http://i11www.iti.uni-karlsruhe.de/~kiv/KIV-KA.html
32. KIV: KIV Verifications on the Web, http://www.informatik.uni-augsburg.de/swt/projects/
33. 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)
34. Mondex KIV: Web presentation of the Mondex case study in KIV, http://www.informatik.uni-augsburg.de/swt/projects/mondex.html

A Concept-Driven Construction of the Mondex Protocol Using Three Refinements

Gerhard Schellhorn[1] and Richard Banach[2]

[1] Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de
[2] School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract. The Mondex case study concerns the formal development and verification of an electronic purse protocol. Several groups have worked on its specification and mechanical verification, their solutions being (as were ours previously), either one big step or several steps motivated by the task's complexity. A new solution is presented that is structured into three refinements, motivated by the three concepts underlying Mondex: a message protocol to transfer money over a lossy medium, protection against replay attacks, and uniqueness of transfers using sequence numbers. We also give an improved proof technique based on our theoretical results on verifying interleaved systems.

1 Introduction

Mondex smart cards implement an electronic purse [1]. They were the target of one of the first ITSEC evaluations at level E6 [2] (now EAL7 of Common Criteria [3]), which requires formal specification and verification. The formal specifications were given in [4] using Z [5], together with manual correctness proofs. 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.

Mechanizing the security and refinement proofs of [4] was recently proposed as a challenge for theorem provers (see [6] for more information on the challenge and its relation to 'Grand Challenge 6'). Several groups took up the challenge. For a survey see [7] — more details on some are given in Section 7. Results to date have been focused on solving the problem either as closely as possible to the original, or by adapting the problem to fit the style of the tool, thereby simplifying it.

The first author works in the Augsburg group, which uses KIV. This has derived solutions for both styles. [8] gives a solution that formalizes the original data refinement theory of [9] and uses the original backward simulation. Alternatively, since KIV supports the Abstract State Machines (ASM, [10], [11]) style of specifying operations, we have also given a solutions using ASMs in [12], as one refinement that uses generalized forward simulations of ASMs ([13], [14], [15], [16], [17]). This solution simplified

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 57–70, 2008.

the deduction problem by using purse-local invariants (inspired by [18]), and by using big commuting diagrams for full protocol runs, a technique used previously in ASM refinements. This approach also uncovered a weakness of the original protocol, which can be resolved by a small change. Still, the proof is monolithic, consisting of a single refinement.

Other authors, particularly [19] and [20], have tried to modularize the refinement into several to make deduction simpler, but from our point of view they have not isolated the Mondex *concepts* into separate refinements, allowing a clean explanation. However, their work has strongly influenced ours.

Isolating the Mondex concepts is a necessity when explaining the Mondex protocol live to an audience. This prompted the attempt to formalize them as separate refinements. The essential concepts of the Mondex protocol are the following:

- Implementing transfers by sending messages over a lossy transport medium.
- Adding checks that protect against replay attacks.
- A challenge-response system to ensure uniqueness of protocol runs.
- Choosing suitably strong cryptographic functions to encrypt messages.

This paper explains the first three concepts by putting them into three successive refinements. The fourth was absent in the original Mondex work: the Mondex concrete level *assumes* that suitable cryptography can be used to protect messages. Elsewhere [21], we have shown that suitable cryptography can indeed be added using another refinement, and that as an instance of a model-driven approach [22] the resulting ASM can be implemented using Java [23], so we do not repeat this here.

The next section recalls the Mondex abstract specification, and we then explain each of the refinements in turn in the following three sections. We also explain some of the simulation relations and invariants that are needed to verify each refinement with KIV (full details of all specifications and proofs are available at [24]). A final technical refinement, that slightly changes notation to be compatible with the original definitions completes the development. Finally, Section 7 gives related work and Section 8 concludes.

2 The Abstract Specification

The main protocol of Mondex implements electronic cash transfer, using either a device (wallet) with two slots, or an internet connection. Since the key idea is *cash*, the main security concern is that, even in a hostile environment, money cannot be created or destroyed (satisfying the security concerns of the bank and the customer, respectively).

The abstract specification formalizes atomic money transfer: a function balance : name $\rightarrow \mathbb{N}$ gives the current balance of each named purse (type name). A predicate authentic distinguishes legitimate purses from impostors. Successful money transfer is described by the TRANSFEROK rule below. This rule chooses the two authentic participating purses from and to and the amount value to transfer, which should be less or equal than balance(from), and modifies the balances according to a successful money transfer.

TRANSFEROK =
choose from, to, value
with authentic(from) $\wedge$ authentic(to) $\wedge$ from $\neq$ to $\wedge$ value $\leq$ balance(from)
in balance(from) := balance(from) $-$ value
balance(to) := balance(to) + value

In reality, transfers may fail, since power can fail, card memory can run out, and cards may detach from the protocol prematurely. According to the security requirement no money should be lost, so a mechanism must be implemented which saves information about failed transfers. The definition of this information will be the task of the first refinement. At the abstract level, it is simply assumed that there is another field lost: name $\rightarrow$ $\mathbb{N}$ on each card, which saves all outgoing money that is lost in failed transfer attempts, so that it can subsequently be recovered. The rule for failed transfer attempts is then simply

TRANSFERFAIL =
choose from, to, value
with authentic(from) $\wedge$ authentic(to) $\wedge$ from $\neq$ to $\wedge$ value $\leq$ balance(from)
in balance(from) := balance(from) $-$ value
lost(from) := lost(from) + value

With this rule it is obvious that the sum of all balance and lost values of all authentic purses never changes, so money can neither be created nor *truly* lost and the security goal is satisfied. This completes the description of the abstract specification. Runs of the system repeatedly apply ARULE = TRANSFEROK $\vee$ TRANSFERFAIL which chooses nondeterministically between the two possibilities.

3 From Atomic Transfers to Messages

The first refinement towards the Mondex protocol is concerned with implementing atomic transfer using a protocol that sends messages between the two purses, despite the fact that these messages may get lost for any number of reasons.

Sending a single Val(from,value) message from the from purse to the to purse containing the value and its sender will not do, since the message could be lost, and neither party would be able to prove that this happened.

An additional Ack(to,value) message acknowledging that the to purse has received the money improves matters: if the to purse sends this message when receiving money, but the from purse does not receive it, the from purse can write an *exception log* proving that something did not work: either the Val message was not processed properly or the Ack was lost. Dually, if the to purse sends a Req(to,value) message that requests the money, and the from purse only sends Val on receiving it, then the to purse can know that something did not work: either the request was not processed properly, or the Val was lost. Using all three messages enables detection of Val(from,value) message loss by inspecting both cards: the Val(from,value) message has been lost iff *both* purses have a suitable log entry.

Being able to detect failed transfers by checking both purse logs has one caveat: the two log entries must reliably belong to the *same transfer*. Otherwise a first attempt

could lose the Ack(to,value) message, creating a from log entry, and a second attempt could lose the Req(to,value) message, creating a fictitious "matching" pair. Therefore we will assume that each attempt to transfer money is equipped with a unique transfer identifier tid. The implementation of tid by sequence numbers is deferred to the third refinement; in this refinement we just assume there is a global finite set tids : set(tid) that stores the used identifiers and that it is always possible to choose a fresh one.

So, for the protocol we need messages Req(to, value, tid),Val(from, value, tid) and Ack(to, value, tid). Triples consisting of a name, a value and a tid are called *payment details*. They form the content of messages. Payment details are also remembered in exception logs which now replace the lost component. The logs are functions exLogfrom : name $\rightarrow$ set(paydetails) and exLogto : name $\rightarrow$ set(paydetails). To compute the abstract lost(from) we have to sum all values of payment details (to,value,tid) $\in$ exLogfrom(from) for which a matching (from,value,tid) in exLogto(to) exists (and this is already the main information needed for the simulation relation).

To allow message exchange, each purse now has a "message box" of messages awaiting processing. This is the function inbox : name $\rightarrow$ set(message). We first tried an inbox that contained one rather than several messages, but this turned out to be too restrictive, enforcing message sequencing between purses. Losing messages is realized by the following simple rule which may be invoked at any time by a purse receiver:

```
LOSEMSG =
choose msgs with msgs ⊆ inbox(receiver) in inbox(receiver) := msgs
```

Finally, a purse has to know which message it sent last so as to react to missing answers appropriately; function outbox : name $\rightarrow$ message does this. An outbox that is *not* in the middle of a protocol run, can contain either the last sent Ack, or the special value none, when it has not yet sent a message or successfully received an Ack. Both cases are checked with the predicate isNone.

With these data structures, we derive four rules: for sending requests (STARTTO), for receiving a request and sending a value (REQ), for receiving a value and sending an acknowledgement (VAL), and finally for receiving an acknowledgement (ACK).

Like LOSEMSG above, the STARTTO and REQ rules assume that an authentic purse receiver has been chosen to execute the rule. Note that STARTTO chooses a new transfer identifier and is possible only when the purse is currently not involved in a protocol (i.e. when isNone(outbox(receiver))). Postfix selectors msg.pd, msg.value, msg.tid select the full payment details, the value and the tid contained in a message. **seq** is ASM notation to indicate sequential execution (usually all assignments are executed in parallel).

```
STARTTO =
if isNone(outbox(receiver))
then choose na, value, tid
with tid ∉ tids ∧ authentic(na) ∧ na ≠ receiver
in inbox(na) := inbox(na) ∪ {Req(receiver, value, tid)}
   outbox(receiver) := Req(na, value, tid)
   tids := tids ∪ {tid}
```

```
REQ =
choose msg
with   msg ∈ inbox(receiver) ∧ isReq(msg) ∧ authentic(msg.na)
     ∧ msg.na ≠ receiver ∧ msg.value ≤ balance(receiver)
     ∧ isNone(outbox(receiver)) in
  inbox(msg.na) := inbox(msg.na) ∪ {Val(receiver, msg.value, msg.tid)}
  outbox(receiver) := Val(msg.pd)
  balance(receiver) := balance(receiver) − msg.value seq
  inbox(receiver) := inbox(receiver) \ {msg}
```

The VAL rule is similar to REQ: the input is checked to be a Val(pd) message, where the outbox must be Req(pd) with the same payment details pd, and the sent message placed in inbox(msg.na) is an Ack. Also msg.value is added to the balance instead of subtracted. The ACK rule is similar too, but does not change the balance, does not write any output message and sets the outbox to none.

```
VAL =
choose msg
with   msg ∈ inbox(receiver) ∧ isVal(msg) ∧ isReq(outbox(receiver))
     ∧ msg.pd = outbox(receiver).pd in
  inbox(msg.na) := inbox(msg.na) ∪ {Ack(receiver, msg.value, msg.tid)}
  outbox(receiver) := Ack(msg.pd)
  balance(receiver) := balance(receiver) + msg.value seq
  inbox(receiver) := inbox(receiver) \ {msg}


ACK =
choose msg
with   msg ∈ inbox(receiver) ∧ isAck(msg) ∧ isVal(outbox(receiver))
     ∧ msg.pd = outbox(receiver).pd in
  outbox(receiver) := none
  inbox(receiver) := inbox(receiver) \ {msg}
```

Finally, a purse can abort a protocol (for whatever reason); it then executes

```
ABORT =
if isReq(outbox(receiver))
then exLogto(receiver) := exLogto(receiver) ∪ {outbox(receiver).pd} seq
if isVal(outbox(receiver))
then exLogfrom(receiver) := exLogfrom(receiver) ∪ {outbox(receiver).pd} seq
outbox(receiver) := none
```

The full specification chooses an authentic receiver and nondeterministically executes one of the above rules

```
IRULE = choose receiver with authentic(receiver) in
            LOSEMSG ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT
```

In [12], [17] we have proposed the use of purse-local simulation relations and invariants to verify refinements that split up an atomic action into several protocol steps. The approach described there for the simulation relation could be used unchanged. The invariants used in the approach state that "each purse has executed some (maybe no) steps into the protocol". Such invariants are easily expressible in KIV's Dynamic Logic.

Our research in [25,26] has established a general framework, that suggests invariants should be *protocol-local*, not *purse-local*. Therefore we generalized the approach to use the following idea for invariants:

> "For every protocol already running (identified by a tid $\in$ tids), there are two purses from and to that have executed some of the steps of the protocol. These steps determine the part of the state involving tid"

To get a formal definition involving ASM rules we have to do two things. Firstly, we have to formalize "some protocol steps". For the Mondex protocol these are

(1) no step.
(2) STARTTO and possibly an ABORT of the to purse.
(3) STARTTO, then REQ, then possibly ABORT(from) or ABORT(to) or both.
(4) STARTTO, REQ, and VAL and then possibly ABORT(from).
(5) The full protocol STARTTO, REQ, VAL and ACK.

The states reached by executing some steps of the protocol therefore correspond directly to the final states st1 of the nondeterministic program

```
SOMESTEPS(st,from,to) =
(1)   skip ∨
(2)   STARTTO; {skip ∨ ABORT(to) ∨
(3)     REQ; {{skip ∨ ABORT(to)};{skip ∨ ABORT(from)} ∨
(4)       VAL; {skip ∨ ABORT(from) ∨
(5)         ACK}}}
```

when started some initial state st, where tid was still unused (tid $\notin$ tids). Note that the parameters from and to were dropped where it was obvious: STARTTO is really STARTTO(to), i.e. the to purse is used in the STARTTO rule in place of receiver. The fact, that st1 is a final state of some terminating run of SOMESTEPS is expressed, using Dynamic Logic [27] in KIV, as

$\langle$SOMESTEPS(st,from,to)$\rangle$ st = st1

but the approach is not tied to the ASM formalism and Dynamic Logic: using a relational encoding of ASM rules, a relation somesteps could be defined similarly to the SOMESTEPS program above, using relational composition instead of compounds. The Dynamic Logic formula would then be equivalent to

somesteps(st,from,to,st1)

Secondly, we have to give a formal equivalent of the assertion "the protocol steps determine the part of the state involving tid". The part of the state that involves tid is easy to define. It consists of those messages in in- and outboxes, and those exception logs, that have tid in their payment details. To define what it means for the protocol steps to determine the part of the state that involves tid, we define a predicate eqtid(tid,st1,st2). This predicate compares two states. The first state st1 is the final state of running *just* the protocol steps involving tid from some initial state. It is a possible result of running SOMESTEPS. The second state st2 is the result of running these protocol

steps interleaved with an arbitrary number of protocol steps of other protocol instances. eqtid(tid,st1,st2) specifies that indeed the parts of the state involving tid of st1 and st2 are equal, since other protocol instances cannot interfere with the current protocol instance. There are two small exceptions: LOSEMSG may delete messages from inboxes, and a final Ack-message in an outbox may be overwritten after the current protocol has finished.

Putting together the SOMESTEPS program and the eqtid predicate we get the following invariant that encodes the informal idea given above:

$$\begin{array}{l} \mathsf{INV(st2)} \leftrightarrow \\ \quad \forall\, \mathsf{tid} \in \mathsf{tids2}.\ \exists\, \mathsf{from, to, st, st1}. \\ \qquad \mathsf{tid} \notin \mathsf{tids} \wedge \langle \mathsf{SOMESTEPS(st)} \rangle\ \mathsf{st} = \mathsf{st1} \wedge \mathsf{eqtid(tid,st1,st2)} \end{array}$$

The formula states, that for every protocol currently running (tid $\in$ tids2) there was an initial state st before the protocol was started and two participants from,to, such that the current part of the state involving tid is nearly the same (eqtid) as the state resulting from some terminating run of SOMESTEPS(st).

This is already the full invariant that is needed, except for the trivial invariant stating that no messages in inboxes, outboxes or exception logs, mention a tid that is not yet in tids.

The invariance proof reduces to proofs for every single protocol instance (i.e. for every tid), and for every protocol step. It has two cases: either the protocol step executed is one of the steps of the protocol instance for tid or it is a step of some other protocol instance. In the first case we essentially get trivial proof obligations, since "some steps of the protocol have been executed" is trivially invariant when executing yet another step. Essentially these proof obligations check that SOMESTEPS indeed encodes *all* possible protocol runs. For the second case we have to prove that steps of other protocol instances will not create messages or exception logs involving tid. The proof obligations check that eqtid correctly captures all potential interference from the other protocol instances.

Compared to the earlier proof of the full refinement in one step [12], which used purse-local invariants (which already simplified the many invariants needed for the original proof [4] that we used in [21]), the invariant has again been simplified: the purse-local approach required predicate logic properties that related the two states of the from and to purses participating in a protocol run. These are not needed any more.

4 Protection against Replay Attacks

Our next refinement is concerned with protection against replay attacks. The original development assumed that Req, Val, Ack messages are cryptographically protected, so we do the same. So an attacker cannot create such messages.

But even with this assumption, an attacker could destroy money on a from purse by saving and replaying Req and Ack messages. Indeed our first protocol is vulnerable to such an attack. For the new level, we assume an attacker who can intercept (and/or delete) messages, save them, and replay them. To model this formally, we assume a global set of messages ether : set(message) that contains at most all messages that

were sent so far. Since the union of all inboxes is a subset of ether, we can delete the inboxes altogether from the ASM state and let purses pick a message directly from ether. This corresponds to the attacker's ability to intercept and replace the message sent to a purse. Placing messages into a global ether instead of the inbox of the recipient has as immediate consequence: the intended recipient of the message must now be a component of the payment details of messages, and must be checked to be correct by the actual recipient. Otherwise the attacker could redirect messages from one purse to another. Since the attacker can still delete messages and messages might still be lost, LOSEMSG becomes

LOSEMSG = **choose** msgs **with** msgs $\subseteq$ ether **in** ether := msgs

To protect against replay attacks the states of purses must be enhanced with usedTids : name $\rightarrow$ set(tid), which gives the tids a purse receiver has seen previously. When a purse receives a Req, it saves the tid, and subsequently rejects messages with these transfer ids. Note that it is not necessary to add the tid of a Val message to the usedTids: accepting such a message only when the last sent message was a Req with the same payment details (and which must therefore have had a new tid!) is enough. This gives the following new rules for sending and receiving messages:

STARTTO =
choose na, value, tid **with** tid $\notin$ tids $\wedge$ authentic(na) $\wedge$ na $\neq$ receiver **in**
if isNone(outbox(receiver))
then ether := ether $\cup$ {Req(na, receiver, value, tid)}
outbox(receiver) := Req(na, receiver, value, tid)
tids := tids $\cup$ {tid}

REQ =
choose msg **with** msg $\in$ ether **in**
if isReq(msg) $\wedge$ msg.from = receiver $\wedge$ authentic(msg.to)
$\wedge$ msg.to $\neq$ receiver $\wedge$ msg.value $\leq$ balance(receiver)
$\wedge$ msg.tid $\notin$ usedTids(receiver) $\wedge$ isNone(outbox(receiver))
then ether := ether $\cup$ {Val(msg.pd)}
outbox(receiver) := Val(msg.pd)
balance(receiver) := balance(receiver) $-$ msg.value
usedTids(receiver) := usedTids(receiver) $\cup$ {msg.tid}

VAL =
choose msg **with** msg $\in$ ether **in**
if isVal(msg) $\wedge$ isReq(outbox(receiver)) $\wedge$ msg.pd = outbox(receiver).pd
then ether := ether $\cup$ {Ack(msg.pd)}
outbox(receiver) := Ack(msg.pd)
balance(receiver) := balance(receiver) + msg.value

ACK =
choose msg **with** msg $\in$ ether **in**
if isAck(msg) $\wedge$ isVal(outbox(receiver)) $\wedge$ msg.pd = outbox(receiver).pd
then outbox(receiver) := none

Aborting can now be simplified slightly: since the payment details contain the names of both purses, there is no further need to distinguish exLogfrom and exLogto. A single exLog: name → paydetails will do, and ABORT becomes

ABORT = **if** isReq(outbox(receiver)) ∨ isVal(outbox(receiver))
then exLog(receiver) := exLog(receiver) ∪ {outbox(receiver).pd} **seq**
outbox(receiver) := none

All together we have:

ERULE = **choose** receiver **with** authentic(receiver) **in**
LOSEMSG ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT

Whereas the previous refinement splits atomic steps into a protocol, this one is a typical data refinement: abstract and concrete rules correspond pairwise.

The simulation relation needed for verification consists of three parts:

- The union of all inboxes is always a subset of the ether.
- All requests in ether can only be in an inbox, if they have a tid that is not in usedTids(from) of the from purse that this message is sent to.
- Enhancing the union of the two logs exLogfrom(receiver) and exLogto(receiver) with "receiver" as a new component of the payment details gives exLog(receiver) for each authentic purse receiver.

Three invariants are needed for the concrete level:

- ether contains only messages with authentic names of different purses.
- tid's saved in the outbox-, exLog- or usedTids- field of an authentic purse are always also in tids.
- outbox(receiver) has payment details enhanced with "receiver" as to/from component for Req and Ack/Val.

5 Sequence Numbers as Challenges

The next refinement guarantees the uniqueness of protocol runs without using the global data structure tids. Instead we use a challenge-response scheme, like session keys, to ensure uniqueness. Mondex uses sequence numbers, which are used only once and then incremented. An alternative design decision would be to use random numbers ("nonces"). The state is now enhanced with a new component nextSeqNo : name → ℕ, while the global set tids and the usedTids of each purse are removed. To be secure, both purses participating in a protocol run provide and increment their nextSeqNo, guaranteeing that each abstract tid is implemented by a unique (fromseqno(tid), toseqno(tid)) pair; the two functions fromseqno and toseqno are the essence of the simulation relation. To ensure no faked sequence numbers get used, we need to send the sequence number as a challenge to both purses. For the from purse, Req can be used for the purpose. For the to purse a new message startTo(from,nextSeqNo(from),to, nextSeqNo(to),value), which is assumed to be encrypted too, is needed. On receiving a startTo/Req message, the to/from purse must check whether it contains the correct sequence number; both checks together guarantee, that Req and Val are never sent

on faked sequence numbers. Finally, for the from purse to send startTo, we need a startFrom(to,nextSeqNo(to),value) message, that sends nextSeqNo(to) to the from purse. This comes from the terminal, when the transfer amount has been entered. It need not be encrypted; at worst an invalid startTo message gets rejected by the to purse. For our ASM, we assume *all* startFrom messages are in the ether initially, modelling the ability of the attacker to generate such messages at will.

Note that this model deviates slightly from the original Mondex protocol [4], which assumes an unencrypted startTo, sent together with the startFrom, from the terminal. The original protocol cannot guarantee that a Req contains a correct nextSeqNo(to), and leads to the weakness described in [12].

The ASM of the resulting protocol is:

```
SRULE =
choose receiver with authentic(receiver) in
  LOSEMSG ∨ STARTFROM ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT


STARTFROM =
choose msg, n with msg ∈ ether ∧ nextSeqNo(receiver) < n in
if   isStartFrom(msg) ∧ authentic(msg.name) ∧ msg.name ≠ receiver
  ∧ msg.value ≤ balance(receiver) ∧ isNone(outbox(receiver))
then outbox(receiver) :=
          startTo(receiver, nextSeqNo(receiver)
                  msg.name, msg.nextSeqNo, msg.value)
nextSeqNo(receiver) := n
ether := ether ∪ {outbox(receiver)}


STARTTO =
choose msg, n with msg ∈ ether ∧ nextSeqNo(receiver) < n in
if   isStartTo(msg) ∧ authentic(msg.from) ∧ msg.from ≠ receiver
  ∧ msg.to = receiver ∧ msg.tono = nextSeqNo(receiver)
  ∧ isNone(outbox(receiver))
then outbox(receiver) := Req(msg.pd)
     nextSeqNo(receiver) := n
     ether := ether ∪ {Req(msg.pd)}


REQ =
choose msg with msg ∈ ether in
if   isReq(msg) ∧ isStartTo(outbox(receiver))
  ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := Val(msg.pd)
     balance(receiver) := balance(receiver) − msg.value
     ether := ether ∪ {Val(msg.pd)}


VAL =
choose msg with msg ∈ ether in
if isVal(msg) ∧ isReq(outbox(receiver)) ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := Ack(msg.pd)
     balance(receiver) := balance(receiver) + msg.value
     ether := ether ∪ {Ack(msg.pd)}
```

```
ACK =
choose msg with msg ∈ ether in
if isAck(msg) ∧ isVal(outbox(receiver)) ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := none

ABORT =
choose n with nextSeqNo(receiver) ≤ n in
if isReq(outbox(receiver)) ∨ isVal(outbox(receiver))
then exLog(receiver) := exLog(receiver) ∪ {outbox(receiver).pd} seq
       nextSeqNo(receiver) := n
       outbox(receiver) := none

LOSEMSG =
choose newether with newether ⊆ ether in ether := newether
```

The rules are largely unchanged except that tid's are replaced by pairs of sequence numbers. ABORT is now allowed to increment nextSeqNo to conform to the final Mondex protocol.

To verify the refinement we consider 1:1 diagrams for the common operations. The new STARTFROM step implements an abstract skip. The simulation relation asserts that two functions fromseqno and toseqno with domain = tids exist with the following three properties:

- outboxes, messages in ether and exception logs of the concrete level have tid replaced with fromseqno(tid) and toseqno(tid). There are two exceptions: an outbox of the concrete level may already contain a startTo of a new protocol run when the abstract outbox still satisfies isNone. The concrete ether may contain additional startFrom and startTo messages.
- If tid_1 and tid_2 appear in payment details of the abstract level with the same purses from and to, then $fromseqno(tid_1) \neq fromseqno(tid_2)$ or $toseqno(tid_1) \neq toseqno(tid_2)$. This guarantees that every protocol run between the same two purses uses a different pair of sequence numbers.
- If on the concrete level outbox(receiver) = startTo(pd) and Req(pd) ∈ ether, then there is a corresponding Req(pd) (with tid instead of sequence numbers) in the abstract ether and it's tid is not in usedTids(receiver). This property describes the new situation after sending a startTo message.

The concrete ASM also needs an invariant stating:

- outboxes never contain startFrom messages.
- The nextSeqNo of each purse is larger than any sequence number contained in any payment details in messages, inboxes, outboxes and exLogs.
- If outbox(receiver) contains a startTo, then the value of the message is less than or equal to balance(receiver).

6 Renaming to Use the Original Data Structures

The final refinement step is a purely technical one. It adjusts two small differences between SRULE and the final Mondex protocol. Since the full ASM was already given earlier in [12], we just give a short description of the differences.

In the real protocol, the outbox information is split into two: a pdAuth component which stores the payment details, and a status field, which stores the type of the last sent message: epr ("expecting request") for a startTo message, epv ("expecting value") for a Req message, epa ("expecting acknowledge") for a Val message, idle for an Ack message or none.

The second difference is a small change in control structure: nondeterministic choice between SRULE's disjuncts is replaced by deterministic choice over the type of message; if the test of the rule fails, an ABORT is executed. Finally, losing messages is done while adding a message to ether.

7 Related Work

The work of this paper is heavily based on the original work in [4] and the mechanized proofs in [7]. Several of the solutions described therein are monolithic (including our own); however, two structured the development into several refinements.

We first discuss the work of M. Butler and D. Yadav [19], since it is closest to ours. Their development uses Event-B, which like ASMs uses an operational style of specification (in contrast to the original Z which is relational). Event-B is based on the idea of structuring a development into many small steps to achieve a high degree of automation. So [19] used 9 refinements to develop a Mondex-like protocol. One key idea in their work is to view Mondex protocol runs as instances of transactions, viewing the state of all the purses as a kind of database (our work in [25,26] also picks up on this idea). Because of this, their first refinements do not introduce messages (like ours), but define transactions and status information. This leads to an elegant development with small steps and a high degree of automation, but the price to pay is that intermediate levels use concepts (like a purse being in several transactions simultaneously), which are not present in the Mondex protocol.

Our goal in this paper was different: we wanted to cleanly separate the concepts present in the original Mondex protocol, and made no attempt to generalize. We also did not attempt to automate proofs further than in our earlier work. In fact, the effort for proving the 4 refinements of this paper was slightly higher than for the single refinement [12], due to revisions of intermediate levels.

Despite the different aims of these papers, there is one key idea we also used: abstract tid's to identify protocol runs (or transactions), since it abstracts nicely from the use of sequence numbers to identify protocol runs. Use of tid's leads to similarities between the Event-B machines and our ASMs. Although there are differences (no startTrans in our development; at this stage, our protocol has three messages), the biggest similarities are between the Event-B machines derived after around 6 refinements, and the one that our first refinement derives. This agrees with our experience, that the first refinement is still the most complex to verify. Also, their refinements 6 and 7 introduce sequence numbers, which we define in the third refinement.

The other work on Mondex that uses a structured development is the one of C. George and A.E. Haxthausen [20]. The work is based on the RAISE specification language and derives the Mondex protocol using two refinements, starting from a specification that can be viewed as a first refinement of our abstract specification. The key

idea of this specification is: to transfer money from one purse to another there has to be a sending step (called transferLeft which either puts money "in transit" or moves it to lost), a successful receiving step (called transferRight, which moves money from in transit to balance(to)), and a step which moves money from in transit to lost (called Abort). The two steps of the refinement then show that all steps of the Mondex protocol implement one of these steps (e.g. REQ, that sends the Val message, implements transferLeft). This development has the advantage that the propagation of the security goals to the refined machines becomes easy. However the resulting refinement steps are rather different from the ones we give here.

8 Conclusion

In this paper we have analyzed the core concepts of the Mondex protocol, and we have shown that it is possible to place each concept into one concept-specific refinement. We have also given a slight improvement of the technique of *purse-local* invariants, explained in [17], by using *protocol-local* simulation relations, as suggested by our recent results on a framework for interleaved protocols [25]. This has led to the verification of each protocol run as one big commuting diagram, which moves much of the complexity of the first refinement into generic theory. The generic framework has now been verified in KIV [26], and holds promise for further extension and application.

Acknowledgement. We would like to thank Bogdan Tofan, who has done many of the KIV proofs that ensure correctness of this work.

References

1. MasterCard International Inc.: Mondex, http://www.mondex.com
2. UK ITSEC Certification Body: UK ITSEC Scheme Certification Report No. P129 MONDEX Purse. Technical report (1999), http://www.cesg.gov.uk/site/iacs/itsec/media/certreps/CRP129.pdf
3. CCIB: Common Criteria for Information Technology Security Evaluation, Version 3.1 (ISO 15408) (November 2007), http://csrc.nist.gov/cc
4. Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Lab (2000), http://www-users.cs.york.ac.uk/susan/bib/ss/z/monog.htm
5. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. PHI (1992)
6. Woodcock, J.: First Steps in the Verified Software Grand Challenge. IEEE Computer 39(10), 57–64 (2006)
7. Jones, C., Woodcock, J. (eds.): Formal Aspects of Computing, vol. 20 (1). Springer, Heidelberg (January 2008)
8. 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)
9. Cooper, D., Stepney, S., Woodcock, J.: Derivation of Z Refinement Proof Rules. Technical Report YCS-2002-347, University of York (2002), http://www-users.cs.york.ac.uk/susan/bib/ss/z/zrules.htm

10. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford Univ. Press, Oxford (1995)
11. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
12. Schellhorn, G., Grandy, H., Haneberg, D., Moebius, N., Reif, W.: A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In: Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis. LNCS, Springer, Heidelberg (2008); (older version available as Techn. Report 2006-27 at [24])
13. Börger, E., Rosenzweig, D.: The WAM—Definition and Compiler Correctness. In: Logic Programming: Formal Methods and Practical Applications. Studies in CS and AI, vol. 11, pp. 20–90. North-Holland, Amsterdam (1995)
14. Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. J.UCS 7(11), 952–979 (2001)
15. Börger, E.: The ASM Refinement Method. FAC 15 (1-2), 237–257 (2003)
16. Schellhorn, G.: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. TCS 336, 403–435 (2005)
17. Schellhorn, G.: ASM Refinement Preserving Invariants. In: Proceedings of the ASM workshop 2007, Grimstad, Norway (2008) (to appear in J.UCS)
18. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1,1) Forward Refinements. Fund. Inf. 77, 29–69 (2007)
19. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. FAC 20(1) (January 2008)
20. Haxthausen, A., George, C.: Specification, Proof, and Model Checking of the Mondex Electronic Purse using RAISE. FAC 20(1) (January 2008)
21. 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 20(1) (January 2008)
22. Moebius, N., Haneberg, D., Schellhorn, G., Reif, W.: A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In: International Conference on Software Engineering Advances (ICSEA). IEEE Press, Los Alamitos (2007), http://ieeexplore.ieee.org
23. Grandy, H., Bischof, M., Schellhorn, G., Reif, W., Stenzel, K.: Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014. Springer, Heidelberg (2008)
24. Mondex KIV: Web presentation of the Mondex case study in KIV, http://www.informatik.uni-augsburg.de/swt/projects/mondex.html
25. Banach, R., Schellhorn, G.: On the Refinement of Atomic Actions. In: Proceedings of REFINE 2007. ENTCS, vol. 201, pp. 3–30 (2007)
26. Banach, R., Schellhorn, G.: Atomic Actions, and their Refinements to Isolated Protocols. In: FAC (2008)
27. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)

A Scenario-Based Validation Language for ASMs

A. Carioni[2], A. Gargantini[1], E. Riccobene[2], and P. Scandurra[2]

[1] Dip. di Ing. Informatica e Metodi Matematici, Università di Bergamo, Italy
angelo.gargantini@unibg.it
[2] Dip. di Tecnologie dell'Informazione, Università di Milano, Italy
{carioni,riccobene,scandurra}@dti.unimi.it

Abstract. This paper presents the AValLa language, a domain-specific modelling language for scenario-based validation of ASM models, and its supporting tool, the AsmetaVvalidator. They have been developed according to the model-driven development principles as part of the asmeta(ASM mETAmodelling) toolset, a set of tools around ASMs. As a proof-of-concepts, the paper reports the results of the scenario-based validation for the well-known LIFT control case study.

1 Introduction

The success of developing complex systems depends on the use of a pertinent method for identifying the requirements on the target system and to make sure that the produced system will actually meet these requirements. Validation is intended as the process of investigating a model (intended as formal specification) with respect to its user perceptions, in order to ensure that the specification really reflects the user needs and statements about the application, and to detect faults in the specification as early as possible with limited effort. Validation should precede the application of more expensive and accurate methods, like formal requirements analysis and verification of properties, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Techniques for validation include scenarios generation, development of prototypes, animation, simulation, and also testing [28].

In [21], we defined the AsmetaL language as concrete syntax to write Abstract State Machine (ASM) models and the AsmetaS simulator to execute AsmetaL programs. In order to validate AsmetaL specifications, we here investigate the *scenario-based* approach for system validation. In this context, scenarios describe the behavior of a system from a *global* perspective by looking at the observable interactions between the system and its environment in specific situations. Scenarios are useful to ensure correct capture of informal requirements and to explore system functionalities and alternative design solutions. To make this approach effective by allowing the designer to interact with the specification, we define a language, called AValLa (ASM Validation Language), which provides suitable commands to express, at ASM model level, the interaction between a system and its environment (in the sense of UML use-cases) and the interaction

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 71–84, 2008.

between a system and an external observer who likes to play with the system model and check the system state.

AVALLA has been developed according to the model-driven language engineering principles which require the abstract syntax of a language be defined in terms of an (object-oriented) model, called *metamodel*, characterizing syntax elements and their relationships. A concrete notation can be then (automatically) derived from the abstract syntax. The language semantics is given in terms of ASMs, here used as formal semantic framework to express the operational semantics of metamodel-based languages.

AVALLA is supported by the ASMETAV (ASM Validator) tool to execute AVALLA scenarios. Both have been developed within the ASMETA(ASM mETAmodelling) tool-set [17,19,4] by exploiting the metamodelling approach.

In this paper, we first motivate in Sect. 2 our work on scenario-based system validation in relation to other similar approaches. In Sect. 3, we present our basic idea on how targeting validation in the ASM context. In Sect. 4 we present the AVALLA language to build scenarios for ASM models, and we describe how AVALLA has been defined following the model-driven engineering process. In Sect. 5, we provide the semantics of the AVALLA constructs exploiting the ASM-based semantic framework for metamodel-based languages. Our scenario-based validator ASMETAVis presented in Sect. 6, while Sect. 7 presents a case study. Conclusions are given in Sect. 8.

2 Motivations and Related Work

The *scenarios* technique has been applied in different research areas and a variety of definitions, ways of use and ways of interaction with the user are given. In particular, scenarios have been used in the area of Software Engineering [33,2,32], Business-process reengineering [3], User Interface Design [9], Documentation and demonstration of software and many more. In addition, the term "script" used in Artificial Intelligence [35] and in Object-behavior Analysis [36], is very similar to the various definitions of scenarios.

Authors in [8] classify scenarios according to their use in systems development ranging from requirements analysis, user-designer communication, examples to motivate design rationale, envisionment (imagined use of a future design), software design, through to implementation, training and documentation.

The telecommunication system development is one of the main field where scenarios have been successfully applied [1]. Message Sequence Charts (MSCs) [31] is one of the most used (graphical) notation by telecommunications companies and standard bodies. MSCs can be adapted to describe embedded systems and software, although, for software, UML notations are more used. The Life Sequence Charts (LSCs) [11] extend the MSCs by providing the "clear and usable syntax and a formal semantics" MSCs lack of.

In the object-oriented community, scenarios are intended as instances of a *use case* [39] defining a goal-oriented set of interactions between external actors (i.e. parties outside the system that interact with the system) and the system under

consideration. The system is treated as a *black box*, and the interactions with it, including system responses, are perceived as outside the system. A complete set of use cases specifies all different ways to use the system, and therefore defines all required behavior, bounding the scope of the system. For complex systems, the complete set of use cases might be unfeasible, and in this case it is useful to proceed in an incremental way.

The idea of using scenarios as a means for validating a specification has been extensively adopted in the past, but its application has been mostly of informal nature. [37] provides a mini tutorial explaining the concepts and process of scenario-based requirements engineering. The relationships between scenarios, specifications and prototypes are explored, and the SCRAM method (Scenario-based Requirements Analysis Method), where scenarios are used with early prototypes to elicit requirements in reaction to a preliminary design, is presented. In [26], a systematic way to analyze and validate requirements is proposed and applied to a simple PBX system. This formal-approach to scenario analysis views the user as the starting point to form scenarios and uses prototyping in order to validate the scenarios and refine the specifications. In [27], a case study is presented to show how functional requirements can be successfully decomposed and analyzed using scenarios. In [30], authors show the CineVali approach in which scenarios are formal and automatically generated by the user and by the analyst in accordance with their purposes.

The main obstacles to an effective use of scenarios for formal validation are mainly due to the non executable nature of formal models, or in the case of executable specifications, due to the lack of simulation engines and suitable tools allowing the designer to interact with the (complete or only sketched) specification in order to observe the system behavior and/or check the system states.

A method for constructing a formal description of the system from scenarios expressing the stakeholders' requirements, is presented in [25]. The authors use the Albert II formal language and scenarios are represented by MSCs. A requirements validation tool that stakeholders can use to explore different possible behaviors of the system to develop, is presented. These behaviors are automatically checked against the formal requirements specification.

In the context of ASMs, the authors in [22,5] show how SpecExplorer and its language Spec#, can be applied for scenario-oriented modelling. They describe how Spec# models can be instrumented for validation purposes by a set of instructions, which allow SpecExplorer to execute scenarios. They also describe scenarios in an algorithmic way with the ultimate goal to have a tailored notation, like MSCs, as front-end for scenarios description. Grieskamp et al. also provide an engine within the SpecExplorer tool for checking conformance of implementation against models.

Our approach is targeted to build scenarios for ASM ground models written in AsmetaL. We like to keep the algorithmic vision of building scenarios as in Spec#/SpecExplorer, since we consider this view closer to the view of programming and able to show the temporal sequence of steps between the system and its external environment. We keep the view of scenarios as paths through the

use cases as inherited from the object-oriented community. Therefore, in our view a scenario will express interaction sequences of external actor actions and reactions of the machine to be analyzed.

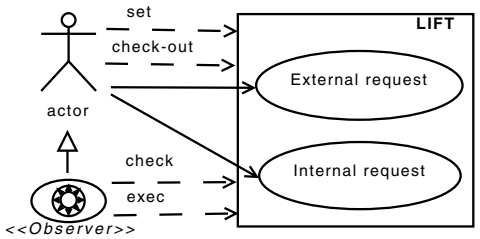
From a practical point of view, we believe that a validation activity in which the designer has a black box view of the system might be complemented by a testing activity requiring an internal view of the system. As in [10], we argue that a scenario notation should be also able to describe internal details of the system. MSCs and LSCs are very useful to describe lengthy black-box interactions between system components in a graphical way, while we want scenarios to be also able of describing, by means of a textual notation, possibly white-box interactions in a component independent way. To this regard our approach is more similar to the classical unit testing. Note that several scenario notations are derived form testing notations, for example the Use Case Maps, for which and ASM based semantics exists [24], and Use Case Trees are strongly related to the TTCN testing notation.

Therefore, in our scenario-based approach, we support two kinds of external actors: the *user*, who has a black box view of the system, and the *observer* having, instead, a gray box view. By allowing different actions to the two actors, we are able to build scenarios useful for classical validation (those including user actions and machine reactions), and scenarios useful for testing activity (those including also observer actions) requiring the inspection of the internal configurations of the machine. Therefore, our scenario-based validation approach goes behind the UML use-cases it was inspired from, and has the twofold goal of model validation and model testing.

3 Scenario-Based Validation of ASM Models

In our approach of scenario-based validation of ASM models, we start from the idea of UML use-cases and their descriptions in terms of scenarios. A scenario is a description of external actor actions and reactions of the system. The formalization (complete or incomplete) of the system behavior is given in terms of an ASM specification. We extend the concept of actor (here called *user actor*) in UML use-cases with the concept of *observer actor* (see Fig. 1(a)). A user actor is able to interact with the system by *setting* the values of the external environment, so asking for a particular service, waits for a *step* of the machine as reaction to his/her request, and can check the values only of system outputs. A user actor has, therefore, a black box view of the system. An observer actor has the further capabilities of inspecting the internal state of the system (i.e. values of machine functions and locations), to require the *execution* of particular system (sub-)services of the machine, and to check the validity of possible invariants of a certain scenario. Therefore, an observer actor has a *gray box* view of the system under development.

Use-cases are described by a set of scenarios and each scenario represents a single path through the use case. Usually, in the UML, scenarios are depicted using sequence diagrams describing the actor(s)-system interaction, but also the



(A)

```
check floor(lift1) = 0;
// external request for UP
set existsCallFromTo(0, UP) := true;
// internal request to floor 4
set hasToDeliverAt(lift1, 4) := true;
// start moving
step
check ctlState(lift1) = MOVING;
check dir(lift1) = UP;
check existsCallFromTo(0, UP) = false;
```

(B)

Fig. 1. Use cases (A) and a scenario (B) for the `Lift` model

system components interaction to provide a certain service. We prefer to describe scenarios in an algorithmic way as interaction sequences consisting of *actions*, where each action in turn is an activity of a user actor who `set`s the environment (i.e. the values of monitored/shared functions) and `check`s for the machine outputs (i.e. the values of out functions), possibly combined with an activity of the observer actor who has the extra ability to `check` the machine (also internal) state and ask for the `exec`ution of given transition rules, and an activity of the machine which makes one `step` as reaction of the actor actions.

Fig. 1(b) shows a script relative to a scenario of the LIFT case study taken from [6] and encoded in AsmetaL. The scenario shows the interaction between a lift lying at ground floor and a user stating at the same floor and asking for the lift to go up. Once getting into, he/she asks for reaching floor 4. The observer initially checks (first **check**) that the lift is at the ground floor before the interaction takes place, and upon the machine makes a step, he/she checks that the lift is moving in the up direction and that the (external) request has been removed. Note that `existsCallFromTo(floor,dir)` specifies an external request function (reflecting the user action of pressing the `up` or `down` button outside the lift at a certain `floor`), while `hasToDeliverAt(lift,floor)` formalizes an internal request function (reflecting the user action of pressing a button inside the lift).

4 The AValLa Language

The AValLa language has been defined as a domain-specific language (DSL) in the context of scenario-based validation of ASM models written in AsmetaL. As required for model-driven language definition, the abstract syntax of AValLa is defined in terms of an (object-oriented) model which is usually referred in the MDE context [29] to as *Domain Definition MetaModel* (DDMM). The DDMM represents concepts and their relations of the underlying domain and allows the separation of the abstract syntax and semantics of the language constructs from their different and alternative concrete notations (textual, visual, or mixed) for various goals. A concrete syntax is usually defined by a transformation that maps the DDMM onto a "display surface" metamodel (like XML, EBNF, etc.) [29].

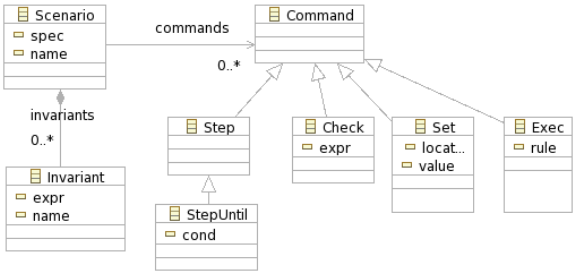


Fig. 2. AVALLA metamodel (MM)

Domain Definition Metamodel (abstract syntax). The MM (Meta Model) of the AVALLA is specified in EMF/Ecore [12]. Fig. 2 shows the metamodel using a UML class diagram. This metamodel captures concepts and their relations of the ASMs-based scenario modelling domain mentioned in Sect. 3.

An instance of the class `Scenario` represents a scenario of a provided ASM specification. Basically, a scenario has an attribute `name`, an attribute `spec` denoting the ASM specification to validate, and a list of target commands of type *`Command`*. Additionally, a scenario may contain the specification of some critical properties, here referred to as *scenario invariants*, that should always hold (and therefore verified) for the particular scenario – to not be confused with general *axioms* one specifies for an ASM spec as invariants over functions or domains constraining the ASM states. The composite associations between the `Scenario` class (the *whole*) and its component classes (the *parts*) `Invariant` and *`Command`* assures that each part is included in at most one `Scenario` instance.

The abstract class *`Command`* and its concrete sub-classes provide a classification of scenario commands. The `Set` command updates monitored or shared function values that are supplied by the user actor as input signals to the system. Commands `Step` and `StepUntil` represent the reaction of the system, which can execute one single ASM step and one ASM step iteratively until a specified condition becomes true. The `Check` class represents commands supplied by the user actor to inspect external property values and/or by the observer actor to further inspect internal property values in the current state of the underlying ASM. Finally, an `Exec` command executes an ASM transition rule when required by the observer actor.

Concrete Syntax. A concrete syntax for AVALLA has been implemented as textual notation according to the model-to-grammar mapping described in [16] and already used for deriving the AsmetaL notation [4] from the ASM Metamodel (*AsmM*) representing the abstract syntax of the ASM language as given in [4,17]. A grammar (written in JavaCC) and a parser [4] are derived from the AVALLA MM to automatically parse textual scenario scripts into scenario models. Other tools, as TEF (Textual Editing Framework) [38] allowing creation of textual editors for EMF-based languages, could be exploited for the same goal. Table 1 reports the AVALLA concrete syntax in a EBNF form, in which terminal

Table 1. The AVALLA textual notation

Abstract syntax	Concrete syntax
Scenario	**scenario** name **load** spec_name Invariant* Command*
spec_name is the spec to load; invariants and commands are the script content	
Invariant	**invariant** name ':' expr ';'
expr is a boolean term made of function and domain symbols of the underlying ASM	
Command	(Set | Exec | Step | StepUntil | Check)
Set	**set** loc := value ';'
loc is a location term for a monitored function, and value is a term denoting a possible value for the underlying location	
Exec	**exec** rule ';'
rule is an ASM rule (e.g. a choose/forall rule, a conditional if, a macro call rule, ect.)	
Step	**step** ';'
StepUntil	**step until** cond ';'
cond is a boolean-valued term made of function and domain symbols of the ASM	
Check	**check** expr ';'
expr is a boolean-valued term made of function and domain symbols of the ASM	

symbols are in bold and elements in the first column represent non terminals. Examples of scenario scripts are provided in Sect. 7 for the `Lift` case study.

5 The AVALLA Semantics

Currently, metamodelling environments (like Eclipse/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.) allow to cope with most syntactic and transformation definition issues in the specification of a DSL, but they lack of any standard and rigorous support to provide the dynamic semantics of metamodels and metamodel-based languages, which is usually given in natural language (the most well-known example is the UML [39]). Below, we briefly present the approach we adopted to define the AVALLA semantics.

An ASM-based semantic framework for DDMMs. A language has a well-defined semantics if a semantic domain S is identified and a semantic mapping M_S from the language syntax to S is provided [23]. As semantic domain S, we assume the semantic domain S_{AsmM} of the ASM language, namely the first-order logic extended with a logic for function updates and for transition rule constructors formally defined in [6]. Therefore, the semantic mapping $M_S : DDMM \rightarrow S_{AsmM}$ which associates a well-formed terminal model[1] m conforming to $DDMM$ with its semantic model $M_S(m)$, can be defined as

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

[1] According to the definition in [29], a *terminal model* is a model written in the language L and *conforming* to the language metamodel.

where $M_{S_{AsmM}}$ is the semantic mapping (of the ASM language) that associates a theory conforming to the S_{AsmM} logic with a model conforming to $AsmM$ (representing the abstract syntax of the ASM language), and the function

$$M : DDMM \longrightarrow AsmM$$

associating an ASM to a terminal model m. The M function *hooks* the semantics of a metamodel to the S_{AsmM} domain and, therefore, the problem of giving the language semantics is reduced to define the function M. Exploiting this approach, the semantics of a metamodel-based language is expressed in terms of ASM transition rules.

Language Semantics. According to the approach explained above, to endow the AVALLA language with a semantics we have to define a function M: $MM \longrightarrow AsmM$ associating an ASM (i.e. a model conforming to the AsmM metamodel) with a scenario model m conforming to the AVALLA MM. This ASM machine is automatically induced from the elements of the source scenario model m and their typing elements in the AVALLA MM, and from the original ASM to validate (which is linked to a scenario by its attribute `spec`). The resulting ASM is obtained from the original ASM in the following way.

A scenario (instance of the `Scenario` class) is mapped into the original ASM to validate (instance of the `Asm` class in AsmM), except that: monitored and shared functions are turned into controlled functions; a new 0-ary controlled function `currentRule` of `Rule` type is added to the signature to denote the current rule of the original ASM being executed; for notifying check-command's property violations, a boolean-valued 0-ary function `all_checks_OK` is added to the signature together with an axiom stating that flag `all_checks_OK` is always true; the original initial state is extended to set `currentRule` to an initial rule `r_step_0` and for setting `all_checks_OK` to true; finally, the main rule consists only into invoking the value of the `currentRule`.

Invariants and commands of the particular scenario are then taken into consideration in order to further modify the structure of the ASM (see Table 2). Scenario invariants are mapped into axioms of the final ASM. The commands list is then partitioned into groups: each group is a block of consecutive commands terminated with either a step-command, a *step-group*, or a stepUntil-command, *stepUntil-group*. For each group one or two rules are added to the final ASM according to the following directives. The i-th step-group $[C_1 \ldots C_n$ **step**$]$ is mapped into a macro rule declaration of form:

$$\begin{aligned} r\_step\_i = \textbf{seq} \\ & R_1 \ldots R_n \\ & old\_main[] \\ & currentRule :=\ll r\_step\_i + 1 \gg \\ \textbf{endseq} \end{aligned} \tag{1}$$

where R_i are rules generated from the corresponding commands C_i, old_main is the invocation of the main rule of the original ASM, and the last rule is the

Table 2. Mapping from AVALLA to AsmM

AValLa	AsmM
A `Invariant` instance	An `Axiom` instance
A `Set` instance *l:=v*	An `UpdateRule` instance *l:=v*
A `Check` instance with expression *expr*	A `ConditionalRule` with guard *expr* and then-body *all_checks_OK:=false*
A `Exec` instance for a rule *R*	A `Rule` instance
A step-group $C_1 \dots C_n$ **step**	A `MacroDeclaration` instance *r_step_i* as in (1)
A stepUntil-group $C_1 \dots C_n$ **step until** *cond*	Two `MacroDeclaration` instances *r_step_i* and *r_step_i_until* as in (2)

update for the `currentRule` variable. The *i*-th stepUntil-group [$C_1 \dots C_n$ **step until** *cond*] leads to two rules of form:

```
r_step_i=                    r_step_i_until =                              (2)
  seq                          if cond then currentRule :=≪ r_step_(i + 1) ≫
    R_1 ... R_n                else par
    r_step_i_until[]                 old_main[]
  endseq                             currentRule :=≪ r_step_i_until ≫
                                   endpar
                               endif
```

where symbols R_i and *old_main* take the same meaning as above. Note that the starting rule `r_step_0` is produced by the first command group. The ASM rules R_i for the remaining commands are produced as follows. A set-command is directly mapped into an update rule. An exec-command is a request for executing a rule *R* of the original ASM, and therefore it is mapped into an instance of `Rule` class of *AsmM*. A check-command is mapped into a conditional rule having as guard the expression to check and as then-body an update for setting the `all_checks_OK` flag to false (and therefore causing an axiom violation).

6 The AsmetaV Validator

The AsmetaV validator is a simple Java application based on a transformation engine which automatically maps any scenario model conforming to the AValLa metamodel into an *AsmM* instance as described in Sect. 5, and on the AsmetaS simulator [18]. AsmetaV reads a scenario written by the user (see Fig. 3) in the AValLa language, builds the scenario as instance of the AValLa metamodel by means of the AValLa parser, and transforms the scenario and the AsmetaL specification which the scenario refers to, to an executable AsmM model.

Then, AsmetaV invokes the interpreter to simulate the scenario. During simulation the user can pause the simulation and observe, through a watching window, the current state and value of the update set produced at every step. During

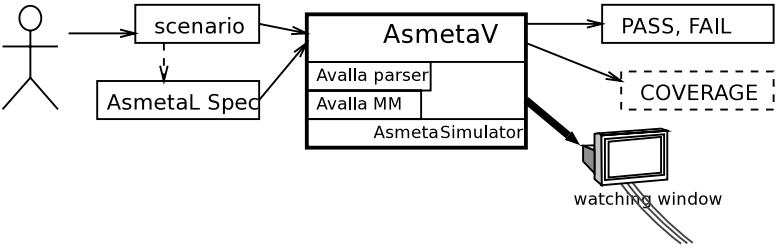


Fig. 3. AsmetaV validator

simulation, AsmetaV captures any check violation and if none occurs it finishes with a "PASS" verdict.

Besides a "PASS"/"FAIL" verdict, AsmetaV collects also some information about the *coverage* of the original model, obtained by running the scenario. Currently, AsmetaV keeps track of all the rules that have been called and evaluated and it prints the list of them at the end. This is useful to check if the scenario has exercised all transition rules of the original model. We plan to further refine this feature in order to monitor also which rules have actually contributed to the update set, which conditions in conditional rules have been tested true and/or false and eventually to apply the definition of coverage criteria as in [14,13].

7 The LIFT Case Study

To illustrate the use of the AsmetaVtool to validate an ASM specification by means of some input AValLa scenarios, we use the `Lift` example (see [6], Sect. 2.3) concerning the logic of moving n lifts between m floors.

For the sake of simplicity, we restrict to the case of one lift (`lift1`) for $m = 5$ floors. Moreover, we assume that the level of abstraction at which the `Lift` ground model is defined includes also the refinement step for the *request manipulations* (see [6], Sect. 2.3, pag. 57). In the intermediate model that we consider, the monitored function $hasToDeliverAt(L, floor)$ formalizes an internal request (reflecting requirement 1 when inside the lift a button is pressed), while an external request is modelled by the function $existsCallFromTo(floor, dir)$ (reflecting requirement 2 when on a floor outside the lift the up or down button is pressed).

Table 3. Some validation scenarios for the `Lift` control

Scenario	Description	Req.	# Commands	Inv.	Coverage
s0	No requests at all	3	19	–	6/8
s1	An external request	1	24	–	7/8
s2	An external request plus an internal one	2.(a)	22	–	8/8
s3	All external buttons pushed	2.(a)	4	1	8/8

These two functions are shared between the lift user (who sets them, being part of the environment) and the lift control (which has to reset them in the CANCEL-REQUEST macro to satisfy requirements 1 and 2). We do not consider, instead, to handle exceptional cases when the given machine either has no well-defined behavior or should be prevented from executing; we suppose therefore that the machine describes the functionality of faultless behavior.

Table 3 summarizes some of the scenarios used to validate the ASM specification of the `Lift` control. A more detailed description of these scenarios follows.

s0 *Description*: The lift is halted at ground floor (# 0) with no requests at all.
Requirements coverage: 3. The lift should remain in its final position.

```
1 // .... setting initial state
2 check floor(lift1) = 0;
3 check ctlState(lift1) = HALTING;
4 check dir(lift1) = UP;
6 step
7 check floor(lift1) = 0;
8 check ctlState(lift1) = HALTING;
9 check dir(lift1) = UP;
```

 # commands: set(14), check (4), step (1)
Rule coverage: 6/8 *Verdict*: PASS

s1 *Description*: The lift is halted at ground floor. A user gets into the lift and asks for reaching floor 4.
Requirements coverage: 2. The lift should move in the up direction and the external request at ground floor should be cancelled (as being satisfied). See Fig. 1(b) in Sect. 3.
 # commands: set(16), check (5), step (3)
Rule coverage: 7/8 *Verdict*: FAIL (see remark below for explanation)

s2 *Description*: The lift is halted at ground floor. A user calls the lift at floor 4 and once getting into the lift he/she asks for reaching floor 2.
Requirements coverage: 2.(a) The lift satisfies the user request by reaching floor 4 and then reaching floor 2. Once satisfied, the requests must be removed:

```
1  // .... setting initial state
2  // An external request to floor 4
3  set existsCallFromTo(4, DOWN) := true;
4  // The lift goes to floor 4
5  step until ctlState(lift1) = HALTING
6                 and floor(lift1) = 4;
7  // A request to floor 2
8  set hasToDeliverAt(lift1, 2) := true;
9  step
10 // must go down to floor 2, down dir
11 check dir(lift1) = DOWN;
12 // the request at floor 4 is cancelled
13 check not existsCallFromTo(4, DOWN);
14 // goes to floor 2
15 step until ctlState(lift1) = HALTING
16                 and floor(lift1) = 2;
17 // request to floor 2 is cancelled
18 check not hasToDeliverAt(lift1, 2);
```

 # commands: set(16), check (3), step (1), step-until (2)
Rule coverage: 8/8 *Verdict*: PASS

s3 *Description*: The lift is halted at ground floor. All external buttons (UP and DOWN) have been pushed.
Requirements coverage: 2.(a) The lift should move sequentially in the up direction from the ground floor 0 to the top floor 4. After reaching floor 4,

all UP requests should be canceled, while the DOWN ones should be still pending.
Scenario invariants: The lift should not change direction while going up: dir(lift1) != DOWN;
 # commands: check (2), step-until(1), exec (1)
Rule coverage: 8/8 *Verdict*: FAIL (see remark below for explanation)

```
scenario s3
load lift.asm
invariant neverDown: dir(lift1) != DOWN;
exec //set floor requests (all external buttons UP and DOWN have been pushed)
    forall $i in {0..4} do
    par
        hasToDeliverAt(lift1, $i) := false
    if $i != top then existsCallFromTo($i, UP) := true endif
    if $i != ground then existsCallFromTo($i, DOWN) := true endif
    endpar;
//the lift goes up to floor 4, then goes down to complete existsCallFromTo(0, DOWN)
step until ctlState(lift1) = HALTING and floor(lift1) = 4;
check (forall $i in {0..4} with existsCallFromTo($i, DOWN) = true);
check (forall $i in {0..4} with existsCallFromTo($i, UP) = false);
```

Remark. Scenarios s1 and s3 fail since the `Lift` specification fails to cancel an external request when it occurs at a given floor where the lift is halted, and the lift has already the same requested direction. This fault can be corrected either by constraining external requests, or by cancelling this kind of external request when the lift departs. We preferred to include a CancelRequest rule invocation within the Depart rule (see [6], Sect. 2.3), rather than to add further constraints.

8 Conclusions

This work is part of our ongoing effort in developing a set of tool around ASMs for model validation and verification. In this paper, we proposed a scenario-based approach for ASM model validation.

We have been testing our validation methodology on case studies from the embedded systems domain [20,7]. The ASMs are used as formal support to deliver formal analysis techniques for visual models developed with the UML profile for SystemC [34] – an UML extension for high-level modelling of embedded systems on chip.

In the future, we plan to integrate AsmetaV with the ATGT tool [15] in order to be able to automatically generate some scenarios by using ATGT and ask for a certain type of coverage (rule coverage, fault detection, etc.).

References

1. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommunication Systems 24(1), 61–94 (2003)
2. Anderson, J.S., Durney, B.: Using scenarios in deficiency-driven requirements engineering. In: Proceedings of the International Symposium on Requirements Engineering, pp. 134–141. IEEE, Los Alamitos (1993)
3. Anton, A.I., McCracken, W.M., Potts, C.: Goal decomposition and scenario analysis in business process reengineering. LNCS, vol. 811, pp. 94–104. Springer, Heidelberg (1994)
4. The Abstract State Machine Metamodel website (2006), `http://asmeta.sf.net/`
5. Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N., Veanes, M.: Validating use-cases with the asmL test tool. In: 3rd International Conference on Quality Software (QSIC 2003), Dallas, TX, USA, November 6-7, 2003, pp. 238–246. IEEE Computer Society, Los Alamitos (2003)
6. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
7. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: Scenario-based Validation of Embedded Systems. In: FDL 2008: Proceedings of Forum on Specification and Design Languages (2008)
8. Carroll, J.M.: Five reasons for scenario-based design. Interacting with Computers 13(1), 43–60 (2000)
9. Carroll, J.M., Rosson, M.B.: Getting around the task-artifact cycle: How to make claims and design by scenario. ACM Transactions on Information Systems 10(2), 181–212 (1992)
10. Chandrasekaran, P.: How use case modeling policies have affected the success of various projects (or how to improve use case modeling). In: Addendum to the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Lanuages, and Applications, pp. 6–9 (1997)
11. Damm, W., Harel, D.: LCSs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)
12. Eclipse Modeling Framework (2008), `http://www.eclipse.org/emf/`
13. Gargantini, A.: Using model checking to generate fault detecting tests. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 189–206. Springer, Heidelberg (2007)
14. Gargantini, A., Riccobene, E.: Asm-based testing: Coverage criteria and automatic test sequence. J. of Universal Computer Science 7(11), 1050–1067 (2001)
15. 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)
16. 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)
17. 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)
18. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based simulator for ASMs. In: Prinz, A. (ed.) Proceedings of the 14th International ASM Workshop (2007)

19. Gargantini, A., Riccobene, E., Scandurra, P.: Ten reasons to metamodel ASMs. In: Rigorous Methods for Software Construction and Analysis - Papers Dedicated to Egon Börger on the Occasion of His 60th Birthday. LNCS, vol. 5115. Springer, Heidelberg (2007)
20. Gargantini, A., Riccobene, E., Scandurra, P.: A Model-driven Validation & Verification Environment for Embedded Systems. In: Proc. of the IEEE third Symposium on Industrial Embedded Systems (SIES 2008). IEEE, Los Alamitos (2008)
21. Gargantini, A., Riccobene, E., Scandurra, P.: A language and a simulation engine for abstract state machines based on metamodelling. In: JUCS (accepted, 2008)
22. Grieskamp, W., Tillmann, N., Veanes, M.: Instrumenting scenarios in a model-driven development environment. Information & Software Technology 46(15), 1027–1036 (2004)
23. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? IEEE Computer 37(10), 64–72 (2004)
24. Hassine, J., Rilling, J., Dssouli, R.: An ASM operational semantics for use case maps. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), Paris, France, August 29 - September 2, 2005, pp. 467–468. IEEE Computer Society, Los Alamitos (2005)
25. Heymans, P., Dubois, E.: Scenario-based techniques for supporting the elaboration and the validation of formal requirements. Requir. Eng. 3(3/4), 202–218 (1998)
26. Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., Chen, C.: Formal approach to scenario analysis. IEEE Software 11(2), 33–41 (1994)
27. Kaindl, H., Kramer, S., Kacsich, R.: A case study of decomposing functional requirements using scenarios. In: 3rd International Conference on Requirements Engineering (ICRE 1998), pp. 156–163. IEEE Computer Society, Los Alamitos (1998)
28. Kemmerer, R.: Testing formal specifications to detect design errors. IEEE Trans. Soft. Eng. 11(1), 32–43 (1985)
29. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA Companion, pp. 602–616 (2006)
30. Lalioti, V., Theodoulidis, B.: Visual scenarios for validation of requirements specification. In: SEKE 1995, The 7th Int. Conference on Software Engineering and Knowledge Engineering, pp. 114–116. Knowledge Systems Institute (1995)
31. Message sequence chart (MSC). ITU-T Recommendation Z.120, International Telecommunications Union (November 1996)
32. Nielsen, J.: Scenarios in discount usability engineering. In: Scenario-Based Design, pp. 59–83. John Wiley & Sons, Chichester (1995)
33. Potts, C., Takahashi, K., Anton, A.I.: Inquiry-based requirements analysis. IEEE Software 11(2), 21–32 (1994)
34. 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 Press, New York (2005)
35. Rich, E., Knight, K.: Artificial Intelligence. McGraw-Hill, New York (1991)
36. Rubin, K.S., Goldberg, A.: Object behavior analysis. Communications of the ACM 35(9), 48–62 (1992)
37. Sutcliffe, A.: Scenario-based requirements engineering. In: 11th IEEE Joint Int. Conference on Requirements Engineering (RE 2003), pp. 320–329 (2003)
38. Textual Editing Framework (2007), `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html`
39. OMG. The Unified Modeling Language (UML), v2.1.2 (2007), `http://www.uml.org`

Data Flow Analysis and Testing of Abstract State Machines

Alessandra Cavarra

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD UK
alessandra.cavarra@comlab.ox.ac.uk

Abstract. This paper introduces an approach to apply data flow testing techniques to Abstract State Machines specifications. Since traditional data flow coverage criteria are strictly based on the mapping between a program and its flow graph, they cannot be directly applied to ASMs. In this context we are interested in tracing the flow of data between states in ASM runs as opposed to between nodes in a program's flow graph. Therefore, we revise the classical concepts in data flow analysis and define them on two levels: the syntactic (rule) level, and the computational (run) level. We also specify a family of ad hoc data flow coverage criteria and introduce a model checking-based approach to generate automatically test cases satisfying a given set of coverage criteria from ASM models.

1 Introduction

Model-based testing is a technique for generating a suite of test cases from a model encoding the intended behaviour of the system under test. This model can reside at various levels of abstraction. Testers adopting this approach shift their attention from hand-crafting individual tests to the model of the system under test and a test generation infrastructure.

The aim of this paper is to use specifications written in the Abstract State Machine language as oracles for data flow analysis and testing. The idea behind data flow testing is to monitor the lifecycle of a piece of data, searching for inappropriate definitions, use in predicates, computations and termination. Data flow coverage criteria are based on the intuition that one should not feel confident that a variable has been updated in a correct way at some stage in the program if no test causes the execution of a computation path from the point where the variable is assigned a given value to the point where such value is subsequently used (in a predicate or computation).

This idea fits well the ASM approach where in a given state a number of functions are updated and used to compute updates, provided that certain conditions are satisfied. Nevertheless, to our knowledge, data flow coverage criteria have never been defined for ASMs.

Methods using ASM models for automatic test generation exist in literature. In particular, in [18] and [19] Gargantini et al. present a set of interesting coverage criteria together with two model checking-based tools (using, respectively,

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 85–97, 2008.

SMV [29] and SPIN [24]) to generate test suites that accomplish the desired level of coverage. This approach focuses strictly on the structure of the ASM specification, and can be considered as the equivalent of control flow testing for ASMs. However, full coverage of all the rules and conditions in an ASM will not guarantee that all the possible patterns of usage of a variable are exercised, therefore missing the opportunity to find potential errors in the way the variable is processed.

Classical data flow adequacy criteria are based on the one to one mapping between the code and its control flow graph. However, while control flow is usually explicitly defined in programming languages, it is only implicit in ASMs; therefore, the notion of flow graphs is not applicable to ASMs and, consequently, classical data flow adequacy criteria cannot be directly applied to ASMs.

In Section 2 of this paper we discuss how to address this problem by modifying traditional data flow analysis concepts taking in consideration both the structure of the machine (i.e. its rules) and its computations (i.e. the runs). Also, by varying the required combinations of definitions and uses, we define a family of test data selection and adequacy criteria based on those illustrated in [32]. In Section 3, we introduce an approach based on model checking to generate automatically a collection of test cases satisfying a given set of coverage criteria. Finally, in Section 4 we discuss our results, and related and future work.

Since this paper is intended for an expert audience, we do not provide a description of Abstract State Machines. The interested reader can find a comprehensive introduction to ASMs in [21,3].

2 Data Flow Analysis

In this section we introduce the main concepts of data flow analysis, discuss the obstacles to adapting them to the ASM paradigm, and define ad-hoc coverage criteria for ASMs.

The goal of traditional data flow analysis is to detect data flow errors (in the way data are processed and used) and anomalies. Data flow anomalies indicate the possibility of program faults. For instance, two very common anomalies found in a program are:

- *d-u anomalies*: they occur when a defined variable has not been referenced before it becomes undefined (e.g. out of scope or the program terminates). This anomaly usually indicates that the wrong variable has been defined or undefined.
- *d-d anomalies*: they indicate that the same variable is re-defined without being used, causing a hole in the scope of the first definition of the variable (this anomaly usually occurs because of misspelling).

The major difficulty in adapting classical data flow analysis to ASMs is that it is strictly based on flow graphs. Given a program P, the flow graph associated to it is given by $G = (n_s, n_f, N, E)$, where N is the set of nodes labeled by the statements in P, n_s and n_f are respectively the start and finish nodes, and E is

the set of edges representing possible flow of control between statements. Control flow graphs are built using the concept of programming primes, i.e. sequential, conditional, and loop statements.

While, in general, for any given program it is straightforward to derive its corresponding flow graph (see Figure 1), this is clearly not the case for ASMs, where all the statements in a rule are evaluated and executed simultaneously and therefore there is no sequential flow between rules statements. (Although rule $\mathbf{R}_1$ in Example 1 is syntactically equivalent to program **P** in Figure 1, we know that semantically they are very different.)

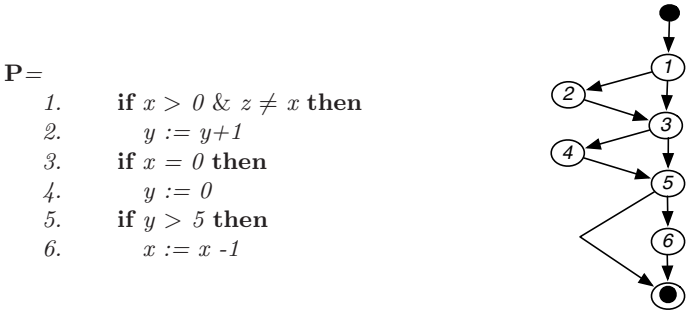


Fig. 1. A control flow graph

In the following, we provide our solution to this problem. We revise data flow concepts and provide ad-hoc definitions at two different levels: at the syntactic (rule) level and at the computational (run) level. We also provide a mapping between the concepts at different levels.

The definitions provided in this paper apply to a simple class of ASMs containing a finite set of rules and executed by a single agent (*Self*). Operators such as **seq** (for sequential updates within a rule) and **loop** are dealt with correctly by the current approach.

Our definitions are in terms of ASM functions, with variables simply being nullary functions.

2.1 Data Flow Concepts at the Rule Level

Functions can appear in different contexts in ASM rules: they can be updated, used in a guard, or used to compute a given value. In the following, we provide a number of definitions formalising the role ASM functions can play within rules.

Let $\mathcal{M}$ be an ASM.

Definition 1. *We say that a function* $f(t_1, \ldots, t_n)$ *is defined—indicated as "def"—* in a rule R_i of $\mathcal{M}$ if it appears on the LHS of an assignment in R_i (i.e. the value of $f(t_1, \ldots, t_n)$ is modified as a result of firing R_i).

We also define the following sets:

- $\text{def}_{R_i}(f(t_1,\ldots,t_n))$ is the set of all the assignments of $f(t_1,\ldots,t_n)$ in R_i.
- $\text{def}_{\mathcal{M}}(f(t_1,\ldots,t_n))$ contains all the assignments of $f(t_1,\ldots,t_n)$ across all the rules in $\mathcal{M}$, i.e. $\text{def}_{\mathcal{M}}(f(t_1,\ldots,t_n)) = \bigcup_{R_i\in\mathcal{M}} \text{def}_{R_i}(f(t_1,\ldots,t_n))$.

  An element of $\text{def}_{\mathcal{M}}(f(t_1,\ldots,t_n))$ is indicated as $d_{R_i}^{f,j}$, meaning that the function $f(t_1,\ldots,t_n)$ is in def in the statement j of R_i.

Function uses can be split into "*c-use*" and "*p-use*" according to whether the function use occurs in a computation or a predicate.

Definition 2. *We say that a function $f(t_1,\ldots,t_n)$ is in computation use—indicated as "c-use"—in a rule R_i of $\mathcal{M}$ if $f(t_1,\ldots,t_n)$ is used in a computation in R_i (i.e. it appears on the RHS of an assignment in R_i).*
We also define the following sets:

- *$c\text{-use}_{R_i}(f(t_1,\ldots,t_n))$ is the set of statements in R_i where $f(t_1,\ldots,t_n)$ is in c-use.*
- *$c\text{-use}_{\mathcal{M}}(f(t_1,\ldots,t_n)) = \bigcup_{R_i\in\mathcal{M}} c\text{-use}_{R_i}(f(t_1,\ldots,t_n))$.*

  An element of $c\text{-use}_{\mathcal{M}}(f(t_1,\ldots,t_n))$ is indicated as $c_{R_i}^{f,j}$, meaning that the function $f(t_1,\ldots,t_n)$ is in c-use in the statement j of R_i.

Definition 3. *We say that a function $f(t_1,\ldots,t_n)$ is in predicate use—indicated as "p-use"—in a rule R_i of $\mathcal{M}$ if $f(t_1,\ldots,t_n)$ is used in a predicate of R_i (i.e. $f(t_1,\ldots,t_n)$ appears in a boolean condition in R_i).*
We also define the following sets:

- *$p\text{-use}_{R_i}(f(t_1,\ldots,t_n))$ is the set of statements in R_i where $f(t_1,\ldots,t_n)$ is in p-use.*
- *$p\text{-use}_{\mathcal{M}}(f(t_1,\ldots,t_n)) = \bigcup_{R_i\in\mathcal{M}} p\text{-use}_{R_i}(f(t_1,\ldots,t_n)$.*

  An element of $p\text{-use}_{\mathcal{M}}(f(t_1,\ldots,t_n))$ is indicated as $p_{R_i}^{f,j}$, meaning that the function $f(t_1,\ldots,t_n)$ is in p-use in the statement j of R_i.

Example 1. Consider the following simple ASM $\mathcal{M}$ consisting of two rules, and three controlled variables *x,y,z* : INT

$\mathbf{R}_1$

```
1.   if x > 0 & z ≠ x then
2.      y := y+1
3.   if x = 0 then
4.      y := 0
5.   if y > 5 then
6.      x := x -1
```

$\mathbf{R}_2$

```
1.   if z = x then
2.      z:= x +y
3.      y := y -1
4    if z > y then
5.      z := z -1
```

Let us calculate the definition and use sets for the variables in $\mathcal{M}$:

$def_{R_1}(x) = \{d^6\}$ $def_{R_1}(y) = \{d^2, d^4\}$ $def_{R_1}(z) = \{\}$
$def_{R_2}(x) = \{\}$ $def_{R_2}(y) = \{d^3\}$ $def_{R_2}(z) = \{d^2, d^5\}$
$def_{\mathcal{M}}(x) = \{d^6_{R_1}\}$ $def_{\mathcal{M}}(y) = \{d^2_{R_1}, d^4_{R_1}, d^3_{R_2}\}$ $def_{\mathcal{M}}(z) = \{d^2_{R_2}, d^5_{R_2}\}$

$p\text{-}use_{R_1}(x) = \{p^1, p^3\}$ $p\text{-}use_{R_1}(y) = \{p^5\}$ $p\text{-}use_{R_1}(z) = \{p^1\}$
$p\text{-}use_{R_2}(x) = \{p^1\}$ $p\text{-}use_{R_2}(y) = \{p^4\}$ $p\text{-}use_{R_2}(z) = \{p^1, p^4\}$
$p\text{-}use_{\mathcal{M}}(x) = \{p^1_{R_1}, p^3_{R_1}, p^1_{R_2}\}$ $p\text{-}use_{\mathcal{M}}(y) = \{p^5_{R_1}, p^4_{R_2}\}$ $p\text{-}use_{\mathcal{M}}(z) = \{p^1_{R_1}, p^1_{R_2}, p^4_{R_2}\}$

$c\text{-}use_{R_1}(x) = \{c^6\}$ $c\text{-}use_{R_1}(y) = \{c^2\}$ $c\text{-}use_{R_1}(z) = \{\}$
$c\text{-}use_{R_2}(x) = \{c^2\}$ $c\text{-}use_{R_2}(y) = \{c^2, c^3\}$ $c\text{-}use_{R_2}(z) = \{c^5\}$
$c\text{-}use_{\mathcal{M}}(x) = \{c^6_{R_1}, c^2_{R_2}\}$ $c\text{-}use_{\mathcal{M}}(y) = \{c^2_{R_1}, c^2_{R_2}, c^3_{R_2}\}$ $c\text{-}use_{\mathcal{M}}(z) = \{c^5_{R_2}\}$

Let's see how Definitions 1-3 relate to the different types of ASM functions. ASM functions are distinguished into basic and derived (defined in terms of basic ones). These in turn can be static (whose value cannot be modified) or dynamic. Finally, dynamic functions can be monitored (read-only functions whose value is modified only by the "environment") or controlled (whose value can be changed by the ASM). In the case of derived functions the same definitions for *p-use* and *c-use* apply, while we consider a derived function to be in *def* in a rule if the value of one of the basic functions it is composed of is modified in that rule, i.e. it is defined in it. Obviously, static functions are in *def* only in the initial state of the machine, while the usual definitions for *p-use* and *c-use* apply. Things are more complicated for monitored functions as their value is updated outside the scope of the ASM, but used in it. As a convention here, we simply assume that they are defined in the initial state of the ASM. This will enforce the creation of tests checking the various usages of these functions in the ASM. A more accurate treatment of monitored functions would require an inter-agent data flow analysis which is matter for future work.

2.2 Data Flow Concepts at the State Level

After defining the possible roles of variables in a program, the next step in traditional data flow analysis would require tracing through the program's control flow graph to search for paths from nodes where a variable is assigned a given value, to nodes where that value is used. Since, as discussed above, in this context we cannot reason in terms of flow graphs, in this section we concentrate on ASM computations.

Abstract State Machines define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{s_i\}$, obtained from a given initial state $\{s_0\}$ by repeatedly executing transitions (rules) δ_i:

$$s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} s_2 \ldots \xrightarrow{\delta_n} s_n$$

In the following we describe how the concepts of definition and computation/predicate use at the rule level relate to ASM states. For this purpose, we need to revisit the definitions in the previous section in terms of ASM runs.

Definition 4. *Let $f(t_1, \ldots, t_n)$ be a function in $\mathcal{M}$. We say that*

- *$f(t_1, \ldots, t_n)$ is in def in a state s—indicated as "def_{state}"—if the value of $f(t_1, \ldots, t_n)$ was modified by the transition leading to s, i.e. s_i results from the application of an update in $def_{\mathcal{M}}(f(t_1, \ldots, t_n))$.*
- *$f(t_1, \ldots, t_n)$ is in p-use in a state s—indicated as "p-use_{state}"—if there exists a predicate in p-$use_{\mathcal{M}}(f(t_1, \ldots, t_n))$ that evaluates to true in s.*
- *$f(t_1, \ldots, t_n)$ is in c-use in a state s—indicated as "c-use_{state}"—if the transition leaving s causes the execution of statements (updates) in c-$use_{\mathcal{M}}(f(t_1, \ldots, t_n))$.*

In particular, we say that f is in def_{state} (respectively p-use_{state}, c-use_{state}) in a state s w.r.t. $d^{f,l}_{R_i}$ (resp. $p^{f,m}_{R_j}$, $c^{f,n}_{R_k}$) if the statement $d^{f,l}_{R_i}$ (resp. $p^{f,m}_{R_j}$, $c^{f,n}_{R_k}$) is executed in s.

Example 2. Consider again the ASM in Example 1. Let us initialise the variables as follows: $S_0 = \{x = 3, y = 6, z = 3\}$ (see Fig. 2). Since predicates $p^{y,5}_{R_1} \in$ p-$use_{\mathcal{M}}(y)$, $p^{x,1}_{R_2} \in$ p-$use_{\mathcal{M}}(x)$, and $p^{z,1}_{R_2} \in$ p-$use_{\mathcal{M}}(z)$[1] are satisfied in S_0, by definition x, y, and z are in *p-use* in S_0. Consequently, rules R_1 and R_2 are enabled to fire. Since x, y, and z are all used in computations in the transition leaving S_0 (statements $c^{x,6}_{R_1}$, $c^{x,y,2}_{R_2}$, $c^{y,3}_{R_2}$ are executed), they are also in *c-use* in S_0. After R_1 and R_2 fire, they produce the state $S_1 = \{x = 2, y = 5, z = 9\}$. Since the values of x, y, and z were modified by the transition incoming S_1, according to the definition they are in *def* in S_1.

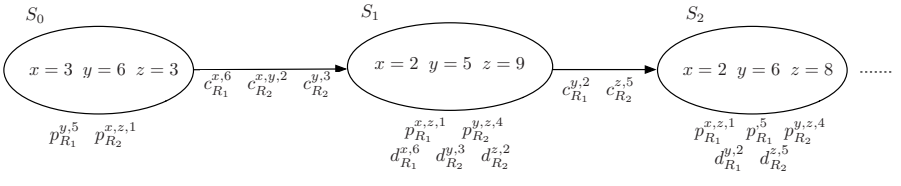


Fig. 2. A partial run of $\mathcal{M}$

We say that a sub-run is *def-clear*$(f(t_1, \ldots, t_n))$ if it contains only states where $f(t_1, \ldots, t_n)$ is not re-defined, i.e. the value of $f(t_1, \ldots, t_n)$ is not updated in any of the states of the sub-run.

Definition 5. *For each assignment $d^{f,j}_{R_i} \in def_{\mathcal{M}}(f(t_1, \ldots, t_n))$, consider a state s such that $f(t_1, \ldots, t_n)$ is in def_{state} in s w.r.t. $d^{f,j}_{R_i}$. We define two sets of states:*

- *$dpu(s, f(t_1, \ldots, t_n))$ includes states s' s.t. there is a def-clear$(f(t_1, \ldots, t_n))$ sub-run from s to s' and $f(t_1, \ldots, t_n)$ is in p-use_{state} in s', i.e. there is a*

[1] When two different functions f and g are used in a predicate (resp. computation) in the same statement i of a rule R_j, we often address both of them with the same symbol, i.e. $p^{f,g,i}_{R_j}$ ($c^{f,g,i}_{R_j}$). E.g. we refer to $p^{x,1}_{R_2}$ and $p^{z,1}_{R_2}$ with $p^{x,z,1}_{R_2}$.

computation that starts with an assignment to $f(t_1, \ldots, t_n)$*, progresses while not reassigning to* $f(t_1, \ldots, t_n)$*, and ends with a state where* $f(t_1, \ldots, t_n)$ *is used within a predicate*
- $dcu(s, f(t_1, \ldots, t_n))$ *includes states* s' *s.t. there is a def-clear*$(f(t_1, \ldots, t_n))$ *sub-run from* s *to* s' *and* $f(t_1, \ldots, t_n)$ *is in c-use*$_{state}$ *in* s'*.*

2.3 Data Flow Coverage Criteria

In this section we adapt the family of coverage criteria based on data flow information proposed by Rapps and Weyuker in [32] (and later extended in [15]). In general, such criteria require the definition of test data which cause the traversal of sub-paths from a variable definition to either some or all of the *p-uses*, *c-uses*, or their combination, or the traversal of at least one sub-path from each variable definition to every *p-use* and every *c-use* of that definition.

For each function $f(t_1, \ldots, t_n) \in \mathcal{M}$ and for each state s such that f is in def_{state} in s, we say that

- a test suite $\mathcal{T}$ satisfies the *all-defs* criterion if it includes one *def-clear*(f) run from s to some state in $dpu(s, f(t_1, \ldots, t_n))$ or in $dcu(s, f(t_1, \ldots, t_n))$
- a test suite $\mathcal{T}$ satisfies the *all-p-uses* (respectively, *all-c-uses*) criterion if it includes one *def-clear*(f) run from s to each state in $dpu(s, f(t_1, \ldots, t_n))$ (respectively, $dcu(s, f(t_1, \ldots, t_n))$)
- a test suite $\mathcal{T}$ satisfies the *all-c-uses/some-p-uses* if it includes one *def-clear*(f) run from s to each state in $dcu(s, f(t_1, \ldots, t_n))$, but if $dcu(s, f(t_1, \ldots, t_n))$ is empty, it includes at least one *def-clear*(f) run from s to some node in $dpu(s, f(t_1, \ldots, t_n))$
- a test suite $\mathcal{T}$ satisfies the *all-p-uses/some-c-uses* criterion if it includes one *def-clear*(f) run from s to each state in $dpu(s, f(t_1, \ldots, t_n))$, but if $dpu(s, f(t_1, \ldots, t_n))$ is empty, it includes at least one *def-clear*(f) run from s to some node in $dcu(s, f(t_1, \ldots, t_n))$
- a test suite $\mathcal{T}$ satisfies the *all-uses* criterion if it includes one *def-clear*(f) run from s to each state in $dpu(s, f(t_1, \ldots, t_n))$ and to each state in $dcu(d, f(t_1, \ldots, t_n))$
- a test suite $\mathcal{T}$ satisfies the *all-du-paths* criterion if it includes all the *def-clear*(f) runs from s to each state in $dpu(s, f(t_1, \ldots, t_n))$ and to each state in $dcu(s, f(t_1, \ldots, t_n))$

Empirical studies [33,16] have shown that there is little difference in terms of the number of test cases sufficient to satisfy the least demanding criterion, *all-def*, and the most demanding criterion, *all-du-paths*. However, there is a hidden cost in satisfying the *all-du-paths* criterion, in that it is substantially more difficult to determine whether or not *all-du-paths* is actually satisfied: many definition-use (du-)paths can actually be non-executable, and it is frequently a difficult and time-consuming job to determine which du-paths are truly non-executable. For this reason, the most commonly adopted data flow criterion is the *all-uses*.

3 Generating Test Cases from ASMs

In the previous section we have defined a family of data flow coverage criteria for ASM specifications. We now address the problem of how to generate test suites satisfying a given set of such criteria for an ASM model. Obviously, the hardest problem we must tackle is the need to reason in terms of all the possible computations of a given machine, i.e. to explore the state space of the machine. In the following, we elucidate an approach we are currently investigating based on model checking.

In the last decade connections between test case generation and model checking have been widely explored. In [27] and [12] on-the-fly model checking algoritms are applied to test generation. In [17] the capability of SMV and SPIN to construct counterexamples is applied to generate test suites satisfying control flow coverage criteria. In [2] an approach to mutation analysis on model checking specifications is discussed.

Model Checking

The underlying idea of this approach is to represent data flow coverage criteria in temporal logic so that the problem of generating test suites satisfying a specific set of coverage criteria is reduced to the problem of finding witnesses for a set of temporal formulas, as proposed in [25]. The capability of model checkers to construct witnesses [9] enables the test generation process to be fully automatic. In particular, in [25] the model checker SMV [29] is used.

When the model checker determines that a formula with an existential path quantifier is true, it will find a computation path that demonstrates the success of the formula (witness). For this specific problem, Hong et al. introduce a subset of the existential fragment of CTL (ECTL) [8], called WCTL. An ECTL formula f is a WCTL formula if (i) f contains only **EX**, **EF**, and **EU**, where **E** ("for some path") is an existential path quantifier, **X** (next time), **F** (eventually), and **U** (until) are modal operators, and (ii) for every subformula of f of the form $f_1 \wedge \ldots \wedge f_n$, every conjunct f_i except at most one is an atomic proposition. For a full description refer to [25].

Since the original approach is strongly based on control flow graphs, once again we need to modify it in order to adapt it to ASMs. Given two statements $d_{R_i}^{f,l}$ (a definition of function f in line l of rule R_i) and $u_{R_j}^{f,m}$ (a computation–$c_{R_j}^{f,m}$–or predicate–$p_{R_j}^{f,m}$–use of function f in line m of rule R_j), a pair $(d_{R_i}^{f,l}, u_{R_j}^{f,m})$ is a *definition-use* pair (for short, "*du-pair*") if there is a *def-clear*(f) path from state s to state s' such that f is in def_{state} in s with respect to $d_{R_i}^{f,l}$, and in *p/c-use*$_{state}$ in s' with respect to $u_{R_j}^{f,m}$ (see Figure 3).

We want to check that there exist a path that contains a state s where the value of f is modified, and from s there is a path in which f is not redefined until we reach a state where the value of f is used (in a predicate or computation). In ASM terms, this means that we are looking for a run such that at some point we

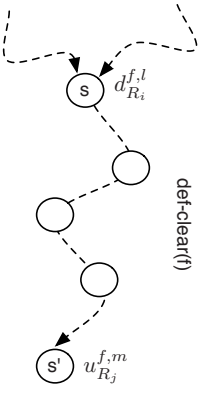


Fig. 3. A du-pair

reach a state where the predicate guarding the selected update $d^{f,l}_{R_i}$ evaluates to true (therefore triggering a rule where the value of f is defined) and then all the predicates guarding updates of f evaluate to false (so f is not redefined) until we reach a state where, in case of predicate use the predicate $p^{f,m}_{R_j}$ holds, in case of computation use the condition guarding the statement $c^{f,m}_{R_j}$ must be true.

This can be formalised in WCTL as:

$$wctl(d^{f,l}_{R_i}, u^{f,m}_{R_j}) = \mathbf{EF}(sd^f_{R_i} \wedge \ \mathbf{EXE}(\neg sd^f_{\mathcal{M}} \ \mathbf{U} \ su^f_{R_j}))$$

where

$$sd^f_{R_i} \equiv guard(d^{f,l}_{R_i})$$

$$sd^f_{\mathcal{M}} \equiv \bigvee_{d \in def_{\mathcal{M}}(f) \setminus \{su^f_{R_j}\}} guard(d)$$

$$su^f_{R_j} = \begin{cases} p^{f,m}_{R_j} & \text{if } u^{f,m}_{R_j} \text{ is a predicate} \\ guard(c^{f,m}_{R_j}) & \text{if } u^{f,m}_{R_j} \text{ is a computation} \end{cases}$$

Observe that $sd^f_{R_i}$ actually identifies the state before the function is updated. This is not a problem as we are guaranteed that, since the guard is true, in the next state the update will take place. A similar remark holds for $su^f_{R_j}$ when $u^{f,m}_{R_j}$ is a computation. Moreover, we do not include the guard of $u^{f,m}_{R_j}$ in the disjunct $sd^f_{\mathcal{M}}$ even when f is defined within its scope because it actually uses the current value of f used before being redefined (see the characterisation of $wctl(d^{y,3}_{R_2}, c^{y,2}_{R_2})$ in Example 3).

If a statement is in the scope of more than one guard, the predicate $guard()$ will denote the conjunction of all the guards it is nested into.

Example 3. Consider again the ASM in Example 1. Suppose we want to find runs covering the pairs $(d^{x,6}_{R_1}, p^{x,1}_{R_2})$ and $(d^{y,3}_{R_2}, c^{y,2}_{R_2})$. This is equivalent to searching for witnesses for the following formulas

$$wctl(d^{x,6}_{R_1}, p^{x,1}_{R_2}) = \mathbf{EF}((y > 5) \wedge \mathbf{EXE}(\neg(y > 5)\ \ \mathbf{U}\ \ (z = x)))$$

$$wctl(d^{y,3}_{R_2}, c^{y,2}_{R_2}) = \mathbf{EF}((z = x) \wedge\ \mathbf{EXE}(\neg((x = 0) \vee (x > 0\ \ \&\ \ z \neq x))\ \ \mathbf{U}\ \ (z = x)))$$

Let us now describe how to generate a set of test sequences satisfying the *all-defs* and *all-uses* criteria for a set of pairs $(d^{f,l}_{R_i}, u^{f,m}_{R_j})$. Basically, we associate a formula $wctl(d^{f,l}_{R_i}, u^{f,m}_{R_j})$ with every pair $(d^{f,l}_{R_i}, u^{f,m}_{R_j})$ and characterise each coverage criterion in terms of *witness-sets* for the formulas $wctl(d^{f,l}_{R_i}, u^{f,m}_{R_j})$. This reduces the problem of generating a test suite to the problem of finding a witness-set for a set of WCTL formulas. We say that Π is a witness-set for a set of WCTL formulas if it consists of a set of finite paths such that, for every formula f in F there is a finite path π in Π that is a witness for f.

All-defs. A test suite $\mathcal{T}$ satisfies the *all-defs* coverage criterion if, for every definition $d^{f,l}_{R_i}$ and some use $u^{f,m}_{R_j}$, there is a test sequence in $\mathcal{T}$ covering some *def-clear*(f) run from a state where f is in def_{state} w.r.t. $d^{f,l}_{R_i}$ to a state where f is in $p/c\text{-}use_{state}$ w.r.t. $u^{f,m}_{R_j}$.

A test suite $\mathcal{T}$ satisfies the *all-defs* coverage criterion if and only if it is a witness-set for

$$\{\bigvee_{u^{f,m}_{R_j} \in use_{\mathcal{M}}(f)} \mathbf{wctl}(d^{f,l}_{R_i}, u^{f,m}_{R_j}) \mid d^{f,l}_{R_i} \in def_{\mathcal{M}}(f)\} \text{ for every function } f \text{ in } \mathcal{M}\}$$

where $use_{\mathcal{M}}(f) = p\text{-}use_{\mathcal{M}}(f) \bigcup c\text{-}use_{\mathcal{M}}(f)$ denotes the sets of all uses in $\mathcal{M}$.

All-uses. A test suite $\mathcal{T}$ satisfies the *all-uses* coverage criterion if, for every definition $d^{f,l}_{R_i}$ and every use $u^{f,m}_{R_j}$, there is a test sequence in $\mathcal{T}$ covering some *def-clear*(f) run from a state where f is in def_{state} w.r.t. $d^{f,l}_{R_i}$ to a state where f is in $p/c\text{-}use_{state}$ w.r.t. $u^{f,m}_{R_j}$. A test suite $\mathcal{T}$ satisfies the *all-uses* coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}(d^{f,l}_{R_i}, u^{f,m}_{R_j}) \mid d^{f,l}_{R_i} \in def_{\mathcal{M}}(f), u^{f,m}_{R_j} \in use_{\mathcal{M}}(f) \text{ for every function } f \text{ in } \mathcal{M}\}$$

Observe that, in the worst case, the number of formulas can be quadratic in the size of statements in the ASM.

Example 4. Consider again the ASM $\mathcal{M}$ defined in Example 1. Suppose we want to satisfy the *all-defs* criterion for variable z. This would involve to find a witness for the following disjunctions

$$\{wctl(d^{z,2}_{R_2}, p^{z,1}_{R_1}) \vee wctl(d^{z,2}_{R_2}, p^{z,1}_{R_2}) \vee wctl(d^{z,2}_{R_2}, p^{z,4}_{R_2}) \vee wctl(d^{z,2}_{R_2}, c^{z,5}_{R_2}),$$
$$wctl(d^{z,5}_{R_2}, p^{z,1}_{R_1}) \vee wctl(d^{z,5}_{R_2}, p^{z,1}_{R_2}) \vee wctl(d^{z,5}_{R_2}, p^{z,4}_{R_2}) \vee wctl(d^{z,5}_{R_2}, c^{z,5}_{R_2})\}$$

If we want to satisfy the *all-uses* criterion for variable z we need to find a witness-set for the set of formulas

$$\{wctl(d^{z,2}_{R_2}, p^{z,1}_{R_1}), wctl(d^{z,2}_{R_2}, p^{z,1}_{R_2}), wctl(d^{z,2}_{R_2}, p^{z,4}_{R_2}), wctl(d^{z,2}_{R_2}, c^{z,5}_{R_2}),$$
$$wctl(d^{z,5}_{R_2}, p^{z,1}_{R_1}), wctl(d^{z,5}_{R_2}, p^{z,1}_{R_2}), wctl(d^{z,5}_{R_2}, p^{z,4}_{R_2}), wctl(d^{z,5}_{R_2}, c^{z,5}_{R_2})\}$$

4 Discussion and Future Work

Data flow coverage criteria can be used to bridge the gap between control flow testing and the ambitious and often unfeasible requirement to exercise every path in a program. Originally, they were developed for single modules in procedural languages [28,31,32], but have since been extended for interprocedural programs in procedural languages [23], object-oriented programming languages [22], and modeling languages such as UML [4]. Tools to check the adequacy of test cases w.r.t data flow coverage criteria are also being developed (see for instance Coverlipse [11]).

In this paper we have presented a family of data flow coverage criteria for Abstract State Machines based on those introduced in [32]. We have discussed why such criteria could not be directly applied to ASMs, and modified them accordingly. The criteria defined here focus on the interaction of portions of the ASM linked by the flow of data rather than merely by the flow of control. Therefore, they can also serve as a guide for a clever selection of critical paths for testing.

We have also introduced a model checking-based approach to generate automatically test suites satisfying the *all-defs* and *all-uses* criteria by formalising such criteria in temporal logic. Our approach builds on the work in [25], which for this purpose uses CTL as temporal logic and SMV as model checker.

We are not advocating that data flow coverage criteria should be applied to all the variables in an ASM model, but to a subset of critical variables. Moreover, to reduce the complexity of the model (and therefore its state space) it would be interesting to investigate the effectiveness of the approach onto ASM slices [30].

At the moment there is no tool support for the theory illustrated in this paper. In fact, although a formal mapping from ASMs to SMV has been defined [34], the interface developed in [7] is linked to the Workbench tool [6] which unfortunately is not maintained anymore. However, there are plans to adapt it to work with the ASM tools currently available [13,1]; this will allow us to develop a testing tool based on our approach and thus to evaluate its effectiveness and scalability by applying it to a number of case studies. We also intend to explore the possibility of adapting our data flow coverage criteria to work with the SPIN model checker, exploiting the ASM to PROMELA mapping defined in [19,14].

For the purpose of this paper we have worked on a minimal class of ASMs. However, we are extending our approach to support inter-agent data flow analysis. Other data flow coverage criteria, such as those proposed by Ntafos [31] and Laski and Korel [28] do not seem to be adaptable to ASMs, as they are intrinsically linked to control flow graphs (they are strictly based on static analysis and observations of control flow graphs).

We are also interested in comparing our criteria to those proposed in [18,19] with the aim of providing a subsumption hierarchy for coverage criteria for ASMs, in line with that presented in [10], and possibly formulate a testing strategy for ASMs.

Finally, we wish to investigate alternative approaches for data flow testing ASM models. In particular, we want to study the problem of using finite state machines (FSMs) generated from Abstract State Machine specifications for data flow testing. The problem of producing FSMs from ASMs for test case generation has been studied in [20,5] where algorithms are given to generate optimised FSMs. Our idea would be to conduct data flow analysis and testing of ASMs by adapting existing techniques for data flow testing FSMs [26,4].

References

1. ASMETA: a tool set for the ASM, `http://sourceforge.net/projects/asmeta`
2. Black, P.E., Okun, V., Yesha, Y.: Mutation operators for specifications. Automated Software Engineering, 81 (2000)
3. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
4. Briand, L.C., Labiche, Y., Lin, Q.: Improving statechart testing criteria using data flow information. In: ISSRE, pp. 95–104 (2005)
5. Campbell, C., Veanes, M.: State Exploration with Multiple State Groupings. In: Abstract State Machines, pp. 119–130 (2005)
6. Castillo, G.D.: 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)
7. Castillo, G.D., Winter, K.: Model Checking Support for the ASM High-Level Language. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 331–346. Springer, Heidelberg (2000)
8. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
9. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: DAC, pp. 427–432 (1995)
10. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Trans. Software Eng. 15(11), 1318–1332 (1989)
11. Coverlipse, `http://coverlipse.sourceforge.net/index.php`
12. de Vries, R.G., Tretmans, J.: On-the-fly conformance testing using spin. STTT 2(4), 382–393 (2000)
13. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An Extensible ASM Execution Engine. Fundam. Inform. 77(1-2), 71–103 (2007)
14. Farahbod, R., Glsser, U., Ma, G.: Model Checking CoreASM Specifications
15. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Trans. Software Eng. 14(10), 1483–1498 (1988)
16. Frankl, P.G., Weyuker, E.J.: A formal analysis of the fault-detecting ability of testing methods. IEEE Trans. Software Eng. 19(3), 202–213 (1993)

17. Gargantini, A., Heitmeyer, C.L.: Using model checking to generate tests from requirements specifications. In: ESEC/SIGSOFT FSE, pp. 146–162 (1999)
18. Gargantini, A., Riccobene, E.: ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. J. UCS 7(11), 1050–1067 (2001)
19. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to Generate Tests from ASM Specifications. Abstract State Machines, 263–277 (2003)
20. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA, pp. 112–122 (2002)
21. Gurevich, Y.: Specification and validation methods (1995)
22. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: SIGSOFT 1994: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, pp. 154–163. ACM, New York (1994)
23. Harrold, M.J., Soffa, M.L.: Interprocedual data flow testing. In: TAV3: Proceedings of the ACM SIGSOFT 1989 third symposium on Software testing, analysis, and verification, pp. 158–167. ACM, New York (1989)
24. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
25. Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data Flow Testing as Model Checking. In: ICSE, pp. 232–243 (2003)
26. Hong, H.S., Kim, Y.G., Cha, S.D., Bae, D.-H., Ural, H.: A test sequence selection method for statecharts. Softw. Test, Verif. Reliab. 10(4), 203–227 (2000)
27. Jéron, T., Morel, P.: Test generation derived from model-checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 108–121. Springer, Heidelberg (1999)
28. Laski, J.W., Korel, B.: A data flow oriented program testing strategy. IEEE Trans. Software Eng. 9(3), 347–354 (1983)
29. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
30. Nowack, A.: Slicing abstract state machines. In: Zimmermann, W., Thalheim, B. (eds.) ASM 2004. LNCS, vol. 3052, pp. 186–201. Springer, Heidelberg (2004)
31. Ntafos, S.C.: On required element testing. IEEE Trans. Software Eng. 10(6), 795–803 (1984)
32. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Software Eng. 11(4), 367–375 (1985)
33. Weyuker, E.J.: More experience with data flow testing. IEEE Trans. Software Eng. 19(9), 912–919 (1993)
34. Winter, K.: Model Checking for Abstract State Machines. J. UCS 3(5), 689–701 (1997)

A Verified AsmL Implementation of Belief Revision*

Christoph Beierle[1] and Gabriele Kern-Isberner[2]

[1] Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2] Dept. of Computer Science, TU Dortmund, 44221 Dortmund, Germany
christoph.beierle@fernuni-hagen.de,
gabriele.kern-isberner@cs.uni-dortmund.de

Abstract. Belief revision is a key functionality for any intelligent agent being able to perceive pieces of knowledge from its environment and to give back sentences she believes to be true with a certain degree of belief. We report on a refinement of a previous, abstract ASM specification of CONDOR, a system modeling such an agent, to a fully operational specification implemented in AsmL. The complete AsmL implementation of various belief revision operators is presented, demonstrating how using AsmL enabled a high-level implementation that minimizes the gap between the abstract specification of the underlying concepts and the executable code in the implemented system. Based on ASM refinement and verification concepts, a full mathematical correctness proof for different belief revision operators realized in CONDOR@AsmL is given.

1 Introduction

The paradigm of an agent as an integrated system embedded in and communicating with its environment has gained much attention and had a major impact on many research areas particularly in AI. Belief revision is a key functionality for any intelligent agent being able to perceive pieces of knowledge (general rules, evidence, etc.) from its environment and to give back sentences she believes to be true with a certain degree of belief (see e.g. [1,10,13,17]).

The CONDOR system represents an approach to model such an intelligent agent. For the representation of knowledge, general conditionals viewed upon as default rules are used. A conditional "*if A then B*" establishes a plausible, probable, possible etc. connection between the *antecedent* A and the *consequent* B, but still allows exceptions. Together with the given pieces of knowledge, the agent's epistemic state determines her reasoning and beliefs, and since the agent is assumed to live in a dynamic environment, she must be able to update or revise her own state of belief in the light of new information.

In [3] we developed a high-level ASM specification CONDORASM for the CONDOR system, enabling us to elaborate crucial interdependencies between

* The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grants BE 1700/7-1 and KE 1413/2-1).

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 98–111, 2008.

different aspects of knowledge representation, knowledge discovery, and belief revision, and to precisely formalize central correctness requirements in the ASM framework. For belief revision of epistemic states with conditionals, these are the *success postulate* (revising or updating an epistemic state Ψ by a set of conditionals $\mathcal{R}$ yields an epistemic state Ψ^* where all conditionals in $\mathcal{R}$ are believed) and the *stability axiom* (if already in Ψ all conditionals in $\mathcal{R}$ are accepted, then $\Psi^* = \Psi$) [14, p. 113, (CSR-1) and (CSR-2)]. In [3], we deliberately left various universes and functions of CondorASM abstract, aiming at a broad applicability of our approach. In particular, by leaving the universe $\mathcal{Q}$ of quantitative and qualitative scales abstract, the specification applies to conditionals in both quantitative logic (e.g. probabilistic logic [18]) and qualitative approaches (e.g. [11,19]).

A refinement CondorASM$_{\mathcal{O}}$ of CondorASM to qualitative conditionals [1,11] equipped with the semantics of ordinal conditional functions (OCFs) [19] is given in [4], where OCFs are used to represent the epistemic state of an agent. The ASM methodology allowed us to precisely describe sophisticated knowledge management tasks investigated in belief change [1,10,13,14,15] (for a recent discussion, see [17]). For instance, we could elaborate the similarities and delicate differences between update and genuine revision operators involving full epistemic states and provide corresponding ASM specifications, relying on the concept of so-called c-revisions [15] as a uniform core-methodology.

In this paper, we report on a refinement of CondorASM$_{\mathcal{O}}$ to a fully operational specification Condor@AsmL using AsmL [2,12], enabling a high-level implementation that minimizes the gap between the abstract specification of the underlying concepts and the executable code in the implemented system. We will demonstrate this aspect by presenting a complete implementation of the sophisticated core function

$$cRevision : \ EpState_{\mathcal{O}} \times \mathcal{P}(Sen_{\mathcal{U}}) \rightarrow EpState_{\mathcal{O}}$$

left abstract in [4]. Based on ASM refinement and verification concepts [7], we will provide a full mathematical correctness proof for different belief revision operators implemented in Condor@AsmL.

A first version of Condor@AsmL was developed in [16]; for a short system description of the current version used in this paper see [5] where also a system walk-through with examples of various reasoning tasks is provided.

Figure 1 shows the start menu of the system where the user can choose among its top-level functionalities as they were already present in [3]; for convenience for the user, additional items for interated revision and update [8] were added. Since our focus was on a lean implementation, currently there is only a command line and file I/O interface. This enables both an easy integration of Condor@AsmL with other systems and also adding a graphical user interface in a modular way.

The rest of this paper is organised as follows: We briefly recall the basic notions of qualitative conditional logic (Sec. 2) and the concepts of consistency and c-revisions for this context (Sec. 3). After presenting the basic universes and functions of Condor@AsmL (Sec. 4), the AsmL implementation of belief

```
*********************************
**    Experience Condor@AsmL   **
**    Vers. 2.2 (2008-02-17)   **
*********************************

Choose an  action:
      [0] Initialization
      [1] Load
      [2] Query
      [3] Belief and Disbelief
      [4] Revision and Update
      [5] Iterated Revision
      [6] Iterated Update
      [7] Diagnosis
      [8] What-if-Analysis
      [9] Exit Program
```

Fig. 1. Start menu of the CONDOR@AsmL system

revision is described in detail in Sec. 5, along with a complete correctness proof. Section 6 tells some experiences made using CONDOR@AsmL and contains concluding remarks.

2 Background

We start with a propositional language $\mathcal{L}$, generated by a finite set Σ of atoms $a, b, c, \ldots$. The formulas of $\mathcal{L}$ will be denoted by uppercase Roman letters $A, B, C, \ldots$. For conciseness of notation, we will omit the logical *and*-connective, writing AB instead of $A \wedge B$, and overlining formulas will indicate negation, i.e. $\overline{A}$ means $\neg A$. Let Ω denote the set of possible worlds over $\mathcal{L}$; Ω will be taken here simply as the set of all propositional interpretations over $\mathcal{L}$ and can be identified with the set of all complete conjunctions over Σ. For $\omega \in \Omega$, $\omega \models A$ means that the propositional formula $A \in \mathcal{L}$ holds in the possible world ω.

By introducing a new binary operator $|$, we obtain the set

$$(\mathcal{L} \mid \mathcal{L}) = \{(B|A) \mid A, B \in \mathcal{L}\}$$

of *conditionals* over $\mathcal{L}$. $(B|A)$ formalizes "*if* A *then* B" and establishes a plausible, probable, possible etc connection between the *antecedent* A and the *consequent* B. Here, conditionals are supposed not to be nested, that is, antecedent and consequent of a conditional will be propositional formulas.

A conditional $(B|A)$ is an object of a three-valued nature, partitioning the set of worlds Ω in three parts: those worlds satisfying AB, thus *verifying* the conditional, those worlds satisfying $A\overline{B}$, thus *falsifying* the conditional, and those worlds not fulfilling the premise A and so which the conditional may not be

applied to at all. This allows us to represent $(B|A)$ as a *generalized indicator function* going back to [9] (where u stands for *unknown* or *indeterminate*):

$$(B|A)(\omega) = \begin{cases} 1 & \text{if } \omega \models AB \\ 0 & \text{if } \omega \models A\overline{B} \\ u & \text{if } \omega \models \overline{A} \end{cases}$$

To give appropriate semantics to conditionals, they are usually considered within richer structures such as *epistemic states*. Besides certain (logical) knowledge, epistemic states also allow the representation of preferences, beliefs, assumptions of an intelligent agent. Basically, an epistemic state allows one to compare formulas or worlds with respect to plausibility, possibility, necessity, probability, etc.

Well-known qualitative, ordinal approaches to represent epistemic states are Spohn's *ordinal conditional functions, OCFs*, (also called *ranking functions*) [19], and *possibility distributions* [6], assigning degrees of plausibility, or of possibility, respectively, to formulas and possible worlds. In such qualitative frameworks, a conditional $(B|A)$ is valid (or *accepted*), if its confirmation, AB, is more plausible, possible, etc. than its refutation, $A\overline{B}$; a suitable degree of acceptance is calculated from the degrees associated with AB and $A\overline{B}$.

The concept underlying CONDOR@AsmL is based on Spohn's OCFs [19]. An OCF

$$\kappa : \Omega \to \mathbb{N}$$

expresses degrees of plausibility of propositional formulas where a higher degree denotes "less plausible". At least one world must be regarded as being normal; therefore, $\kappa(\omega) = 0$ for at least one $\omega \in \Omega$. Each such ranking function can be taken as the representation of a full epistemic state of an agent. Each such κ uniquely extends to a function (also denoted by κ)

$$\kappa : Sen_{\mathcal{U}} \to \mathbb{N} \cup \{\infty\}$$

mapping sentences and rules to $\mathbb{N} \cup \{\infty\}$ and being defined by

$$\kappa(A) = \begin{cases} min\{\kappa(\omega) \mid \omega \models A\} & \text{if } A \text{ is satisfiable} \\ \infty & \text{otherwise} \end{cases}$$

for sentences $A \in Fact_{\mathcal{U}}$ and by

$$\kappa((B|A)) = \begin{cases} \kappa(AB) - \kappa(A) & \text{if } \kappa(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

for conditionals $(B|A) \in Rule_{\mathcal{U}}$.

The degree of belief of an agent being in epistemic state κ with respect to a default rule $(B|A)$ is determined by the satisfaction relation $\models_{\mathcal{O}}$ for quantified sentences $(B|A)[m]$ defined by:

$$\kappa \models_{\mathcal{O}} (B|A)[m] \quad \text{iff} \quad \kappa(AB) + m \; < \; \kappa(A\overline{B}) \tag{1}$$

Thus, $(B|A)$ is believed in κ with degree of belief m if the rank of AB (verifying the unquantified conditional) is more than m smaller than the rank of $A\overline{B}$ (falsifying the unquantified conditional).

The satisfaction relation for unquantified sentences $(B|A)$ is obtained from (1) by

$$\kappa \models_{\mathcal{O}} (B|A) \quad \text{iff} \quad \kappa(AB) \; < \; \kappa(A\overline{B}) \tag{2}$$

Hence κ *accepts* the conditional $(B|A)$ iff $\kappa \models_{\mathcal{O}} (B|A)$.

3 Consistency and C-Revisions

Since a rational agent will never believe both $(B|A)$ and $(\overline{B}|A)$ at the same time, not every set $\mathcal{R}$ of conditionals can be believed in some well-defined epistemic state; this is guaranteed only if $\mathcal{R}$ is consistent.

Definition 1 (consistent). *A set of conditionals $\mathcal{R}$ is* consistent *iff there exists an OCF κ that accepts every $r \in \mathcal{R}$.*

The consistency of $\mathcal{R}$ in a qualitative framework can be characterized by the notion of *tolerance* [11]:

Definition 2 (tolerate; ordered partition). *A conditional $(B|A)$ is* tolerated *by a set of conditionals $\mathcal{R}$ iff there is a world ω such that ω verifies $(B|A)$ (i.e. $(B|A)(\omega) = 1$) and ω does not falsify any of the conditionals in $\mathcal{R}$ (i.e. $r(\omega) \neq 0$ for all $r \in \mathcal{R}$). A partition $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_k$ of $\mathcal{R}$ is an* ordered partition *of $\mathcal{R}$ if each conditional in $\mathcal{R}_m$ is tolerated by $\bigcup_{j=m}^{k} \mathcal{R}_j$, $0 \leqslant m \leqslant k$.*

Theorem 1 (see [11]). *A set $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ of conditionals is consistent iff there is an ordered partition of $\mathcal{R}$.*

C-revisions [15] transform an epistemic state κ and a (consistent) set of conditionals $\mathcal{R}$ into a new epistemic state accepting these conditionals. A characterization theorem of [15] shows that every c-revision of κ and $\mathcal{R}$ can be obtained by adding to each $\kappa(\omega)$ values for each rule $r_i \in \mathcal{R}$, depending on whether ω verifies or falsifies r_i. If $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_k$ is an ordered partition of $\mathcal{R}$, the following yields a c-revision [15]: Set successively, for each partitioning set $\mathcal{R}_m$, $0 \leqslant m \leqslant k$, starting with $\mathcal{R}_0$, and for each conditional $r_i = (B_i|A_i) \in \mathcal{R}_m$

$$\kappa_i^- \; := \; 1 \; + \min_{\substack{\omega \models A_i B_i \\ r(\omega) \neq 0, \forall r \in \mathcal{R}_m \cup \ldots \cup \mathcal{R}_k}} (\kappa(\omega) \; + \sum_{\substack{r_j \in \mathcal{R}_0 \cup \ldots \cup \mathcal{R}_{m-1} \\ r_j(\omega) = 0}} \kappa_j^-) \tag{3}$$

Finally, choose κ_0 appropriately to make

$$\kappa^*(\omega) \; = \; \kappa_0 + \kappa(\omega) + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B}_i}} \kappa_i^- \tag{4}$$

an ordinal conditional function.

4 Basic Universes and Functions of CONDOR@AsmL

The central universes $Rule_{\mathcal{U}}$, $\mathcal{P}(Rule_{\mathcal{U}})$ and $EpState_{\mathcal{O}}$ of [4] are implemented in CONDOR@AsmL by

```
structure Rule
   Conclusion as Formula
   Premise as Formula

type RuleSet = Set of Rule
type State = Map of World to Integer
```

where, given a universe Σ of propositional variables, `Formula` is the type of all propositional sentences over Σ (implementing $Fact_{\mathcal{U}}$ from [4]) and `World` is the type of all possible worlds that can be distinguished using Σ (i.e. `World` is the type of all complete conjuntions over Σ).

Since due to space restrictions we are not able to present the complete CONDOR@AsmL code here, for the three auxiliary functions

```
verify(w as World, in_R as RuleSet) as RuleSet
falsify(w as World, in_R as RuleSet) as RuleSet
accept(kappa as State, in_R as RuleSet) as RuleSet
```

available in CONDOR@AsmL, we will tacitly assume the following assumptions in the rest of this paper: For any world `w` in `World`, any set of rules $\mathcal{R}$ in `RuleSet`, and any epistemic state `kappa` in `State`, CONDOR@AsmL computes

$$\texttt{verify}(\texttt{w}, \mathcal{R}) = \{r \in \mathcal{R} \mid \texttt{w} \text{ verifies } r\} \tag{5}$$

$$\texttt{falsify}(\texttt{w}, \mathcal{R}) = \{r \in \mathcal{R} \mid \texttt{w} \text{ falsifies } r\} \tag{6}$$

$$\texttt{accept}(\texttt{kappa}, \mathcal{R}) = \{r \in \mathcal{R} \mid \texttt{kappa} \text{ accepts } r\} \tag{7}$$

Note that (5)–(7) are ensured easily in CONDOR@AsmL by implementing some fundamental primitives of propositional logic and simple basic operations on `State`.

The AsmL variables

```
var CurrState as State = {->}
var StateBeforeUpdate as State = {->}
var NewRulesSinceUpdate as RuleSet = {}
var Omega as Seq of World = []
var Partition as Map of Integer to RuleSet  = {->}
```

implement the nullary functions *currstate*, *stateBeforeUpdate*, *newRulesSinceUpdate* of [4], `Omega` will contain all complete conjunctions over the propositional variables Σ, and `Partition` will hold an ordered partition of a set of rules checked for consistency.

5 Implementation of Belief Revision

In the following subsections, we will first present the AsmL code for the required auxiliary functions, along with propositions that will be used for the proof of the main theorem stating the correctness of CONDOR@AsmL's implementation of belief revision.

5.1 Computing an Ordered Partition

Figure 2 shows the AsmL code for computing an ordered partition. The binary function `buildPartition(in_R,p)` returns an ordered partition (i.e. a function mapping natural numbers to sets of rules) of `in_R` if it exists, and the empty map otherwise. It takes a set of rules `in_R` (still to be partitioned) and a partition `p` (of those rules that have already been assigned to a particular partition set R_m). Initially, `in_R` contains all given rules and `p` is the empty function {`->`}.

```
buildPartition(in_R as RuleSet, in_partition as Map of Integer to RuleSet)
                                                  as Map of Integer to RuleSet
// recursively build proper partition
 var rules as RuleSet = in_R
 var partition as Map of Integer to RuleSet = in_partition
 let tolerating_worlds = {w | w in Omega where falsify(w,rules) = {} }
 let tolerated_rules = {r | w in tolerating_worlds, r in verify(w,rules)}
 step
  if tolerated_rules = {}
  then partition := {->}   // proper partition does not exist
  else                     // extend current partition
    let next_index = Size(partition)   // next index, starts with 0
    step partition := partition + {next_index -> tolerated_rules}
         rules := rules - tolerated_rules
    step if rules <> {}  // partition remaining rules recursively
         then partition := buildPartition(rules, partition)
 step return partition
```

Fig. 2. AsmL code for determining an ordered partition of a set of rules

Proposition 1. *For any given set of rules $\mathcal{R}$,* `buildPartition(`$\mathcal{R}$`,{->})` *computes an ordered partition if it exists, and the empty map* {`->`} *if no ordered partition of $\mathcal{R}$ exists.*

Proof. We will use the invariant INV($\mathcal{S}$, p) for recursive calls of `buildPartition(`$\mathcal{S}$`,` p`)`, where $\mathcal{S}$ is a set of rules, and p is a partition of a set of rules:

$$\text{INV}(\mathcal{S}, p): \quad \begin{array}{l} \bullet\ \mathcal{R} = \mathcal{S} \cup \bigcup_{i \to R_i \in p} R_i \\ \bullet\ p \text{ is an ordered partition of } \bigcup_{i \to R_i \in p} R_i \\ \bullet\ \text{every rule } r \in \mathcal{S} \text{ is tolerated by } \bigcup_{i \to R_i \in p} R_i \end{array}$$

Let `buildPartition(in_R, in_p)` either denote the initial call `buildPartition(`$\mathcal{R}$`,{->})` or any subsequent recursive call caused directly or indirectly by this initial call. We will prove that INV(`in_R`, `in_p`) holds for any of these calls and show that the conculsion of the proposition follows.

For the inital call, INV(`in_R`, `in_p`) holds trivially since `in_p` is empty. So let `buildPartition(in_R, in_p)` be any subsequent call and assume that INV(`in_R`, `in_p`) holds for this call. The first two let-constructs in Fig. 2 ensure that `tolerating_worlds` contains all worlds that do not falsify any of the rules in `in_R`, and that `tolerated_rules` contains all rules from `in_R` verified by some of these worlds. Using Def. 2, we thus have

$$\texttt{tolerated\_rules} = \{r \mid r \in \texttt{in\_R},\ r \text{ is tolerated by } \texttt{in\_R}\}$$

If `tolerated_rules` is empty, no ordered partition of `in_R` exists according to Def. 2, and together with INV(`in_R`, `in_p`) we conclude that `in_p` can not be extended to an ordered partition of $\mathcal{R}$. Thus, there is no ordered partition of $\mathcal{R}$ and the initial call `buildPartition(`$\mathcal{R}$`,{->})` terminates, returning the empty map.

If `tolerated_rules` is not empty, `in_p` is an ordered partion

$$\{0 \to R_0,\ 1 \to R_1,\ \ldots,\ m\text{–}1 \to R_{m-1}\}$$

where $m =$ `Size(in_p)`. According to Fig. 2, `partition` and `rules` are set to

$$\begin{aligned} \texttt{partition} &= \{0 \to R_0,\ 1 \to R_1,\ \ldots,\ m\text{–}1 \to R_{m-1},\ m \to \texttt{tolerated\_rules}\} \\ \texttt{rules} &= \texttt{in\_R} \setminus \texttt{tolerated\_rules} \end{aligned}$$

From INV(`in_R`, `in_p`) and Def. 2, we conclude that for these values INV(`rules`, `partition`) holds. If rules is empty, the computation terminates and INV(`rules`, `partition`) implies the conclusion of the proposition; if `rules` is not empty, the invariant holds for the recursive call `buildPartition(rules, partition)`.

It remains to be shown that `buildPartition(`$\mathcal{R}$`,{->})` terminates. This follows from the fact that in each recursive invocation of `buildPartition`, the set of rules in the first argument is decreased by at least one rule since `tolerated_rules` is checked to be non-empty. □

5.2 Checking Consistency

The AsmL code for inferring the consistency of a set of rules is now straightforward; it is given in Fig. 3.

```
isConsistent(in_R as RuleSet) as Boolean
  step Partition := buildPartition(in_R, {->})
  step return Partition <> {->}  // consistent iff Partition is non-empty
```

Fig. 3. AsmL code for inferring the consistency of a set of rules

Corollary 1. *For any given set of rules* `in_R`*,* `isConsistent(in_R)` *returns true if* `in_R` *is consistent, and false otherwise.*

Proof. This is a direct consequence of Proposition 1 and Theorem 1 since `isConsistent(in_R)` calls `buildPartition(in_R,{->})`, writes the result to `Partition` and checks whether this is the empty map or not. □

5.3 Computation of Penalties κ_i^-

For each rule $(B_i|A_i)$, we call the value κ_i^- from (3) the *penalty value* for this rule. Figure 4 shows the AsmL code for computing all penalty values according to (3).

Proposition 2. *For any world w, any set of rules $\mathcal{R}$ and any map* `kappaMinus` : `Rule` $\rightarrow$ `Integer` *let* $s =$ `sumPenalty(`w`,`$\mathcal{R}$`,kappaMinus)`*. Then*

$$s = \sum_{r_j \in \mathcal{R},\ r_j(w)=0} \texttt{kappaMinus}(r_j)$$

Proof. According to (6), `falsify(`w`,{`r`}) = {`r`}` says that the world w falsifies r (i.e., $r(w) = 0$); thus, the proposition follows by a straightforward induction on the size of $\mathcal{R}$. □

Proposition 3. *Let* `Partition` *be an ordered partition of a consistent rule set $\mathcal{R}$. For any epistemic state* `in_kappa`*,* `getKappaMinus(in_kappa)` *computes a mapping* `kappaMinus` : `Rule` $\rightarrow$ `Integer` *such that for all rules $(B_i|A_i)$ in $\mathcal{R}$ and the given* `Partition` *of $\mathcal{R}$,*

$$\kappa_i^- = \texttt{kappaMinus}((B_i|A_i))$$

is computed according to (3).

Proof. The AsmL code for `getKappaMinus` in Fig. 4 directly mimics the two nested loops ("for each $\mathcal{R}_m$", "for each $r_i \in \mathcal{R}_m$") in the definition of κ_i^- given in (3). Thus, we have to check that when executing

```
kappaMinus := kappaMinus union {r -> 1 + min x | w -> x in wk}
```

(cf. Fig. 4), the expression

```
1 + min x | w -> x in wk
```

defining the penalty value for rule `r` corresponds exactly to the defining expression for κ_i^- in (3). By an easy induction on the size of the partition we can show that when updating `kappaMinus` as above

$$\begin{aligned} \texttt{r\_next} &= \bigcup_{l \rightarrow R_l \in \texttt{Partition},\ l \in \{m,\ldots,\texttt{max\_index}-1\}} R_l \\ \texttt{r\_before} &= \bigcup_{l \rightarrow R_l \in \texttt{Partition},\ l \in \{0,\ldots,\texttt{m}-1\}} R_l \end{aligned}$$

holds. Thus, using (5) and (6), `ws` contains exactly those worlds (verifying the current rule and not falsifying any of the rules in the next partition parts) over which the minimum in (3) is determined. From Proposition 2, we get that `wk` associates to each world `w` in `ws` the sum given as argument to min in (3) (i.e., `in_kappa(w)` plus the sum of all penalty values of all rules in the previous partition parts falsified by `w`). By induction it follows that for every rule, `getKappaMinus` computes the penalty values according to (3). □

```
getKappaMinus(in_kappa as State) as Map of Rule to Integer
  var kappaMinus as Map of Rule to Integer = {->}
  var r_before as RuleSet = {}  // rules up to partition part r_m
  var r_next as RuleSet = {}    // rules from partition part r_m onwards

  let max_index = Size(Partition) - 1

  step for i = 0 to max_index
    r_next := r_next + Partition(i)  // initialize with all rules

  step for m = 0 to max_index
    let r_m = Partition(m)           // current partition part is r_m

    step foreach r in r_m
      // determine worlds that verify r and do not falsify any rule
      // of next parts:
      let ws = {w | w in Omega where verify(w,{r}) = {r} and
                                              falsify(w, r_next) = {}}
      // for these worlds determine old kappa + sum of penalties for
      // previous parts:
      let wk = {w -> in_kappa(w) + sumPenalty(w, r_before, kappaMinus)
                                                          | w in ws}
      // use minimum penalty to determine final penalty for r:
      kappaMinus := kappaMinus union {r -> 1 + min x | w -> x in wk}

    step r_next := r_next - r_m
    step r_before := r_before + r_m

  step return kappaMinus


sumPenalty(w as World, in_R as RuleSet, kappaMinus as Map of Rule
                                                  to Integer) as Integer
// sum up penalties of all rules falsified by given world
  var s as Integer = 0
  step foreach r in in_R
     if falsify(w, {r}) = {r}
     then s := s + kappaMinus(r)
  step return s
```

Fig. 4. AsmL code computing the penalty values κ_i^- for a (consistent) set of rules

5.4 Computation of a c-Revision

In [4] we used the abstract function

$$cRevision : \ EpState_{\mathcal{O}} \times \mathcal{P}(Sen_{\mathcal{U}}) \rightarrow EpState_{\mathcal{O}}$$

constrained by the condition that (3) and (4) are respected. Figure 5 shows its AsmL implementation.

```
cRevision(in_kappa as State, in_R as RuleSet) as State
// in_R must be consistent, Partition must contain its ordered partition
  var kappa as State = {->}
  step
    if accept(in_kappa, in_R) = in_R    // all rules accepted
    then kappa := in_kappa              // no change of in_kappa needed
    else
      let kappaMinus = getKappaMinus(in_kappa)
      kappa := {w -> k + sumPenalty(w, in_R, kappaMinus) | w -> k in
                                                              in_kappa}
      let kappaNull = min i | world -> i in kappa
      if kappaNull > 0                    // normalization required?
      then kappa := {w -> k - kappaNull | w -> k in kappa}
  step return kappa
```

Fig. 5. Computation of a cRevision of an epistemic state with respect to a set of rules

Proposition 4. *For any epistemic state* ***in_kappa*** *and any consistent rule set* ***in_R*** *with corresponding ordered* ***Partition****,* ***cRevision(in_kappa, in_R)*** *returns an epistemic state* ***kappa*** *satisfying equation (4) if* ***in_kappa*** *does not accept all rules in* ***in_R****; otherwise,* ***in_kappa*** *is returned.*

Proof. If all rules in `in_R` are already accepted by `in_kappa`, the first if-condition in Fig. 5 and (7) eunsure that `in_kappa` is returned. Otherwise, the function `kappa` is defined such that each world `w` is mapped to the sum of `in_kappa(w)` plus the sum of all penalty values of all rules in `in_R` falsified by `w`, where the penalty values have been determined by `get_kappaMinus(in_kappa)`. If this function `kappa` is not an OCF because no world is mapped to 0, `kappa` is updated by subtracting the normalization constant `kappaNull` from each function value. Together with Propositions 2 and 3, we conclude that the returned `kappa` satisfies (4). Moreover, `cRevision` obvioulsy terminates since all called functions terminate. □

5.5 Verification of Belief Revision

The different forms of belief revision realized in CONDOR@AsmL correspond exactly to the forms of belief revision defined in CONDORASM$_{\mathcal{O}}$ [4].

The AsmL code for the two most important belief revision operators, update and (genuine) revision, is shown in Fig. 6 and 7, where the nullary function

```
new_information() as RuleSet
```

provides the rules representing the new information to be taken into account.

Theorem 2 (Correctness of Update and Revision). *The belief revision operators "update" and "(genuine) revision" as implemented in* CONDOR*@AsmL fulfil the success postulate and the stability axiom.*

```
StateBeforeUpdate   := CurrState
NewRulesSinceUpdate := new_information
step if isConsistent(new_information) = false
     then error "NEW_INFORMATION for UPDATE inconsistent"
     else CurrState := cRevision(CurrState, new_information)
```

Fig. 6. Update operation with respect to new information

```
NewRulesSinceUpdate := NewRulesSinceUpdate union new_information
step if isConsistent(NewRulesSinceUpdate) = false
     then error "Set of rules for REVISE inconsistent"
     else CurrState := cRevision(StateBeforeUpdate, NewRulesSinceUpdate)
```

Fig. 7. (Genuine) revision with respect to new information

Proof. We first consider the "update" case (Fig. 6). Let κ = `CurrState` be any epistemic state and let $\mathcal{R}$ = `new_Information` be any set of rules. If $\mathcal{R}$ is inconsistent, this is detected by `isConsistent(`$\mathcal{R}$`)` according to Corollary 1 and an exception is raised. If $\mathcal{R}$ is consistent, let κ^* be the epistemic state computed by `cRevision(`κ`,`$\mathcal{R}$`)`. (Note that κ^* is well defined since `cRevision` terminates according to Proposition 4.) For the success postulate we have to show

$$\forall\,(B|A)\ \in\ R\ \ \kappa^*\ \models_{\mathcal{O}}(B|A)$$

which follows from Propositions 1 – 4 and Theorems 2 and 3 in [15]. The stability axiom

$$\text{if}\ \ \forall (B|A) \in R\ \ \kappa\ \models_{\mathcal{O}}(B|A)\ \ \text{then}\ \ \kappa^* = \kappa$$

directly follows from Proposition 4.

Since "(genuine) revision" (Fig. 7) is implemented analogously to "update" by calling `cRevision` after ensuring that the set of rules passed to `cRevision` is consistent, for revision both the success postulate as well as the stability axiom follow by an analogous argumentation. □

In [4] we also specified *focussing* in the context of diagnosis and hypothetical reasoning. These functionalities also rely on the core function *cRevision* which in any case is called only after a previous consistency check was successful. Hence, Theorem 2 can easily be adapted to prove that also these forms of belief revision implemented in CONDOR@AsmL fulfil the success postulate and the stability axiom.

6 Experiences with CONDOR@AsmL and Conclusions

Both reasoning with conditionals and adapting complete epistemic states is a cumbersome task even in small applications when doing it without the support of a system. Hence, CONDOR@AsmL proved to be very helpful in carrying out

experiments and case studies. For this, having a correct implementation is of course essential, which additionally motivated the request for verification.

Validating and verifying CONDOR@AsmL also helped us to detect a special case which was not dealt with properly in the higher-up specification CONDORASM$_{\mathcal{O}}$ in [4]: The rule for loading an epistemic state given in [4] must be extended by updating the bookkeeping functions *stateBeforeUpdate* and *newRulesSinceUpdate* (to the state loaded and {}, respectively) so that their correct values are available for a subsequent (genuine) revision step.

Among the many unsettled issues in belief change there is the question how to capture best the notion of *minimal change* (e.g. [1,8,10,14]). While c-revisions are designed with this principle in mind [14,15], we are currently working on an exact ASM characterization of it in the OCF framework as used in this paper.

References

1. Alchourrón, C.E., Gärdenfors, P., Makinson, P.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic 50(2), 510–530 (1985)
2. AsmL webpage (2007), `http://research.microsoft.com/foundations/asml/`
3. Beierle, C., Kern-Isberner, G.: Modelling conditional knowledge discovery and belief revision by Abstract State Machines. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 186–203. Springer, Heidelberg (2003)
4. Beierle, C., Kern-Isberner, G.: An ASM refinement and implementation of the Condor system using ordinal conditional functions. In: Prinz, A. (ed.) Proceedings 14th International Workshop on Abstract State Machines (ASM 2007). Agder University College, Grimstad, Norway (2007)
5. Beierle, C., Kern-Isberner, G., Koch, N.: A high-level implementation of a system for automated reasoning with default rules (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proc. of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008). LNCS, vol. 5195, pp. 147–153. Springer, Heidelberg (to appear, 2008)
6. Benferhat, S., Dubois, D., Prade, H.: Representing default rules in possibilistic logic. In: Proceedings 3rd International Conference on Principles of Knowledge Representation and Reasoning KR 1992, pp. 673–684 (1992)
7. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
8. Darwiche, A., Pearl, J.: On the logic of iterated belief revision. Artificial Intelligence 89, 1–29 (1997)
9. DeFinetti, B.: Theory of Probability, vol. 1,2. John Wiley & Sons, Chichester (1974)
10. Friedman, N., Halpern, J.Y.: Modeling belief in dynamic systems, Part II: Revison and update. Journal of Artificial Intelligence Research 10, 117–167 (1999)
11. Goldszmidt, M., Pearl, J.: Qualitative probabilities for default reasoning, belief revision, and causal modeling. Artificial Intelligence 84, 57–112 (1996)
12. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Theoretical Computer Science 343(3), 370–412 (2005)
13. Katsuno, H., Mendelzon, A.O.: On the difference between updating a knowledge base and revising it. In: Proceedings Second International Conference on Principles of Knowledge Representation and Reasoning, KR 1991, San Mateo, Ca., pp. 387–394. Morgan Kaufmann, San Francisco (1991)

14. Kern-Isberner, G.: Conditionals in Nonmonotonic Reasoning and Belief Revision. LNCS (LNAI), vol. 2087. Springer, Heidelberg (2001)
15. Kern-Isberner, G.: A thorough axiomatization of a principle of conditional preservation in belief revision. Annals of Mathematics and Artificial Intelligence 40(1-2), 127–164 (2004)
16. Koch, N.: Repräsentation und Verarbeitung konditionalen Wissens mit AsmL. Thesis, Master of Computer Science, FernUniversität in Hagen (2007) (in German)
17. Lang, J.: Belief update revisited. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, IJCAI 2007, pp. 2517–2522 (2007)
18. Paris, J.B., Vencovska, A.: In defence of the maximum entropy inference process. International Journal of Approximate Reasoning 17(1), 77–103 (1997)
19. Spohn, W.: Ordinal conditional functions: a dynamic theory of epistemic states. In: Harper, W.L., Skyrms, B. (eds.) Causation in Decision, Belief Change, and Statistics, II, pp. 105–134. Kluwer Academic Publishers, Dordrecht (1988)

Direct Support for Model Checking Abstract State Machines by Utilizing Simulation

Jörg Beckers, Daniel Klünder, Stefan Kowalewski, and Bastian Schlich

Embedded Software Laboratory, RWTH Aachen,
Ahornstr. 55, 52074 Aachen, Germany
{beckers,kluender,kowalewski,schlich}@cs.rwth-aachen.de
http://www-i11.informatik.rwth-aachen.de

Abstract. This paper presents an approach to model checking abstract state machines (ASMs) without the need for translation of the ASM specification into the modeling language of an existing model checker. Instead, our model checker [MC]SQUARE uses the simulation capabilities of COREASM to build the state space, thereby directly supporting ASMs and circumventing a possible loss of expressiveness in a translation process. This enables our approach to present counterexamples and witnesses directly as sequences of ASM states and at the same time supports the major features of COREASM like distributed ASMs, n-ary functions or extended rule forms. We show the applicability of this approach in a case study that also reveals possible improvements desirable for minimizing the duration needed for building the state space and its memory consumption.

1 Introduction

The ever increasing presence of information technology along with the pervasion of hardware and software into everyday life boosts the need for engineering methods and tools for the development of high quality systems. This trend is further amplified by the rising complexity of such systems and their usage for safety critical tasks. However, full or excessive testing is often not possible because of time or budget constraints of the specific product.

It has long been recognized that formal methods can help to improve software quality by providing means for finding and fixing errors as well as ambiguities. Of special interest for this paper is the possibility to automatically prove certain system properties: model checking is a formal method used for the automatic verification of systems. It uses an exhaustive search over all reachable system states to check whether the system (model) satisfies a given property (specification). Among other things, the abstraction capabilities of abstract state machines (ASMs) allow the verification to take place early in the design process, thereby minimizing the costs introduced by fixing found errors. Furthermore, ASMS have been shown to be an effective formal method for specification in an industrial context as e. g. shown by Börger et al. [1].

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 112–124, 2008.

In this paper we present a novel approach to automatically verifying dynamic properties for given ASMs that takes advantage of the fact that ASMs are not logical formulae, but machines coming with a notion of run [2]. We utilize the simulation capabilities of CoreASM [3], adapting it to branch into all possible successor states instead of choosing a random successor when faced with scheduling of distributed ASMs or nondeterministic choose. The so-built state space can be model checked by querying the boolean values of the verification formula's atomic propositions from CoreASM.

The rest of this paper is structured as follows: the next section summarizes related work, emphasizing the differences to the approach presented here. Section 3 gives an introduction to the model checker [mc]square focusing on its architecture which is the main reason for its applicability to the verification of CoreASM models. The interaction of [mc]square and CoreASM as well as the slight modifications to the latter are detailed in section 4 and evaluated in a case study in section 5. Finally, section 6 concludes the paper and gives an outlook to future work and possible improvements.

2 Related Work

There are several other publications on model checking of ASMs. Del Castillo and Winter translate specifications written in the ASM Workbench [4] into the input language of the SMV model checker [5,6]. Gargantini and Riccobenne produce Promela specifications for the Spin model checker from the AsmGofer tool [7]. Bounded model checking of ASMs using Answer Set Programming is presented by Tang and Ternovska [8]. An approach to model checking of AsmL is proposed by Kardos [9]. More recently, Ouimet and Lundqvist showed the verification of TASM models using UPPAAL by including information on time into ASMs [10]. Farahbod et al. support the same ASM constructs as our approach by translating CoreASM specifications into Promela and using Spin for model checking [11,12].

The approach presented in this paper differs from the ones above by not translating a given ASM into some other language but rather using their simulation in CoreASM to produce all possible states which are afterwards serialized and checked in [mc]square [13]. Thereby it directly supports ASMs, avoiding possible losses of expressiveness in a translation process. The main advantages of this approach are the ability to present counterexamples and witnesses of a verification as a sequence of ASM states and its direct support for the modeling and simulation features of CoreASM including n-ary functions and extended rule forms. Specifically, it also directly supports distributed ASMs and can be naturally extended to new features of CoreASM. On the other hand, it suffers from the representation of states in the data structures of CoreASM that were built with the goals of comprehensiveness and extendibility in mind and hence are not optimized for performance in a model checker.

3 [mc]square

[MC]SQUARE stands for Model Checking MicroController and is a discrete CTL model checker that was built for verifying microcontroller assembly code of embedded systems [13,14]. It supports several different microcontrollers and uses a local CTL algorithm first introduced by Vergauwen and Lewi [15] and later adapted by Heljanko [16]. Input models are accepted as assembly programs in Executable and Linking Format (ELF) and checked versus a specification that may contain propositions about registers, I/O registers, and variables. Additionally, [MC]SQUARE checks for stack collisions, stack overflows, and non-intended use of microcontroller features such as write access to reserved registers.

For this purpose, [MC]SQUARE utilizes simulators of the supported microcontrollers for state space building as well as for retrieving information about atomic propositions. The simulator is asked to return either a serialized representation of all possible successors given a starting state or the boolean values of certain atomic propositions in that state which are part of the specification formula. For lowering the state space size, serialized states are compressed and several abstraction techniques are used. While the abstractions are not used in the current implementation for model checking CoreASM files, some of them could provide notable benefits in the future. Furthermore, for storing the created state space [MC]SQUARE can transparently use either heap or hard disc memory.

Since model checking assembly code is inevitably hardware dependent, one of the main design goals of [MC]SQUARE was easy extendibility for new microcontrollers and corresponding assembly instructions. As CoreASM offers simulation capabilities for ASMs we plugged it into [MC]SQUARE as to represent a new, virtual microcontroller. The next subsections give a short summary of [17], detailing the important facts about the model checking process and the architecture of [MC]SQUARE which enable this layout.

3.1 Model Checking in [ms]square

Fig. 1 illustrates the model checking process: the user provides a CTL formula and a program (ELF file) which can be accompanied by the corresponding C file. First, the formula is parsed and the ELF file is disassembled into an assembly program. Afterwards, both are passed to the model checker which queries the initial state from the state space and begins to check the formula.

As the model checker uses an on-the-fly algorithm, it asks the state space for successor states whenever it needs them to check the current logical formula or one of its subformulae. If the state space has not yet created the needed successors of a state, it uses the simulator to create them on demand. The simulator creates successors by simulating the effects of the current instruction on the current state. If the model checker finds a state that does not satisfy the formula, the algorithm aborts.

After model checking has been conducted the counterexample generator is invoked and presents the counterexample or witness in the assembly code, in the C code (if present), and as a state space graph.

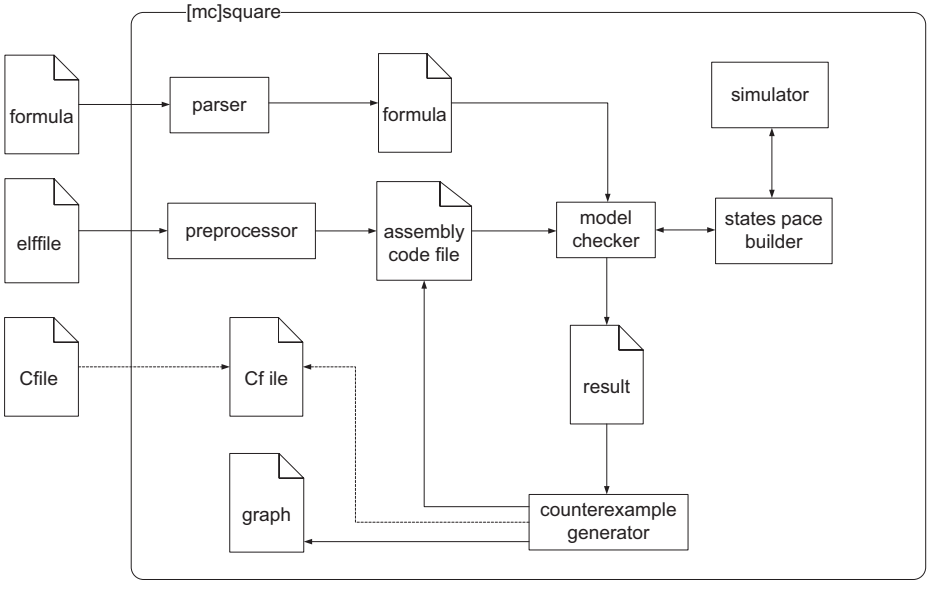


Fig. 1. Model checking process in [MC]SQUARE used for the verification of microcontroller code

3.2 Architecture

Fig. 2 depicts the architecture of [MC]SQUARE as a UML class diagram. One of the main quality requirements motivating this design was the ease of extending the tool to support new microcontrollers. The core classes ModelChecker, StateSpace, and Simulator correspond to the equivalent components shown in Fig. 1.

An object of class ModelChecker consists of the user provided Formula and a StateSpace which in the beginning is filled just with the initial state provided by the HardwareSimulator. To keep the state space hardware-independent, all hardware-dependent information is stored in a byte array that is used by the simulator to generate the real hardware state (processor state, contents of memory, etc.). All other information stored in the states is hardware-independent (e.g. name, successors, and truth values of the subformulae).

A Simulator combines a microcontroller Program and a Hardware device that are both implemented hardware-dependent. They represent the assembly instructions and the hardware features, like e.g memory and registers, respectively. The effect of a single instruction on the hardware is simulated and returned to the state space as an array of byte arrays each representing one possible successor state. The simulator also handles the non-determinism that is involved, e. g. whenever inputs from the environment are read, by creating an over-approximation of the real state space.

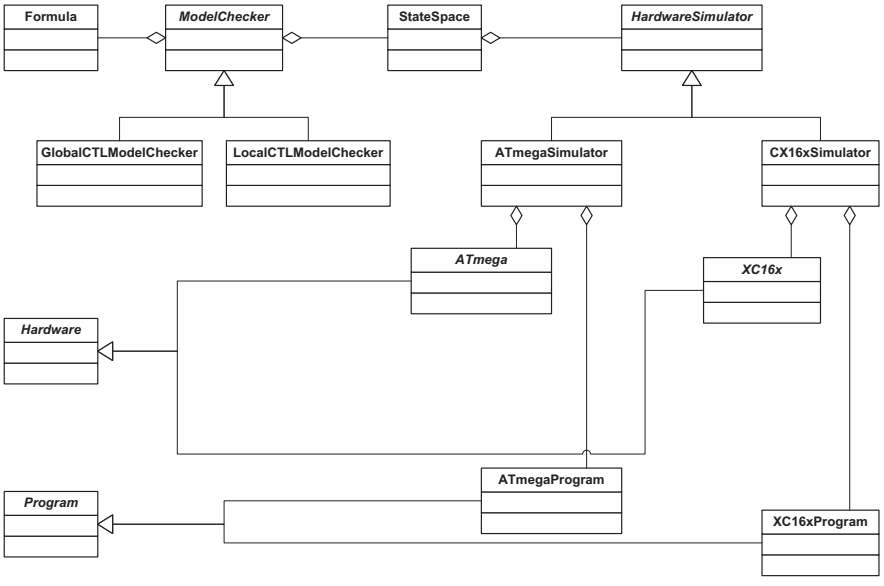


Fig. 2. Architecture of [MC]SQUARE represented by a UML class diagram. The figure only shows classes that are important to understand the general idea. Not all subclasses are shown and some packages as e. g. GUI and utilities are omitted completely.

The changes that are needed to add new microcontrollers to [MC]SQUARE vary depending on the microcontroller family. Currently the ATmega family as well as the Infineon XC167 microcontroller are supported.

4 Verification of CoreASM Models Using [mc]square

We utilize COREASM to introduce a new HardwareSimulator (c. f. Fig. 2) to [MC]SQUARE, virtually representing an ASM microcontroller. A simulator encapsulates both a program and the hardware which are replaced by an ASM specification and the abstract machine it runs on, respectively. Hence, none of the implementations of Formula, ModelChecker, or StateSpace have to be modified. The following paragraphs describe the necessary modifications of COREASM and the glue code to connect it to the state space builder of [MC]SQUARE.

Serialization. As mentioned above, all hardware-dependent information of a state, i. e. in the case of COREASM all the universes, functions and rules of an ASM state, has to be represented as an array of bytes by the simulator. In COREASM this information is stored in a so called AbstractStorage that was designed to be comprehensive and easily extendible via COREASM's plugin architecture.

We have chosen to transform a COREASM state into a byte array and vice versa using the native Java support for serialization and deserialization of objects. This technique is typically used for object persistence and does not demand

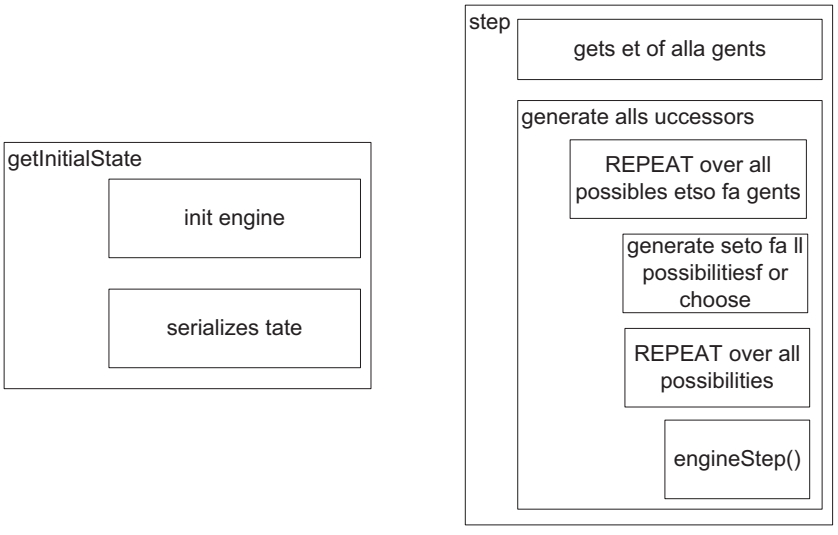


Fig. 3. Finding all successors of an ASM state with CoreASM for the initial state and all successors

many modifications to CoreASM's source code and facilitates the extension to future plugins of CoreASM. On the other hand this method produces rather big memory footprints and is one of the major areas for improvements in future work.

Stepping. The most challenging modifications to CoreASM are due to the fact that not only one but all possible successors of a state have to be produced by a step of the CoreASM engine. This affects especially the scheduler that determines the set of agents executing concurrently in a distributed ASM and the choose plugin that introduces nondeterminism into the model. Nevertheless, the implementation was eased by CoreASM's thoughtful architecture. In particular the possibility of extension via plugins allowed us to replace only those two parts of the engine with our own versions.

While the original plugins randomly select a set of agents or one of a set of values respectively we have to gather all possible sets of agents and all possible values and execute each combination of the two. For each combination the original state has to be recovered and the random selection has to be replaced by the selected combination while at the same time paying attention to the consistency of the update set. As is depicted in Fig. 3 this is done using an adopted version of the CoreASM engine step.

Equality of States. One unexpected problem with the serialization described above is the equality of states. A CoreASM state is internally constructed from several orderless mappings. This can lead to a situation where apparently equal states are considered to be different by [mc]square e. g. just because two

functions have switched places in the byte representation of the state. Therefore we introduce an order relation for all state elements that is used to sort all mappings right before serialization takes place.

Check Atomics. Finally, the CoreASM simulator in [mc]square has to evaluate atomic propositions on the elements of a state given in its byte representation. Such a state is therefore deserialized into its object representation in CoreASM. Afterwards, all the elements of the state, its universes, rules, and functions can be queried. This means that verification formulas can contain propositions about all these elements including, e. g. the activity of a certain agent, the values of certain locations, or the existence of certain elements in a universe.

5 Case Study

In this section we analyze the applicability of our approach and compare its results to those achieved with the other tools mentioned in section 2. Therefore, the following subsections describe the performance reached when using [mc]-square on the well-known algorithm for distributed termination detection by Dijkstra et al. [18] as well as on the Flash cache coherence protocoll [19].

5.1 Distributed Termination Detection

The algorithm is to detect the termination of a distributed program running on a network of computation nodes. Each node can be either in active or in passive state. Only nodes in active state can send messages to other nodes to transfer parts of their computation. After having received a message, a passive node switches to active. The transition from active to passive state occurs when a node finishes its computation. The state in which all nodes are passive and no messages are on their way is the state which is to be detected by the algorithm.

Eschbach [20] gives the ASM specification of such an algorithm by Dijkstra et al. [18] and manually proves some of its properties. A CoreASM version of this algorithm is used by Ma [12] to analyze the translation of CoreASM to Promela which limits the maximum number of messages each node may send. In this case study we use Ma's specification to verify the following CTL property:

$$\text{AG } (\text{termination}(.)=\text{true} \Rightarrow \text{AF terminationDetected}(.)=\text{true})$$

It states that each termination will eventually be detected. Table 1 shows the results when model checking this property for two nodes with a maximum of one message per node. Compared, the translation from CoreASM to Promela takes only one second [12].

Obviously, the runtime is clearly above those reached by earlier approaches. Nevertheless, our approach shows its applicability and general functioning. The number of states stored and the memory consumption thereof are remarkably small compared to other model checking problems which is due to the level of abstraction offered by ASMs.

Table 1. Verifying termination detection with [MC]SQUARE

number of states stored	89208
number of transitions	430787
number of states created	440952
runtime	15 hours. 14 minutes
memory consumption	261857 KB

5.2 Flash Cache Coherence Protocol

The Stanford FLASH multiprocessor is an architecture for distributed processors that share their memory [19]. This sharing is coordinated by the FLASH cache coherence protocol which was used by Winter [5] as well as Ma [12] to demonstrate the capabilities of their aforementioned model checkers for ASMs. The memory is distributed over all nodes of the processor network and divided into lines which are associated to nodes. Each processor can have local copies of each file and is assured by the coherence protocol that all files are up-to-date.

We use the COREASM specification of this coherence protocol given by Ma [12] to check the following two properties for a setup of two processors and one memory line:

– safety (S):

AG !(CCState(a1,l1)=exclusive & CCState(a2,l1)=exclusive)

– liveness (L):

AG AF (CurPhase(a1,l1)=ready & CurPhase(a2,l1)=ready)

The safety condition expresses that the two nodes should never have access two one single line at the same time while the liveness condition ensures that all requests are processed at some time. Neither of these two properties is fullfilled by the protocol. Table 2 shows the runtime needed for verification of a corresponding ASM using the tools CoreASM2Promela and ASM2SMV as presented by Ma [12] and table 3 shows the corresponding results reached using [MC]SQUARE.

Although the test runs have been conducted on different machines due to the need for different operating systems it can be conducted that [MC]SQUARE reaches a runtime performance comparable to those of CoreASM2Promela and ASM2SMV. This is due to the fact that [MC]SQUARE uses an on-the-fly algorithm

Table 2. Runtime needed for verification of the FLASH cache coherence protocol using the tools CoreASM2Promela and ASM2SMV as presented by Ma [12]

Property	S	L
CoreASM2Promela [12]	43s	7s
ASM2SMV [5]	438s	921s

Table 3. Model checking of the FLASH cache coherence protocol using [MC]SQUARE

property	**S**	**L**
number of states stored	873	173
number of transitions	2240	247
number of states generated	3011	323
runtime	339s	35s
memory consumption	3051 KB	13 KB
counterexample		
number of states	91	7
number of transitions	140	7

which aborts state space building in case of an existing counterexample. Fig. 4 shows an extract from the graphical representation of the counterexample for the liveness condition. This feature distinguishes [MC]SQUARE from the other approaches as it shows the counterexample as an ASM state space.

5.3 Evaluation

The generation of all possible successor states by altering a step in COREASM works as described in section 4. State sizes depend on the ASM specification used and are mainly influenced by the efficiency of serialization. During our test runs we reached a maximum number of roughly 250.000 states stored in the main memory of our test machine with 16 Gbytes of RAM. This number can be further increased by using the hard disc for storing state spaces. But as before, the available RAM limits the state space size, because it stores an index to the state location on the hard disc. Since we use an on-the-fly model checking algorithm, the time for building the state space can not be distinguished from the time needed for verification.

The case studies show that our approach is applicable and supports the promised features. Specifically the handling of nondeterminism and scheduling of concurrent asynchronous ASMs was analyzed to work correctly. Furthermore, the study sets the direction for future work which should concentrate on a more efficient creation of COREASM states.

We observed relatively large states but surprisingly little problems with memory consumption. This is in part due to the abstraction capabilities offered by the ASM theory and on the other hand due to efficient state compression used in [MC]SQUARE. Nevertheless, the size of each single state is one of the areas for future improvement as discussed in section 6.

The main problem at present is the runtime performance of our approach. Since [MC]SQUARE shows competitive performance results when used for model checking assembly code we further analyzed its interaction with COREASM. When using different compression techniques for state space minimization, i. e. zip compression vs. run-length encoding vs. no compression, we don't observe any runtime differences compared to differences up to a factor of ten when verifying

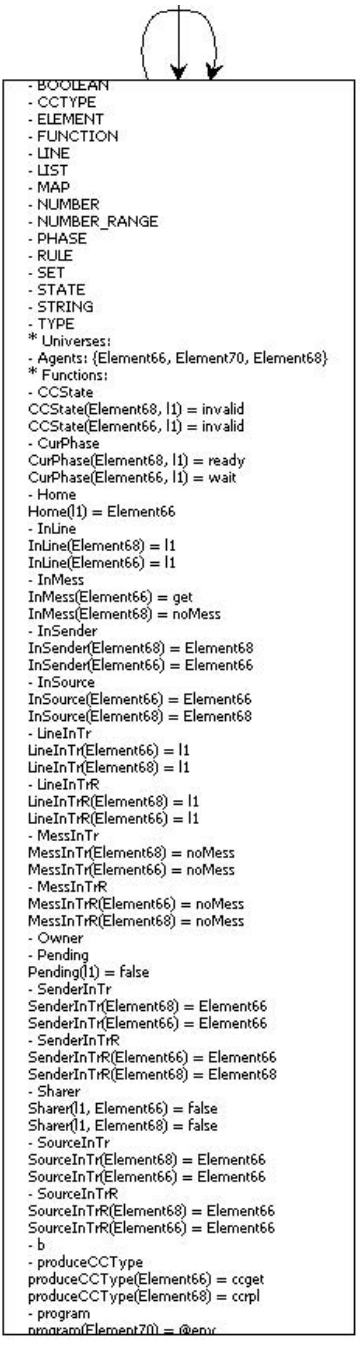


Fig. 4. Extract from the liveness counterexample

assembly code [13]. This indicates that the CPU idles during model checking and can hence dabble in compression. We believe that this is due to a high number of context switches while simulating a step using CoreASM. A state in CoreASM is highly object-oriented and was designed to be comprehensive and easily extendible. This leads to a very branched structure that is not optimal for usage in our scenario. Nevertheless, we still believe that our approach can reach competitive performance figures as a similar problem arose when we used an existing microcontroller simulator for model checking assembly code using [mc]square. Meanwhile our own optimized microcontroller simulator shows a runtime improvement up to a factor of 1000.

When verifying invalid formulas [mc]square shows the advantage of its on-the-fly algorithm which aborts state space building in case of an existing counterexample. Additionally this counterexample can be represented as a trace of the ASM state space, thereby eliminating the need to understand the mapping from a counterexample presented in a language other than ASM.

6 Conclusion and Future Work

This paper introduces an approach to model checking ASM specifications and is the first of its kind that offers direct support for ASMs. We have applied the discrete CTL model checker [mc]square that uses microcontroller simulation to verify hardware-dependent assembly code by adopting the CoreASM tool as a simulator for a virtual ASM microcontroller. Therefore, we have enhanced the normal simulation to include all possible successors of a state and serialized those states into a byte representation. In a case study on a distributed termination detection algorithm as well as on the Flash cache coherence protocol the approach has proven its applicability and shown beneficial directions for future improvements.

This work has benefited enormously from the excellent designs of the two combined tools: CoreASM and [mc]square. CoreASM's plugin architecture greatly supports its adaptability that enabled its usage in this context and [mc]-square has shown its extendibility way beyond its original purpose.

The most promising modifications for the future deal with the optimization of memory consumption and runtime performce needed for building the state space. The first step in this direction clearly is the improvement of the serialization e. g. by using a mapping between state elements, all of which are currently stored as Java Strings, and a smaller data type better adapted to its serialization. Furthermore, static analyses could help to minimize the number of states generated. Especially, delayed nondeterminism, a technique introduce by [mc]square for model checking microcontroller code, seems promising. Nondterministic locations split the state space at the very latest possibility, i. e. when they are first read after their creation. Finally, one can imagine support for symbolic model checking of ASMs, where regions of the state space are represented by first order formulas as suggested in [2]. Runtime optimization can mainly be reached by optimizing the time needed for simulation.

References

1. Börger, E., Päppinghaus, P., Schmid, J.: Report on a Practical Application of ASMs in Software Design. In: Gurevich, Y., Kutter, P., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 361–366. Springer, Heidelberg (2000)
2. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
3. Farahbod, R., Gervasi, V., Glässer, U.: Coreasm: an extensible asm execution engine. Fundamenta Informaticae 77, 71–103 (2007)
4. Castillo, G.D.: Towards comprehensive tool support for abstract state machines. In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 311–325. Springer, Heidelberg (1999)
5. Winter, K.: Model Checking Abstract State Machines. PhD thesis, Technical University of Berlin, Germany (2001)
6. Castillo, G.D., Winter, K.: Model checking support for the asm high-level language. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 331–346. Springer, Heidelberg (2000)
7. Gargantini, A., Riccobene, E.: ASM-based Testing: coverage criteria and automatic tests generation. In: Moreno-Díaz, R., Quesada-Arencibia, A. (eds.) Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Canary Islands, Spain, Universidad de Las Palmas de Gran Canaria, February 2001, pp. 262–265 (2001)
8. Tang, C.K.F., Ternovska, E.: Model checking abstract state machines with answer set programming. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 443–458. Springer, Heidelberg (2005)
9. Kardos, M.: An approach to model checking asml specifications. In: Proceedings of the 12th International Workshop on Abstract State Machines, pp. 289–304 (2005)
10. Ouimet, M., Lundqvist, K.: The timed abstract state machine language: Abstract state machines for real-time system engineering. In: Proceedings ASM 2007 (2007)
11. Farahbod, R., Glässer, U., Ma, G.: Model checking coreasm specifications. In: Proceedings ASM 2007 (2007)
12. Ma, G.Z.: Model checking support for coreasm: Model checking distributed abstract state machines using spin. Master's thesis, Simon Fraser University, Burnaby, Canada (2007)
13. Schlich, B., Kowalewski, S.: [MC]SQUARE: A model checker for microcontroller code. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2nd Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation (IEEE-ISoLA). IEEE Computer Society, Los Alamitos (to appear, 2006)
14. Schlich, B., Salewski, F., Kowalewski, S.: Applying model checking to an automotive microcontroller application. In: Proc. IEEE 2nd Int. Symp. Industrial Embedded Systems (SIES), pp. 209–216. IEEE, Los Alamitos (2007)
15. Vergauwen, B., Lewi, J.: A linear local model checking algorithm for ctl. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 447–461. Springer, Heidelberg (1993)
16. Heljanko, K.: Model checking the branching time temporal logic ctl. Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland (May 1997)
17. Schlich, B., Kowalewski, S.: An extendable architecture for model checking hardware-specific automotive microcontroller code. In: Schnieder, E., Tarnai, G. (eds.) Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT), GZVB, Braunschweig, Germany, pp. 201–212 (2007)

18. Dijkstra, E., Feijen, W., Gasteren, A.: Derivation of a Termination Detection Algorithm for Distributed Computations. Information Processing Letters 16(5), 217–219 (1983)
19. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., et al.: The Stanford FLASH multiprocessor. In: Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., et al. (eds.) Proceedings the 21st Annual International Symposium on Computer Architecture, pp. 302–313 (1994)
20. Eschbach, R.: A Termination Detection Algorithm: Specification and Verification. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1720–1737. Springer, Heidelberg (1999)

On the Purpose of Event-B Proof Obligations*

Stefan Hallerstede

University of Southampton
United Kingdom
sth@ecs.soton.ac.uk

Abstract. Event-B is a formal modelling method which is claimed to be suitable for diverse modelling domains, such as reactive systems and sequential program development. This claim hinges on the fact that any particular model has an appropriate semantics. In Event-B this semantics is provided implicitly by proof obligations associated with a model. There is no fixed semantics though. In this article we argue that this approach is beneficial to modelling because we can use similar proof obligations across a variety of modelling domains. By way of two examples we show how similar proof obligations are linked to different semantics. A small set of proof obligations is thus suitable for a whole range of modelling problems in diverse modelling domains.

1 Introduction

Event-B [5] is a formal modelling method for discrete systems based on refinement [8,9,10]. The main purpose of creating models in Event-B is to reason about them and understand them. Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. We insist that reasoning is an essential part of modelling because it is the key to understanding complex models. When we create a complex model, usually, our understanding of it is incomplete at first; and the first duty of a modelling method is to help improve our understanding of the model.

To reason about a model we consider its proof obligations. Proof obligations have a two-fold purpose. On the one hand, they show that a model is sound with respect to some behavioural semantics. On the other hand, they serve to verify properties of the model. This goes so far that we only focus on the proof obligations and do not present a behavioural semantics at all. This approach permits us to use the same proof obligations for very different modelling domains, for instance: reactive, distributed and concurrent systems [7], a probabilistic variant [16]; sequential programs [4]; or electronic circuits [15]. All of this, without being constrained to semantics tailored to a particular domain. Event-B is a calculus for modelling that is independent of the various models of computation.

In this article we present two examples of Event-B semantics showing the viability of this approach. For this purpose, we introduce enabledness proof obligations into the

* This research was carried out as part of the EU research project DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) http://www.deploy-project.eu/

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 125–138, 2008.

Event-B method. We go on to show how they are incorporated into relative deadlock-freeness proofs with respect to the failures model of CSP [17] and into soundness proofs of sequential program development [4]. The theoretical results as such are not new. In [9] a temporal *leadsto*-operator and deadlock-freeness are introduced, where the *leadsto*-operator is modelled by means of a while-loop. In this article we discuss the use the same (few) proof obligations to reason about different semantic models. We present the derivation of the proof obligations from the semantic models to demonstrate what is involved. The theory used in this article is more based on [11,21] than on [1,3]; the latter are geared towards sequential program development.

A complication arises (by our choice), because the first semantics uses a relational model [18] and the second set transformers [11,13]. This complication is hidden in Event-B by means of its proof obligations: to the user of Event-B it all looks the same. Simple restrictions on proof obligations achieve soundness in either case. Because Event-B models do not have a (behavioural) semantics a priori, we are free to chose one and with it a set of appropriate proof obligations. If we were to fix some semantics for Event-B, we would have difficulties applying it to the various domains mentioned in the introduction.

Outline. Section 2 presents Event-B in terms of its proof obligations. In Sections 3 and 4 we relate a reactive systems semantics and a sequential program semantics to proof obligations presented in Section 2. Sections 3 and 4 are somewhat technical. We have chosen to present the material in this way to demonstrate how enabledness proof obligations arise in the two cases. As a consequence of this decision there is no space to present more examples. It is not our intention to present a complete list of semantics for Event-B. That list is open-ended. In future, new applications of Event-B may emerge that require new kinds of semantics. In the same sense, the two examples presented are not intended be understood as fully representing the corresponding domains, reactive systems modelling and sequential program modelling. The two seem reasonable based on our experience. They could be adapted to fit particular modelling needs and development processes. Whenever we want to use Event-B with some specific semantics we can prove how Event-B suits that semantics.

2 Event-B

We present the core of Event-B in terms of its proof obligations concerned with refinement and consistency. For the purposes of this article the proof obligations are only stated as set-theoretic expressions. In order to make them easier to digest we introduce some rudimentary notation of Event-B and define all employed sets and relations based on the notation.

Behavioural aspects of Event-B models are expressed by means of *machines*. A machine M may contain *variables*, *invariants*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. (Variables occurring free in a formula are indicated in parentheses.) Possible state changes are described by means of events E_m, for $m \in \alpha M$. (In the following sections it will prove useful to have events associated with indices drawn from finite sets αM. We introduce them here to

achieve a more coherent presentation.) Each event E_m is composed of a *guard* $G_m(v)$ and an *action* $v :| S_m(v, v')$. We denote an event E_m by

$$\mathsf{when}\ G_m(v)\ \mathsf{then}\ v :| S_m(v, v')\ \mathsf{end}.$$

A dedicated event that has true as its guard and $v :| A(v')$ as its action is used for *initialisation*. (The predicate $A(v')$ does not refer to unprimed variables.)

The action $v :| S_m(v, v')$ describes the relationship between the state just before the action has occurred (represented by unprimed variable names v) and the state just after the action has occurred (represented by primed variable names v').

The guard of an event states the necessary condition under which the event may occur, and the action describes how the state variables evolve when the event occurs.

In order to simplify the main part of this article, we do not present local variables of events here. For the same reason we state actions as single nondeterministic assignments $v :| S_m(v, v')$. For a detailed description of events, actions and assignments see [8].

We assume familiarity with basic set-theoretic notation defining sets and relations corresponding to all of the above:

$$\begin{array}{rcl} \Phi & \widehat{=} & \{v \mid \top\}^{1} \\ i & \widehat{=} & \{v \mid I(v)\} \\ g_m & \widehat{=} & \{v \mid G_m(v)\} \\ s_m & \widehat{=} & \{v \mapsto v' \mid S_m(v, v')\} \\ a & \widehat{=} & \{v' \mid A(v')\}, \end{array}$$

where Φ denotes the entire state space.

2.1 Machine Consistency

For each event E_m of a machine M, *feasibility* must be proved:

$$i \cap g_m \quad \subseteq \quad s_m^{-1}[\Phi]. \tag{1}$$

By proving feasibility, we ensure that S_m provides an after state whenever G_m holds. This means that the guard indeed represents the enabling condition of the event.

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called *invariant preservation*:

$$(g_m \lhd s_m)[i] \quad \subseteq \quad i.^{2} \tag{2}$$

Similar proof obligations are associated with the initialisation event of a machine: feasibility of initialisation is $a \neq \varnothing$ and invariant establishment is $a \subseteq i$.

[1] Φ is the Cartesian product of the types $\Delta_1, \Delta_2, \ldots, \Delta_\varkappa$ of the variables $v_1, v_2, \ldots, v_\varkappa$. Writing $\{v \mid \top\}$ we avoid introducing the component types $\Delta_1, \Delta_2, \ldots, \Delta_\varkappa$.

[2] $\lhd$ denotes *domain restriction*: $x \mapsto y \in (g \lhd s) \ \equiv \ x \in g \ \wedge \ x \mapsto y \in S$.

2.2 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model [8]. For more on the well-known theory of refinement, we refer to the Action System formalism [10] that has inspired the development of Event-B.

A machine N can refine at most one other machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$, where v are the variables of the abstract machine and w the variables of the concrete machine.

Let E_m, for $m \in \alpha M$, be the abstract events; and let F_n, for $n \in \alpha N$, with αN a finite set and $\alpha M \subseteq \alpha N$, be the concrete events of the form:

$$\mathsf{when}\ H_n(w)\ \mathsf{then}\ w :|\ T_n(w, w')\ \mathsf{end};$$

and let $w :|\ B(w')$ be the action of the initialisation.

The corresponding set-theoretic definitions are:

$$\begin{array}{rcl} \Psi & \widehat{=} & \{w \mid \top\} \\ k & \widehat{=} & \{v \mapsto w \mid I(v) \wedge J(v, w)\} \\ j & \widehat{=} & \{v \mapsto w \mid J(v, w)\} \\ h_n & \widehat{=} & \{w \mid H_n(w) \\ t_n & \widehat{=} & \{w \mapsto w' \mid T_n(w, w')\} \\ b & \widehat{=} & \{w' \mid B(w')\}. \end{array}$$

Each event E_m of the abstract machine is *refined* by a concrete event F_m. Somewhat simplified, we can say that F_m refines E_m if the guard of F_m is stronger than the guard of E_m, and the gluing invariant $J(v, w)$ establishes a simulation of F_m by E_m:

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;j.\,^{3} \tag{3}$$

Using (2) we can infer from (3)

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;k. \tag{4}$$

In the course of refinement, *new events* can be introduced into a model. New events must be proved to refine the implicit abstract event $skip$ that does nothing; that is, its guard is true and its action is $v :|\ v' = v$. In the notation used in this article new events are just those with indices drawn from the set $\alpha N \setminus \alpha M$.

Convergence. Moreover, it may be proved that new events do not collectively diverge by means of a well-founded relation r. We refer to the corresponding proof obligation as *progress*:

$$k\,;(h_n \lhd t_n) \quad \subseteq \quad k\,;r. \tag{5}$$

A common choice for r is $\eta^{-1}\,;\{x \mapsto y \mid x < y\}\,;\eta$ where $\eta = (\lambda\, w \cdot w \in \mathbb{N} \mid V(w))$ and $V(w)$ an integer expression, called *variant*, of N. We call events that satisfy (5) *convergent*.

[3] The corresponding proof obligation for the initialisation is: $b \subseteq j[a]$.

Enabledness. Using (1) we infer from (3),

$$k \rhd h_m \quad \subseteq \quad g_m \lhd k, \tag{6}$$

the guard of the abstract event may be strengthened during refinement. As a consequence, it is sufficient if the guard of the concrete event is false, that is, $h_m = \varnothing$. This means we could refine any abstract event by a concrete event with false as its guard. Such an event can never occur. If we strengthen the guard less extremely, we still have a concrete event that may occur less often than its abstract counterpart. If this is not intended we need also to weaken the guard as discussed in the next paragraph.

Let $m \in \alpha M$ and $L \subseteq \alpha N$. We may prove that whenever the abstract machine may continue by means of event E_m with guard G_m then the concrete machine may continue by means of some F_ℓ for some $\ell \in L$:

$$k[g_m] \quad \subseteq \quad (\bigcup \ell \cdot \ell \in L \mid h_\ell) \quad . \tag{7}$$

By convention we assume that the guard h_m of the concrete event that refines E_m is contained in the union on the right hand side, that is, $m \in L$. If $L = \{m\}$, then combining (6) and (7) yields the equivalence of abstract guards to concrete guards under the (gluing) invariant:

$$g_m \lhd k \quad = \quad k \rhd h_m.$$

If L contains a new event, the relationship gets more complicated; enabledness and convergence interact. This is becomes apparent in our presentation of sequential programs later. In our presentation of reactive systems below this is less visible due to some simplifications that we have made to keep it brief.

3 Reactive Systems Modelling

We base our presentation of reactive systems modelling on the semantics of the process algebra CSP [17,22]. CSP was developed specifically for modelling of such systems [18]. Its semantics is expressed in terms of finite and infinite traces, failures, and divergences describing the behaviour of a system. We focus on failures: failures refinement guarantees that we cannot introduce new deadlocks in a refined model. In Event-B this is achieved by enabledness (7). In this section we show how failures and enabledness are connected. The principle of this connection is not new [12,19]. For this reason, we only present the essential formal ingredients and proofs. We assume that the machines are free of divergences, proved by means of (5), and that all events are image-finite, that is, $\mathsf{finite}(s_m[g_m])$. As a consequence, the behaviour of machines can be described purely in terms of failures, the component most relevant to our analysis of enabledness proof obligations.

3.1 Failure Semantics

We define failures directly in the set-theoretic notation of Section 2; similarly to [14]. Let M be a machine with initialisation a and events with guards g_m and actions s_m.

For machine M and a sequence of event indices t we define the path of t by

$$\begin{array}{rcl} \mathsf{path}_M(\langle\rangle) & \widehat{=} & a \lhd \mathsf{id}_\Phi \\ \mathsf{path}_M(t^\frown\langle m\rangle) & \widehat{=} & \mathsf{path}_M(t)\,;(g_m \lhd s_m). \end{array}$$

A path describes the state transition corresponding to the occurrence of the t. If the path of t is not empty, then t belongs to the behaviour of M; we say such a t is a trace of M. Failures are defined in terms of paths and of refusals introduced next. Being in a state satisfying some refusal R, none of the events indexed by R can occur,

$$\mathsf{refusal}_M(R) \quad \widehat{=} \quad (\bigcap m \cdot m \in R \mid \Phi\backslash g_m) \quad .$$

Failures are traces combined with refusals; the pair $(t \mapsto R)$ is a failure of M if t is a trace of M and after having engaged in t machine M may be in a state where all events indexed by R are refused,

$$(t \mapsto R) \in \mathsf{failure}_M \quad \widehat{=} \quad \mathsf{path}_M(t) \rhd \mathsf{refusal}_M(R) \neq \varnothing$$

Failure semantics does not deal with fairness.

3.2 Failure Refinement

Let $C = \alpha N\backslash\alpha M$ be the indices of all new events, and for a trace t and a set of event names L let $t{\uparrow}L$ be t with all event names in L removed. We say machine N failure-refines machine M,

$$(t \mapsto R \cup C) \in \mathsf{failure}_N \quad \Rightarrow \quad (t{\uparrow}C \mapsto R) \in \mathsf{failure}_M,$$

if the failures of N are contained in the failures of M modulo the new events C. Note, that this definition of failure refinement is not standard. We have combined the plain refinement notion of [17] with hiding of new events in order to shorten the presentation. The given refinement notion is still monotonic because hiding is monotonic. We do not suggest that this is the notion of failures refinement one should be using in practice but believe that it is sufficient to make our point about using Event-B for failure refinement of machines. A variant of it has been used to model introduction of local channels in stated based reactive models [12].

Failure-refinement is proved by relating traces and failures of the two machines [12]. Assume, by means of (4), we have

$$\mathsf{path}_N(t) \quad \subseteq \quad \mathsf{path}_M(t{\uparrow}C)\,;k. \tag{8}$$

We observe

$$\begin{array}{rcll} & (t \mapsto R \cup C) \in \mathsf{failure}_N & & \{\ \text{def. of failure}\ \} \\ \equiv & \mathsf{path}_N(t) \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & & \{\ \text{by (8)}\ \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C)\,;k \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & & \{\ \text{set theory}\ \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd k^{-1}[\mathsf{refusal}_N(R \cup C)] \neq \varnothing & & \{\ \text{see (9) below}\ \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd \mathsf{refusal}_M(R) \neq \varnothing & & \{\ \text{def. of failure}\ \} \\ \equiv & (t{\uparrow}C \mapsto R) \in \mathsf{failure}_M. & & \end{array}$$

that N failure-refines M, provided

$$k^{-1}[\mathsf{refusal}_N(R \cup C)] \quad \subseteq \quad \mathsf{refusal}_M(R) \tag{9}$$

holds. We observe:

$$\begin{array}{lll}
& k^{-1}[\mathsf{refusal}_N(R \cup C)] \subseteq \mathsf{refusal}_M(R) & \{ \text{ def. refusal } \} \\
\equiv & k^{-1}[(\bigcap n \cdot n \in (R \cup C) \mid \Psi \backslash h_n)] \subseteq (\bigcap m \cdot m \in R \mid \Phi \backslash g_m) & \{ \text{ set theory } \} \\
\equiv & k[(\bigcup m \cdot m \in R \mid g_m)] \subseteq (\bigcup n \cdot n \in (R \cup C) \mid h_n) & \{ \text{ set theory } \} \\
\equiv & (\bigcup m \cdot m \in R \mid k[g_m]) \subseteq (\bigcup n \cdot n \in (R \cup C) \mid h_n) & \{ \text{ set theory } \} \\
\equiv & \forall m \cdot m \in R \Rightarrow (k[g_m] \subseteq (\bigcup n \cdot n \in (R \cup C) \mid h_n)) & \{ \text{ set theory } \} \\
\Leftarrow & \forall m \cdot m \in R \Rightarrow (k[g_m] \subseteq (\bigcup n \cdot n \in (\{m\} \cup C) \mid h_n)) \quad . &
\end{array}$$

Refusals are downward closed: if R is a refusal and $m \in R$ then $\{m\}$ is a refusal too. Hence, the strengthening $(\bigcup b \in (R \cup C) \cdot \ldots)$ to $(\bigcup b \in (C \cup \{a\}) \cdot \ldots)$ in the last step is not as severe as it may seem. The formula

$$k[g_m] \quad \subseteq \quad (\bigcup \ell \cdot \ell \in (\{m\} \cup C) \mid h_\ell)$$

in the last step of the calculation is just proof obligation (7) with $L = \{m\} \cup C$.

When we model reactive systems in Event-B, we do not need to be aware of the failures model. The proof obligations form a barrier that shields from the details and complications of the semantic model.

Given the description of Event-B in the introduction it is tempting to interpret Event-B always in the way presented in this section. After all, Event-B is a descendant of Action Systems and has been conceived to model systems. However, the semantics of Event-B is not fixed. We can think about any Event-B machine in terms of any appropriate semantics. In the next section we discuss Event-B for sequential program development — with different semantics but with similar proof obligations to those of this section.

4 Sequential Program Modelling

Event-B has been used for sequential program development [4]. We present a soundness argument resulting from the "defect" of Event-B not to provide preconditions for events: events are guarded and block execution when the guard is false. In sequential program refinement preconditions are more common because they lead certainly to implementable programs. This does not hold for guards. If we were to interpret event guards as preconditions the problem would disappear. (In fact, this interpretation is customary in Z [23,24].) We need an additional proof obligation to rectify this.

Given the problem described above: Why does Event-B not support preconditions and guards? By contrast, this is supported by the B Method [1] but leads to more intricate (and sometimes obscure) proof obligations. In Event-B simplicity of the proof obligations is considered of major importance. It brings two strongly related benefits: proof obligations are easy to understand, and more efficient and comprehensive tool support is possible.

In this section we present how enabledness proof obligations arise when proving loop introduction correct in Event-B. We first present some set transformer theory. In the remainder of this section we prove loop introduction correct with respect to (forward) refinement of set transformers. The enabledness proof obligation will only appear at the very end of the proof.

4.1 Set Transformers

The notions introduced in this section are intended to capture semantical properties of sequential programs. This should not be confounded with the actual Event-B notation that uses first-order predicate logic and set theory presented in Section 2. The model of set transformers we use follows closely the type-theoretical model of [11][4]. However, instead of type theory we use set theory which is easier to relate to Event-B; see also [21]. State spaces are Cartesian products denoted by the letters Φ and Ψ as introduced in Section 2.

Set transformers[5] are functions from sets to sets. Let g and φ be subsets of V and s a relation. In this article we make use of the following set transformers[6]:

$$\begin{array}{lcll} \lfloor g \rfloor(\varphi) & \widehat{=} & g \cap \varphi & (\textit{assertion}) \\ \lceil g \rceil(\varphi) & \widehat{=} & (\Phi \setminus g) \cup \varphi & (\textit{assumption}) \\ \lceil s \rceil(\varphi) & \widehat{=} & \{v \mid s[\{v\}] \subseteq \varphi\} \quad . & (\textit{demonic update}) \end{array}$$

For set transformers P we define precondition $\mathsf{pre}(P)$ and guard $\mathsf{grd}(P)$ by

$$\begin{array}{lcl} \mathsf{pre}(P) & \widehat{=} & P(\Phi) \\ \mathsf{grd}(P) & \widehat{=} & \Phi \setminus P(\varnothing). \end{array}$$

Note, that (1) implies $i \cap \mathsf{grd}(\lceil g_m \rceil ; \lceil s_m \rceil) = i \cap g_m$ and $i \cap \mathsf{pre}(\lfloor g_m \rfloor ; \lceil s_m \rceil) = i \cap g_m$. The informal description of the meaning of a guard in the beginning of Section 2 leaves us a choice for its interpretation. It can be read as an assertion or an assumption. The standard reading of Event-B is as an assertion, that is, event E_m corresponds to the set transformer

$$\lceil g_m \rceil ; \lceil s_m \rceil. \tag{10}$$

Based on set transformers, sequential programs are usually specified in terms of *specification statements*[11,20], namely,

$$\lfloor g_m \rfloor ; \lceil s_m \rceil, \tag{11}$$

[4] Our presentation is based on first-order set theory instead of higher-order logic. For this reason, we use *set transformers* instead of *predicate transformers*.

[5] We use the definitions of [11] over that of [1] because they seem to be easier to handle during proof; to avoid a notational clash we use $\lfloor \cdot \rfloor$ instead of $\{ \cdot \}$ and $\lceil \cdot \rceil$ instead of $[\cdot]$.

[6] *Angelic update* $\lfloor s \rfloor(\varphi) \ \widehat{=} \ \{v \mid s[\{v\}] \cap \varphi \neq \varnothing\}$ is missing from the list. We do not need it in this article.

where $\mathsf{grd}(\lfloor g_m \rfloor ; \lceil s_m \rceil) = \Phi$ would be required as a healthiness condition [13]. The two simple laws

$$\lfloor g \rfloor ; \lceil g \rceil \quad = \quad \lfloor g \rfloor \tag{12}$$
$$\lceil g \rceil ; \lfloor g \rfloor \quad = \quad \lceil g \rceil \tag{13}$$

permit us to switch between the two representations (10) and (11) in suitable contexts.

4.2 Refinement of Set Transformers

Denoting by $\sqsubseteq$ the ordering of set transformers

$$P \sqsubseteq Q \quad \widehat{=} \quad (\forall \varphi \cdot \varphi \subseteq \Phi \Rightarrow P(\varphi) \subseteq Q(\varphi)),$$

an extensive refinement theory can be developed for set transformers [11,21]. For a relation k let $\lceil k \rceil^{\sim}$ be the left adjoint of the set transformer $\lceil k \rceil$. It has the following simple characterisation [21]:

$$\lceil k \rceil^{\sim}(\varphi) \quad = \quad k[\varphi]. \tag{14}$$

A set transformer P is said to be *forward refined* by a set transformer Q, denoted by $P \sqsubseteq_k Q$, if

$$\lceil k \rceil^{\sim} ; P \quad \sqsubseteq \quad Q ; \lceil k \rceil^{\sim}.$$

Taking P and Q to be either of the form (10) or (11), forward refinement can be rephrased in relational terms [11,21]:

$$\lceil g \rceil ; \lceil s \rceil \sqsubseteq_k \lceil h \rceil ; \lceil t \rceil \quad \Leftrightarrow \quad k ; (h \lhd t) \subseteq (g \lhd s) ; k \tag{15}$$
$$\lfloor g \rfloor ; \lceil s \rceil \sqsubseteq_k \lfloor h \rfloor ; \lceil t \rceil \quad \Leftrightarrow \quad g \lhd k \subseteq k \rhd h \ \wedge \ g \lhd (k ; t) \subseteq s ; k \tag{16}$$

At its core refinement in Event-B corresponds to forward refinement of universally conjunctive set transformers of the form (10). This is the interpretation used in Section 3. But Event-B does not have to be interpreted in this way. This is discussed in more detail in the remainder of this section:

We want to verify that introducing a loop as described in [4] in Event-B is sound. Note that because of

$$g \lhd k \subseteq k \rhd h \ \wedge \ k ; (h \lhd t) \subseteq (g \lhd s) ; k \quad \Rightarrow \quad g \lhd (k ; t) \subseteq s ; k$$

it is sufficient to prove just $g \lhd k \subseteq k \rhd h$ on top of (15) so as to obtain (16). This indicates where to begin with a theory of sequential program refinement in Event-B. Matters get complicated by the presence of while loops and associated new events. We consider only this case because the case where loops are not involved is quite trivial as we have just seen.

4.3 Introduction of a While Loop

Let $m \in \alpha M$ and $n \in \alpha N \backslash \alpha M$. Our aim is to prove that the abstract event E_m is refined by a loop composed of the new event F_n followed by an assignment, the action of the concrete event F_m:

$$\begin{array}{l}\mathsf{while}\ H_n\ \mathsf{do}\\ \quad T_n\\ \mathsf{end}\,;\\ T_m\end{array}\quad .$$

We model the loop by the least fix point $(\mu X \cdot B(X))$:

$$\lfloor g_m \rfloor \,;\, \lceil s_m \rceil \quad \sqsubseteq_k \quad (\mu X \cdot B(X)) \,;\, \lceil t_m \rceil, \tag{17}$$

where the body of the loop is given in terms of the new event F_n [7]

$$B(X) \quad \widehat{=} \quad (\lceil h_n \rceil \,;\, (\lfloor h_n \rfloor \,;\, \lceil t_n \rceil) \,;\, X) \sqcap \lceil \Psi \backslash h_n \rceil.$$

(Because of law (13), we can simplify $B(X)$ to $(\lceil h_n \rceil \,;\, \lceil t_n \rceil \,;\, X) \sqcap \lceil \Psi \backslash h_n \rceil$. We may not want to carry out this simplification, though, if F_n is refined further. In that case we would want to replace $\lfloor h_n \rfloor \,;\, \lceil t_n \rceil$ in the longer formula by whatever refines it. In that case we would like a more concise loop guard than just the guard $\lceil h_n \rceil$ of the new event. We are not concerned about the exact form of the loop guards here, however. A systematic way of deriving them is presented in [4].)

Proof of (17). We assume event F_m refines event E_m,

$$\lceil g_m \rceil \,;\, \lceil s_m \rceil \sqsubseteq_k \lceil h_m \rceil \,;\, \lceil t_m \rceil, \tag{18}$$

and the loop $(\mu X \cdot B(X))$ forward refines $skip$, that is,

$$\lceil \mathsf{id}_\Phi \rceil \quad \sqsubseteq_k \quad (\mu X \cdot B(X)), \tag{19}$$

Note, that the update $\lceil \mathsf{id}_\Phi \rceil$ does not diverge, hence, the refinement (19) requires the new concrete event F_n to be convergent. Now,

$$\begin{array}{ll} & (17)\\ \equiv & \quad \{\ (14) \text{ and def. of } \lfloor\text{-}\rfloor \text{ and } \sqsubseteq_k\ \}\\ & \lfloor g_m \rfloor \,;\, \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lceil t_m \rceil\\ \equiv & \quad \{\ (*)\ \}\\ & \lfloor g_m \rfloor \,;\, \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lceil h_m \rceil \,;\, \lceil t_m \rceil\\ \equiv & \quad \{\ (12)\ \}\\ & \lfloor g_m \rfloor \,;\, \lceil g_m \rceil \,;\, \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lceil h_m \rceil \,;\, \lceil t_m \rceil\\ \Leftarrow & \quad \{\ \lfloor g_m \rfloor \ \sqsubseteq_k\ \lfloor k[g_m] \rfloor\ \}\\ & (18) \wedge (19) \end{array}$$

[7] The operator $\sqcap$ denotes *demonic choice* of set transformers: $(P \sqcap Q)(\varphi) \ = \ P(\varphi) \wedge Q(\varphi)$.

Only the inference marked by $(*)$ is missing. We close the gap by proving the following claim

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil \quad = \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)), \tag{20}$$

permitting to eliminate the guard h_m of the concrete event from the left hand side. This is done in the next two sections: in Section 4.4 we prove two claims facilitating the conclusion of the proof of (20) in the ensuing Section 4.5.

4.4 Analysing the While Loop

Our aim is to establish (20). In order to eliminate $\lceil h_m \rceil$, propagating some information through the loop seems a good idea. Hence, we have a closer look at the set transformer $\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X))$. Assuming the new event is convergent —as we do: (19)— we can exchange the least against the greatest fix point [11,18]:

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad = \quad \lfloor k[g_m] \rfloor ; (\nu X \cdot B(X))$$

So we can carry out fix point calculations using the greatest fix point.

First, we show $\boldsymbol{k[g_m] \subseteq (\nu X \cdot B(X))(k[g_m])}$**.** We know that the abstract guard g_m is an invariant of the concrete action s_n because the concrete event refines $skip$:

$$k[g_m] \quad \subseteq \quad \lceil t_n \rceil (k[g_m]) \quad . \tag{21}$$

We state without proof (compare [11, Lemma 21.9], for instance):

$$(\nu X \cdot B(X))(\varphi) \quad = \quad (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \ \cup \ ((\Psi \backslash h_n) \cap \phi)). \tag{22}$$

We prove that $k[g_m]$ is an invariant of the loop $(\nu X \cdot B(X))$. We calculate:

$$\begin{array}{lll}
 & (\nu X \cdot B(X))(k[g_m]) & \{ \text{ (22) } \} \\
= & (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \ \cup \ ((\Psi \backslash h_n) \cap k[g_m])) & \{ \text{ see def. of } b(x) \text{ below } \} \\
= & (\nu x \cdot b(x)) & \{ \text{ see below } \} \\
\supseteq & k[g_m]. &
\end{array}$$

We define $b(x)$ by $b(x) \ \widehat{=} \ (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap k[g_m])$ and prove the remaining claim $k[g_m] \ \subseteq \ (\nu x \cdot b(x))$; we insert $k[g_m]$ into $b(x)$:

$$\begin{array}{lll}
 & b(k[g_m]) & \{ \text{ def. of } b(x) \text{ } \} \\
= & (h_n \cap \lceil t_n \rceil (k[g_m])) \ \cup \ ((\Psi \backslash h_n) \cap k[g_m]) & \{ \text{ (21) } \} \\
\supseteq & (h_n \cap k[g_m]) \ \cup \ ((\Psi \backslash h_n) \cap k[g_m]) & \{ \text{ set theory } \} \\
= & k[g_m]. &
\end{array}$$

Using the fix point property (e.g. [11]),

$$\phi \subseteq b(\phi) \quad \Rightarrow \quad \phi \subseteq (\nu x \cdot b(x)) \quad ,$$

we conclude $k[g_m] \ \subseteq \ (\nu x \cdot b(x))$ as desired.

Second, $(\nu X \cdot B(X))((\Psi \backslash h_n) \cap \phi) = (\nu X \cdot B(X))(\phi)$**.** In other words, we can also show that $(\nu X \cdot B(X))$ establishes the negated guard of the concrete event; see [11]:

$$\begin{array}{lll} & (\nu X \cdot B(X))((\Psi \backslash h_n) \cap \phi) & \{\ (22)\ \} \\ = & (\nu x \cdot (h_n \cap \lceil t_n \rceil(x)) \ \cup \ ((\Psi \backslash h_n) \cap (\Psi \backslash h_n) \cap \phi)) & \{\ \text{set theory}\ \} \\ = & (\nu x \cdot (h_n \cap \lceil t_n \rceil(x)) \ \cup \ ((\Psi \backslash h_n) \cap \phi)) & \{\ (22)\ \} \\ = & (\nu X \cdot B(X))(\phi). & \end{array}$$

4.5 Use of the Enabledness Proof Obligation

Combining the first and second claim of the preceding section, we have proved:

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad = \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor \ . \quad (23)$$

Finally, we can discharge (20) by means of (7):

$$\begin{array}{lll} & \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil & \{\ (23)\ \} \\ = & \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor ; \lceil h_m \rceil & \{\ (*)\ \} \\ = & \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor & \{\ (23)\ \} \\ = & \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)), & \end{array}$$

where the step marked by $(*)$ depends on

$$k[g_m] \quad \subseteq \quad h_m \cup h_n \quad (24)$$

which corresponds to the enabledness proof obligation (7) with $L = \{m, n\}$. Using this proof obligation, we have proved something about preconditions. If we were committed to the failures semantics of Event-B, we would have had difficulties seeing this. Intuitively, deadlock-freeness appears quite distant from preconditions. The enabledness proof obligations permit us to weaken preconditions as usual in sequential program refinement [20]; we have $k[g_m] \ \subseteq \ h_m \cup h_n$ but only $k^{-1}[h_m] \ \subseteq \ g_m$.

Preservation of enabledness properties is achieved by simple rules governing their refinement [9]; the guard of each abstract event must imply the guard of the concrete event or the guard of some new event. This is just what we have shown to be necessary in this section. Loop introduction is proved by refinement. If we continue in this way correctness is preserved.

When developing sequential programs in Event-B we do not need to apply the possibly complex underlying theory directly but only know about the proof obligations of the kind given in the introduction. We do not need to be aware of the theory while modelling a program. We do not need to be aware of the theory while modelling other kinds of system either but simply rely on the proof obligations presented to us. A large amount of those proof obligations is shared among the the different kinds of system. This makes it easy for the same person to create models in the different domains without having to learn a new approach each time.

4.6 Limitations

In standard situations a system model is based on a specific, usually, well-known semantics. When using Event-B we only consider the kind of model created interesting, within the scope of this article, a sequential program or a reactive system. However, in these situations we do not worry too much by which means soundness was proved with respect to the proof obligations. We simply rely on the proof obligations as they are generated by some tool [6]. In standard situations we ought to be able to focus on modelling, and writing a model become mere routine. However, it is also possible that a model does not fit one of those situations. For instance, in the model described in [2] some events that are newly introduced must be convergent and some need not be. In that case one has to be aware of the semantics of the model justifying the presence of proof obligations and the absence of proof obligations. This is the price of the liberty one can have when modelling in Event-B. We can create models unconstrained by some semantics. This may be particularly useful for experimentation. But we have to be careful about what a model means and justify why we consider a particular model reasonable. For such models the semantics can be considered to be part of the properties of the system modelled — it is no longer given a priori as is the case in standard situations.

5 Conclusion

Event-B addresses various modelling domains among which reactive systems and sequential programs presented in this article. Event-B has a notation based on first-order predicate logic and set theory. Event-B has a set proof obligations that are associated with models.

What Event-B lacks is a behavioural semantics. And that is so intentionally. In fact, it would be difficult to support all those modelling domains using one semantics that would suit all. What we have seen, by way of two examples, is that the proof obligations of Event-B can be used in a way to fit with some intended semantics, be it relational or predicate transformer-based, be it for reactive systems or sequential programs. In some sense, in Event-B semantics is replaced by proof obligations. Possible semantics are characterised but not fixed.

The major advantage of this approach is that proof obligations can be used across the different domains. From our experience we know that they have a lot in common and seems a good idea to exploit this. For the different domains, though, proof obligations can be proved sound with respect to appropriate semantics. We would still like a model that is supposed to represent a sequential program, say, to have proof obligations that are sound with respect to a semantics for sequential programs. And this can be achieved in Event-B by linking the proof obligations to an appropriate semantic theory.

Acknowledgment. I want to thank Michael Butler and the second anonymous reviewer for helpful remarks and suggestions that improved clarity of the presentation, and Jean-Raymond Abrial for the productive discussions on this subject.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Event driven system construction (1999)
3. Abrial, J.-R.: Models of computations (1999)
4. Abrial, J.-R.: Event based sequential program development: Application to constructing a pointer program. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 51–74. Springer, Heidelberg (2003)
5. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (to appear, 2008)
6. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
7. 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)
8. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. Fundamentae Informatica 77(1-2) (2007)
9. Abrial, J.-R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
10. Back, R.-J.: Refinement Calculus II: Parallel and Reactive Programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 67–93. Springer, Heidelberg (1990)
11. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. In: Graduate Texts in Computer Science. Springer, Heidelberg (1998)
12. Butler, M.J.: Stepwise refinement of communicating systems. Science of Computer Programming 27(2), 139–173 (1996)
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
14. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: Bowmann, H., Derrick, J. (eds.) FMOODS 1997, vol. 2, pp. 423–438. Chapman & Hall, Boca Raton (1997)
15. Hallerstede, S.: Parallel hardware design in B. In: Bert, D., Bowen, J.P., King, S., Waldén, M.A. (eds.) ZB 2003. LNCS, vol. 2651, pp. 101–102. Springer, Heidelberg (2003)
16. Hallerstede, S., Hoang, T.S.: Qualitative probabilistic modelling in event-B. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 293–312. Springer, Heidelberg (2007)
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
18. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
19. Morgan, C.C.: Of wp and CSP. In: Feijen, W.H.J., van Gasteren, A.J.M., Gries, D., Misra, J. (eds.) Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra, pp. 319–326. Springer, Heidelberg (1990)
20. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice Hall, Englewood Cliffs (1994)
21. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge Tracts in Theoretical Computer Science 47. CUP (1998)
22. Roscoe, A.W.: Unbounded nondeterminism in CSP. Technical Monograph PRG-67, Programming Research Group, Oxford University (1988)
23. Sekerinski, E.: A calculus for predicative programming. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) MPC 1993. LNCS. Springer, Heidelberg (1993)
24. Woodcock, J., Davies, J.: Using Z. Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

Generating Tests from B Specifications and Test Purposes*

J. Julliand, P.-A. Masson, and R. Tissot

LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{julliand, masson, tissot}@lifc.univ-fcomte.fr

Abstract. This paper is about generating tests from test purposes, in addition to structural tests. We present a method that re-uses a behavioural model and an abstract test concretization layer developed for structural testing, and relies on additional test purposes. We propose, in the B framework, a process of test generation that uses the symbolic animation mechanisms of LTG (Leirios Test Generator) based on constraint solving, and guided by the test purposes. We build for that a B animable model that is the synchronized product of a behavioural B abstract model and a test purpose described as a labelled transition system. We prove the correctness of this method, and illustrate it by means of the IAS case study. IAS is a smart-card application dedicated to the operations of Identification, Authentication and electronic Signature.

Keywords: Model-Based Testing, Test Purpose, IAS Case Study.

1 Introduction

B models are well suited for producing tests of an implementation by means of a *model based testing* approach [UL06]. This approach proceeds by writing a *formal behavioural model* (M) of the expected functionalities of a system. This model is an abstraction of any real implementation, and is supposed to provide a reliable view of the implementation under test (IUT). By applying selection criteria, a test generation tool can automatically extract tests from the model. These tests are particular "executions" of the model. They are sequences of operation calls of the model, with the values of their parameters and their results as predicted by the model. The tests are abstract since they have the same level of abstraction as the model. They are concretized to execute them on the IUT by a *concretization layer* (CL). Comparing the results returned by the IUT with the ones predicted by the model allows delivering a verdict of the tests.

Structural testing uses static (syntactic) selection criteria, essentially providing control flow and data coverage of the model. The tests exercise the functionalities of the system by directly activating and covering the corresponding operations.

* Research partially funded by the French National Research Agency ANR (POSE ANR-05-RNTL-01001) and the Région Franche-Comté.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 139–152, 2008.

Industrial studies have proven the efficiency of the method to detect faults in an implementation (see for example [EFHP02, BLLP04]). Writing M and CL is an important effort, but the cost is justified by the possibility to automatically compute a great number of smart test cases. Nevertheless, static selection criteria appear to be insufficient to exercise the IUT in tortuous situations. We think for example of some elaborate scenarios of attack of systems requiring strong security guarantees. Our objective is to benefit from M and CL to compute some additional tests that use a particular scenario as a selection criterion.

The scenario can be described by means of a *test purpose* (TP), which we consider as a dynamic (semantic) selection criteria that orchestrates the successive calls of the operations of the model. The tests extracted from the model by means of a test purpose are sequences of operation calls corresponding to the scenario.

The context of this work is the test generation from B models. We use LTG [JL07], the test generator from Leirios[1], to automatically extract abstract tests from the model. LTG uses a constraint solver for computing the tests. LTG produces structural tests by applying a static criterion to cover all the paths of the control structure of every operation. Moreover, it is possible to assist the generation of tests by providing LTG with sequences of operation calls that describe the shape of the expected tests.

Our main contribution in this paper is to define in the B framework a process that uses LTG for generating abstract tests, with a dynamic selection criterion provided in the shape of a sequence of operations to LTG.

We give in Sec. 2 some preliminary definitions to our work. We present in Sec. 3 our process for computing and executing tests from a B model and a test purpose. Section 4 describes how to combine a behavioural model and a test purpose to obtain a B model for the test generation. We present the IAS case study and our experimentation in Sec. 5. We conclude and compare our proposition to related works in Sec. 6.

2 Preliminaries

This section gives the background of the paper. First, we give in Sect. 2.1 general notions about B abstract machines. We define the notions of B trace and B execution. We also define the restrictions due to the targeted application class and to the context of test generation. Section 2.2 defines a test purpose as a special kind of labelled transition system. It also presents the notions of TP trace and TP execution associated to the test purpose. The notion of trace is used to guide the test generation tool LTG that computes several executions for each trace.

2.1 B Abstract Machines

First introduced by J.-R. Abrial [Abr96], a B abstract machine defines an open specification of a system by a set of operations. Intuitively, an operation

[1] `http://www.leirios.com`

has a precondition and modifies the internal state variables by a generalized substitution. An operation is provided with a list of parameters and can return results.

We address a particular class of specifications of reactive systems. Our specifications are defensive, i.e. we assume that an operation terminates if it is invoked with well typed parameters. That means that we consider environments that respect a contract: they always call the operations with well typed parameter values. We also assume that any operation returns a status word that codifies a report of its execution. Therefore in the remainder of the paper, operations are defined as in Def. 1.

For defining a B abstract machine, we need to remind the reader of the notions of B predicates and B generalized substitutions. B predicates on a set of variables x are denoted by $P(x)$, $R(x)$, $I(x)$, $T(x)$, ... In the remainder of this paper, the predicate $I(x)$ denotes an invariant and $T(p)$ denotes a typing predicate on the parameter variables p. When there is no ambiguity on x, we simply denote the predicates by P, R, I, ... We denote by S the B generalized substitutions and by E, F, ... the B expressions. Given a substitution S and a post-condition R we are able to compute the weakest precondition P, such that if P is satisfied, then R is satisfied after the execution of S. The weakest precondition, defined in [Abr96], is denoted by $[S]R$. We denote by $\langle S\rangle R$ the expression $\neg[S]\neg R$, intuitively meaning that if $\langle S\rangle R$ is satisfied, then a computation of S exists terminating in a state satisfying R. Given a B substitution S, a particular predicate denoted by $prd_x(S)$ defines the relation between the values of the state variables x before the execution of S and the values of the state variable x' after the execution of S. $prd_x(S)$ is the pre-post predicate of S. It is defined in Def. 2. A B abstract machine is defined as in Def. 3.

Definition 1 (Operation). *Let S_i be a substitution. Let sw_i be a status word and p_i be a list of parameter names. Let $T_i(p_i)$ be a typing predicate on p_i. An operation named op_i is defined as* $sw_i \leftarrow op_i(p_i) =$ PRE $T_i(p_i)$ THEN S_i END.

Definition 2 (prd_x). *Let S be a substitution. The predicate $prd_x(S)$ is defined as $prd_x(S) = \langle S\rangle(x = x')$.*

Definition 3 (B Abstract Machine). *A B abstract machine* M *is a tuple $\langle x, I, Init, OP\rangle$ where*

- *x is a set of state variables,*
- *I is an invariant predicate over x,*
- *$Init$ is a substitution called* initialization*,*
- *OP is a set of operation definitions as in Def. 1.*

We denote as X_{M} (where $X \in \{x, I, Init, OP\}$) a component of the B model M. If there is no ambiguity on the model that is considered, we simply denote it by X. A model M defines a set $\mathcal{A}_{\mathsf{M}}$ of operation names and a set $\mathcal{P}red_{\mathsf{M}}$ of B predicates over the state variables x of M.

The test cases are finite executions. We first define the notion of *B trace* of a B abstract machine in Def. 4. Intuitively, a B trace is a finite sequence of operation names starting after the initialization.

Definition 4 (B Trace). *Let $M = \langle x, I, Init, OP\rangle$ be a B abstract machine. A trace is a finite sequence $\tau_M = Init; op_1; op_2; \ldots; op_n$ where op_i is the name of an operation ($\in \mathcal{A}_M$) defined in OP as in Def. 1.*

Several executions can be associated to a B trace because, for any operation op_i, there are possibly several parameter values v_i of p_i that satisfy the typing predicate $T_i(p_i)$. As can be seen in Def. 5, an execution is an instance of a B trace with parameter values for every operation call that satisfy the precondition $T_i(p_i)$.

Definition 5 (B Execution). *Let $M = \langle x, I, Init, OP\rangle$ be a B abstract machine. Let $\tau_M = Init; op_1; op_2; \ldots; op_n$ be a trace of M. $\sigma_M = (op_1(v_1), w_1); (op_2(v_2), w_2); \ldots; (op_n(v_n), w_n)$ is an execution associated to τ_M, denoted by $\sigma_M \in Exec_B(M, \tau_M)$, if there is a sequence of state variable values $u_0; u_1; u_2; \ldots; u_n$, a sequence of status words $w_1; w_2; \ldots; w_n$ and a sequence of parameter values $v_1; v_2; \ldots; v_n$ such that*

- $[x' := u_0]prd_x(Init)$,
- *for any* $i \in 1..n$: $[p_i := v_i]T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i]prd_x(S_i)$.

Since we assume our specifications to be defensive, there is at least one execution associated to a B trace if $T_i(p_i)$ is a satisfiable typing predicate. Thanks to that, we assume that the executions respect the contract, i.e. the environment (simulated by the test generator) always calls the operations with well-typed parameter values. In other words, the typing precondition is interpreted as a guard in B event systems, in such a way that the test generator chooses parameter values that satisfy the guard, i. e. the typing predicate $T_i(p_i)$. Moreover, the operation call $op_i(v_i)$ from the state u_{i-1} gives the new state variable values u_i and returns the status word w_i. u_{i-1}, u_i, w_i and v_i satisfy the pre-post predicate of S_i.

2.2 Test Purpose

We have defined in [JMT08] a language for describing test purposes, that combines operation calls and target state descriptions. Its semantics is given as a labelled transition system as in Def. 6. A test purpose TP is bound to a B abstract machine M that is the specification of the system under test. We say that TP is defined on M. We give a unique name to any transition in a set $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. The binding between TP and M is such that the transitions of TP are labelled by the names of the operations of M in $\mathcal{A}_\mathsf{M}$, and a state predicate of $\mathcal{P}red_\mathsf{M}$ on the variables x of M is associated to any state of TP.

Definition 6 (Test Purpose). *A test purpose on a model M is a tuple $\langle Q, q_0, T, \lambda, Q_f\rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is the set of terminating states, $T \in \mathcal{T} \to Q \times \mathcal{A}_M \times Q$ is a finite set of named and labelled transitions denoted by $t_i \longmapsto q_{i-1} \overset{op_i}{\to} q_i$, and $\lambda \in Q \to \mathcal{P}red_M$ is a total function that associates a state predicate, denoted by $\lambda(q_i)$, to every state.*

A test purpose TP defines a set of finite traces that represents a set of symbolic test cases. We call each trace a TP trace (see Def. 7). A TP trace is that of a finite

sequence of transitions that must be well formed w.r.t. the transition relation of TP. These symbolic test cases must be instantiated as test cases (non symbolic), called TP executions (see Def. 8) by a symbolic animator from a behavioural model M and some coverage criteria. In Def. 8, an execution is a finite sequence of pairs made of an operation call provided with the values of its parameters, and the expected status word value returned by the operation call.

The executions are easy to compute by a test generator when the TP traces are sequences of operations whose names have all been instantiated. Backtracking may be necessary to satisfy the constraints set by the predicates for the states to reach, and the enabling conditions of the operations.

Definition 7 (TP Trace). *A finite sequence of transitions* $\tau_{TP} = t_1; t_2; \ldots; t_n$ *is a trace of a test purpose* TP *if there are* $q_i \in Q$ *and* $op_i \in \mathcal{A}_M$, $0 < i \leq n$, *such that for any* $i \in 1..n$, $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$ *and* $q_n \in Q_f$.

Given a trace τ_{TP}, there are zero or many executions of τ_{TP} on the B abstract machine on which TP is defined.

Definition 8 (TP Execution). *Let* $M = \langle x, I, Init, OP\rangle$ *be a B abstract machine. Let* $\tau_{TP} = t_1; t_2; \ldots; t_n$ *be a trace of a test purpose* $TP = \langle Q, q_0, T, \lambda, Q_f\rangle$ *defined on* M. $\sigma_{TP} = (t_1(v_1), w_1); (t_2(v_2), w_2); \ldots; (t_n(v_n), w_n)$ *is an execution associated to* τ_{TP}, *denoted by* $\sigma_{TP} \in Exec_{TP}(M, \tau_{TP})$, *if there are a sequence of state values of* TP $q_0; q_1; q_2; \ldots; q_n$, *a sequence of state variable values of* M $u_0; u_1; u_2; \ldots; u_n$, *a sequence of status words values* $w_1; w_2; \ldots; w_n$ *and a sequence of parameter values* $v_1; v_2; \ldots; v_n$ *such that:*

- $[x' := u_0]prd_x(Init)$,
- *for any* $i \in 1..n$: $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$,
- *for any* $i \in 1..n$: $[p_i := v_i]T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i]prd_x(S_i) \wedge [x := u_i]\lambda(q_i)$.

As for the B executions, several TP executions can be associated to a TP trace for the same reasons. But in the TP executions, every operation call $op_i(v_i)$ must moreover lead to a state that satisfies the target state predicate $\lambda(q_i)$ which is associated to the target state q_i of the test purpose. For that, in Def. 8, we have added the following condition for any i: $[x := u_i]\lambda(q_i)$. Consequently, it is also possible that no execution is associated to a TP trace if there is no sequence $u_1; u_2; \ldots; u_n$ of state variable values that satisfy the sequence $\lambda(q_1), \lambda(q_2), \ldots, \lambda(q_n)$ of target state properties.

3 Process of Property Based Testing

Our process for generating tests uses a test purpose as selection criterion and a B behavioural model as oracle.

The complete process is described by Fig. 1. The left part of Fig. 1 shows how the set of abstract test cases is first computed, whereas the right part shows how these tests are finally executed on the IUT and the verdict is delivered.

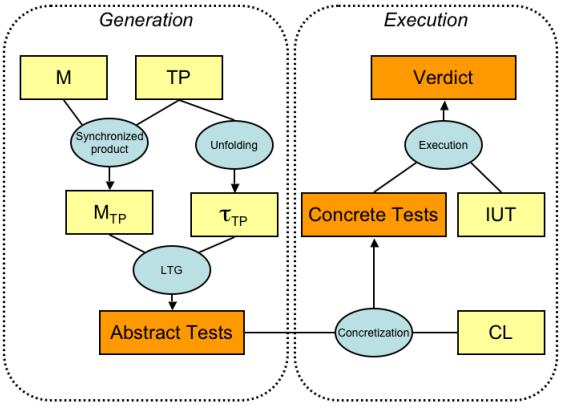


Fig. 1. Process for Generating and Executing Tests from a B model and a Test Purpose

Computing the abstract test cases is obtained by a symbolic animation of the TP traces on a B machine M_{TP} that is the synchronized product between the B model M and the test purpose TP. The synchronized product between M and TP is computed according to the expression in B that is given in Sec. 4. The result is a B machine M_{TP} whose executions are the possible executions from M that conform to TP. Besides, TP is unfolded as a finite set of TP traces (see Def. 7) τ_{TP}, i.e. as sequences of transition names (each one labelled with an unparameterized operation call) defined according to TP, but without the target states. This set computes all the TP traces whose last state is terminating, and whose length is lower or equal to a maximum length defined by the tester.

We use LTG, the test generator from Leirios, to instantiate the TP traces. LTG proceeds by symbolic animation. Notice that any other tool with similar capabilities could be used for that purpose. The principle is to "guess" values for the parameters of the operations that make it possible to execute the sequence of operations as described by a particular trace τ_{TP} of the test purpose. In other words, TP executions are computed from τ_{TP} and M_{TP}. The parameter values are computed in LTG by a constraint solver, that finds some values that make the sequences of operations of τ_{TP} reach the target states given in the TP. No execution is computed when the target states are impossible to reach. The status words are also computed as expected by M_{TP} for these parameters. Additionally, from one TP trace τ_{TP}, LTG will try to compute a different TP execution for each of the *behaviours* of the last operation of τ_{TP}: every branch of an operation described as a control structure (such as a conditional structure) is called a behaviour of the operation.

The tests computed by this procedure have the abstraction level of the model M of the system. They can not be executed as such on the IUT. They have to be concretized by the concretization layer CL which converts the instantiated

operation calls of the TP execution into a script executable on the IUT. This computes a set of *concrete tests*. These concrete tests can then be executed on the IUT, from which the output values (the status words) are observed. The concretization layer also gives the correspondence between the status words from the IUT and the ones from the model. This allows delivering the verdict of the test by comparing the values really returned by the IUT with the ones predicted by the model.

4 Combining a Model and a Test Purpose for Security Test Generation

In Fig. 2, we define how to express in B the synchronized product M_{TP} of a behavioural model M described as a B abstract machine, and a test purpose TP on M. M_{TP} includes the abstract machine M so that it can read the state variables x of M, and it can synchronize any transition t of TP with a call to an operation of M labelled by t. The variable Cq represents the current state reached by the last transition executed in the test purpose TP. The initial state is q_0. For any transition t_i (such that $T(t_i) = q_{i-1} \stackrel{op_i}{\rightarrow} q_i$), we define an operation also called t_i in M_{TP}. Its parameter values must satisfy the typing predicate $T_i(p_i)$ of the operation op_i that is called. This operation is enabled if the current state is q_{i-1} and if there are state variable values x' and a status word value sw'_i after t_i that satisfy the pre-post predicate of the body of the operation op_i and the target state predicate of the test purpose $\lambda(q_i)$. When these conditions hold, the operation t_i calls the operation of the test purpose op_i and places the system in the target state q_i of the test purpose.

Theorem 1 establishes the soundness of the method. For a TP trace $\tau_{\mathsf{TP}} = t_1; t_2; \ldots; t_n$ (see Def. 7), any B execution (see Def. 5) of the B composed abstract machine M_{TP} for the B trace $\tau_{\mathsf{M}_{\mathsf{TP}}} = \mathrm{Init}_{\mathsf{M}_{\mathsf{TP}}}; t_1; t_2; \ldots; t_n$ is a TP execution (see Def. 8) of τ_{TP} on the abstract machine M. Theorem 2 establishes the method completeness.

Theorem 1 (Soundness). *Let M_{TP} be the B composition of a B model M and a test purpose TP on M as in Fig. 2, and let τ_{TP} be a TP trace then,*

$$Exec_B(M_{TP}, Init_{M_{TP}}; \tau_{TP}) \subseteq Exec_{TP}(M, \tau_{TP}).$$

Proof. The proof relies on the fact that, the difference between the B executions of the model M and the TP executions of M, is that, the target predicate $\lambda(q_i)$ holds in every target state q_i of the TP execution. This condition is also satisfied in the B execution of M_{TP} since we add this condition in the guard of its operations t_i (see Fig. 2). Moreover, it is obvious that the B executions of M_{TP} and the TP executions of M compute the same sequence of states as TP, and execute the same sequence of operation calls as M.

Theorem 2 (Completeness). *Given a B composition M_{TP} of a B model M, a test purpose TP on M and a TP trace τ_{TP},*

$$Exec_{TP}(M, \tau_{TP}) \subseteq Exec_B(M_{TP}, Init_{M_{TP}}; \tau_{TP}).$$

MACHINE M
VARIABLES x
INVARIANT I
INITIALISATION Init
OPERATIONS
  ...
  $sw_i \leftarrow op_i(p_i) =$
    PRE $T_i(p_i)$ THEN S_i END
  ...
END

MACHINE M_{TP}
INCLUDES M
SETS $Q = \{q_0, \ldots, q_n\}$
VARIABLES Cq
INVARIANT $Cq \in Q$
  /* Cq : current state of TP */
INITIALISATION $Cq := q_0$
OPERATIONS
  /* for any $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$ */
  /* we define an operation t_i s.t. */
  ...
  $sw_i \leftarrow t_i(p_i) =$
  PRE $T_i(p_i)$ THEN
    SELECT $Cq = q_{i-1} \wedge \exists(x', sw_i') \cdot$
      $(prd_x(S_i) \wedge [x := x']\lambda(q_i))$
    THEN $sw_i \leftarrow op_i(p_i) \,||\, Cq := q_i$
    END
  END;
  ...
END

Fig. 2. Combination of a model M and a test purpose TP on M

The proof is straightforward.

Our implementation with LTG computes the B execution of M_{TP} with the semantics given in Def. 5. It is sound, but not complete because the constraint solving algorithm is time limited.

5 Case Study

5.1 IAS Case Study

This work was done in the framework of the RNTL POSE project, that brings together industrial (GEMALTO, LEIRIOS, SILICOMP/AQL) and academic (LIFC/INRIA CASSIS project, LIG) partners. The problem is the validation of a system conformity to its security policy, especially for smart cards.

Experiments have been made with a real size industrial application, the IAS platform. Prior to the project, a behavioural model in B had been written by the LIFC and Leirios, from which structural tests had been computed and executed on an IAS implementation by Gemalto. We have extended these tests with security ones.

IAS is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified in [GIX04] by GIXEL. IAS provides services to the other applications running on the card. IAS conforms to the ISO 7816 standard.

The file system of IAS is illustrated with an example in Fig. 3. Files in IAS are either *Elementary Files* (EF), or *Directory Files* (DF), e.g. `file_01` and `file_02`

in Fig. 3. The file system is organized as a tree structure whose root is designed as MF (*Master File*).

The *Security Data Objects* (SDO) are objects of an application that contain highly sensitive data such as PIN codes (e.g. `pin2` in Fig. 3) or cryptographic keys, that can be used to restrict the access to some of the application data.

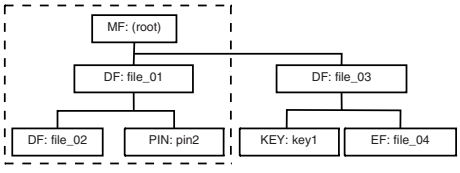


Fig. 3. A sample IAS tree structure

The access to an object by an operation in IAS is protected by security rules based on security attributes. The access rules can possibly be expressed as a conjunction of elementary access conditions, such as *Never* (which is the rule by default, stating that the command can never access the object), *Always* (the command can always access the object), or *User* (user authentication: the user must be authenticated by means of a PIN code).

Let us present the variables of the model that we use in an example of a test purpose given in Sec. 5.2. Let `X_ID` be a set of X identifiers, where X is either DF, PIN, OBJ or SDO. The variable `current_DF` ($\in$ `DF_ID`) stores the current selected DF. The variable `pin2_dfParent` ($\in$ `PIN_ID` $\nrightarrow$ `DF_ID`) associates to a PIN the DF where it is located. The variable `rule_2_obj` ($\in$ `SDO_ID` $\cup \{$`always`, `never`$\} \leftrightarrow$ `OBJ_ID`) associates to a SDO the object that it protects. If the object is always (resp. never) accessible, then the SDO is replaced by the value `always` (resp. `never`). The variable `pin_authenticated_2_df` ($\in$ `PIN_ID` $\leftrightarrow$ `DF_ID`) associates to a PIN the DF where the PIN is authenticated.

Consider for example the data structure shown in Fig. 3. `pin2` $\mapsto$ `file_01` $\in$ `pin2_dfParent` means that the PIN object `pin2` is located in the DF `file_01`. `pin2` $\mapsto$ `file_02` $\in$ `rule_2_obj` means that the access to the DF `file_02` is protected by a user authentication over the SDO `pin2`. If `pin2` $\mapsto$ `file_02` $\in$ `pin_authenticated_2_df`, then the access to the DF `file_02` is authorized, otherwise it is forbidden.

For creating objects, the commands are `CREATE_FILE_DF, PUT_DATA_OBJ_PIN_CREATE, ...` For navigating, they are `SELECT_FILE_DF_PARENT, SELECT_FILE_DF_CHILD, ...` For setting the values of attributes, they are `RESET_RETRY_COUNTER, CHANGE_REFERENCE_DATA, VERIFY, ...` For changing the life cycle state of objects, they are `DEACTIVATE_FILE, ACTIVATE_FILE, TERMINATE_FILE,...`

5.2 Test Purpose Example

Here, we exhibit one of the test purposes written for the experimentation of our approach. The property to be tested is "*to access an object protected by a PIN*

```
. (VERIFY | CHANGE_REFERENCE_DATA
| (RESET . SELECT_FILE_DF_CHILD) | RESET_RETRY_COUNTER
| (SELECT_FILE_DF_PARENT . SELECT_FILE_DF_CHILD))
   ⇝(current_DF = file_01 ∧ file_01 ∉ pin_authenticated_2_df[{pin2}])
. SELECT_FILE_DF_CHILD⇝(current_DF = file_02)
.[ CREATE_FILE_DF | DELETE_FILE | ACTIVATE_FILE | DEACTIVATE_FILE
| TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE ]
```

Fig. 4. Example of a test purpose — execution step

code, the PIN must be authenticated". We associate with this property a test purpose that causes the loss of the PIN authentication in all possible ways, and then tries to access the object.

This test purpose is instantiated on the example of Fig. 3, for which we imagine that the access to the DF `file_02` is protected by an authentication over the PIN `pin2`. The tester can describe this test purpose by regular expressions, as illustrated in Fig. 4. They are easily translated into the automaton shown in Fig. 5. The state properties in the states s_1 to s_7 are defined as B predicates over the state variables of the B model M, as in Fig. 4. The transitions from the state s_0 to the state s_4 aim at building the data structure surrounded by a dashed line in Fig. 3. The first transition creates a new DF (`file_01`). The second creates a PIN object (`pin2`) into the DF `file_01`, and gains an authentication over it. The third transition creates the DF `file_02` into the DF `file_01`. The fourth transition resets the current DF to `file_01`, in order to start the core of the test. As a result, the DF `file_02` is protected by the PIN `pin2` (located in the DF `file_01`) for all possible access commands. The PIN `pin2` is authenticated.

The following transitions translate the regular expression in Fig. 4, and show the core testing stage, describing the testing of the security property in three steps. First, the transitions between s_4 and s_5 describe all the possible ways for losing the authentication (for instance, a failure of the `VERIFY` command or a reset of the retry counter) over the PIN `pin2`. The transition from s_5 to s_6 selects the DF `file_02`. Finally, the transitions between s_6 and s_7 describe the application of the access commands inside the DF `file_02` to test the access conditions. The state s_7 is the terminating state.

5.3 Experimentation and Results

In this part, we give the results of an experimentation done with the B model of IAS which is 15500 lines long. The complete IAS commands have been modelled as a set of 60 B operations.

We first discuss what knowledge of the model is required to write the test purposes, and then we present our experimental results.

Designing test purposes. The description language is based upon regular expressions, which makes it easy to use. But designing a test purpose requires some knowledge of the model. The tester must know the names of the different operations of the model, and of the state variables and constants to describe

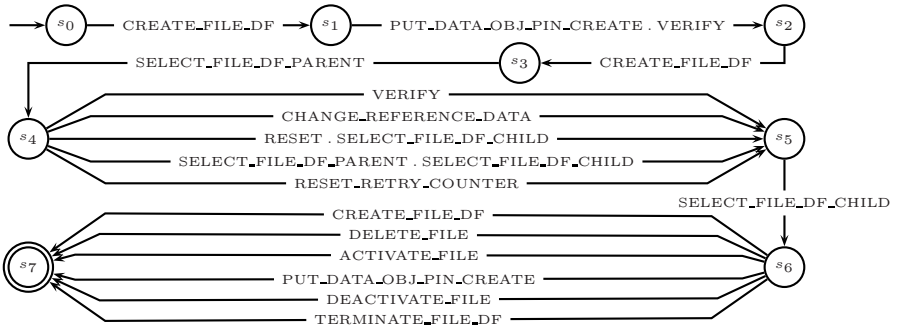


Fig. 5. Example of a test purpose

the states to reach. Moreover, he must choose the right behavioural level for the description of the test purposes, to ensure good performances of the test generation. For example, inserting a state to reach between two operation calls allows reducing the search space, but requires searching which conditions ensure the execution of the second operation without reducing its reachable behaviours.

Experimentations. We have experimented with three different test purposes, which gave a total of 183 tests that have been run on the IAS implementation. The test purpose example shown in Fig. 5 gave 30 test TP traces, which have produced 35 TP executions. This is because sometimes there are several possible behaviours to cover in the last operation of the TP trace.

The two other test purposes were for testing possible bad interpretations of the access conditions due to a mechanism of short references to the security objects, and the effects of life cycle changes on the authentication of a PIN. In these various test campaigns, we have successfully instantiated every TP trace, except when they contained unreachable states (w.r.t. the constraints on the operations sequencing). Furthermore, we have begun an experimentation to test the POSIX compliance of a file system. We have already generated 250 test sequences (from 5 test purposes) for this case study. By now, these sequences are able to test non-trivial executions of the system with basic operations.

Experimental results. The tests that we have generated are not redundant w.r.t. the tests computed with static coverage criteria like behaviour coverage. This is because the test purposes force the test generator to reach some given states or to apply some operation sequences, which would not have been necessarily reached or covered otherwise. These tests address some situations which have been identified as potential vulnerabilities, and which were not addressed by the previously generated structural tests.

We illustrate the difference between tests based on a test purpose and structural tests through the example of the aforementioned property: the access to DF `file_02` is protected by PIN code `pin2`. The automaton associated to the TP that we have considered for this property is shown in Fig. 5. Let us imagine

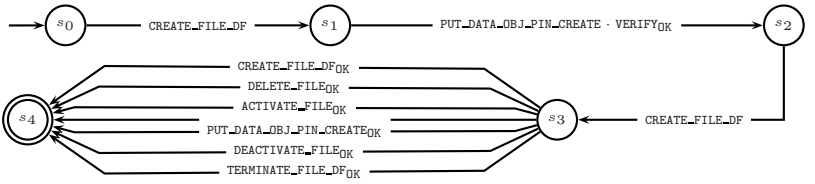


Fig. 6. Structural testing for an authorized access

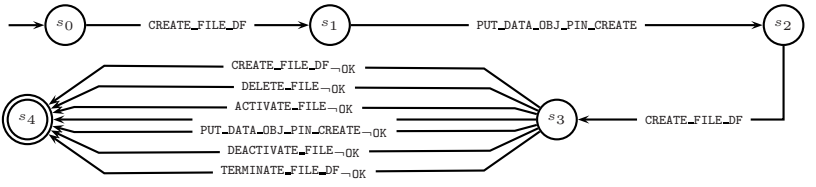


Fig. 7. Structural testing for an access denial

the same kind of automaton, but for a structural test related to this property. Structural testing will exercise the property in two ways: by gaining an authentication over `pin2` and successfully accessing `file_02`, or by not gaining the authentication and thus failing to access `file_02`. The simplest (and shortest) way not to gain the authentication is by not calling the `VERIFY` command: this is what LTG does for this example.

The automata for these two cases are respectively given in Fig. 6 and Fig. 7. In these two automata, the initialization stage is (almost) identical to the automaton for the TP, because we want to compare the tests based on the same data structure. In comparison to Fig. 5, the only difference is about the `VERIFY` command that is given with its expected result (denoted by the subscript `OK`) in Fig. 6, and absent from Fig. 7.

Then the core testing stage only consists of trying to access `file_02`, which is always refused, as denoted by the subscript ¬`OK` in Fig. 7, and always allowed in Fig. 6 (in Fig. 5, the expected result of every access command should be ¬`OK`).

The value-added of the tests from the TP is to force the coupling between a successful authentication and (later) an access denial. In other words, two operation behaviours are coupled in the same execution, whereas they were not tested together with structural testing.

Another advantage of using test purposes is that they are issued from a potential vulnerability, to which the tests computed can be linked. This traceability is more difficult to obtain for structural tests.

6 Conclusion and Future Work

We have presented in the B framework a method for generating tests from test purposes in a behavioural model based testing context. The tests generated are

additional w.r.t. the structural ones [BLLP04, SLB05]. The method has been validated on a real-size industrial application. The method makes use of already existing material, written for model based structural testing: the behavioural model and the concretization layer. Additionally, test purposes are written to describe how to test behavioural properties.

The method easily ensures the traceability of the tests generated to the original test purpose, since the tests are computed from them. Also, with the traceability mechanism for functional test generation that we use, we know which operation behaviours have been covered.

Many other works use test purposes as selection criteria to extract tests from a model. The test purposes are described by temporal properties in a temporal logic, input output Labelled (Symbolic) Transition Systems ioLTS (ioSTS), or use cases.

By exploiting its ability to produce counter-examples, a model-checker can be used to compute tests from temporal properties [ADX01]. These techniques are restricted to finite systems. The TGV approach [CJMR07, JJ05], uses explicit test purposes to extract tests from specifications, both given as ioLTS or ioSTS [JJRZ05]. Our approach also addresses infinite systems, like ioSTS. ioSTS are specifications where the data are integers and booleans, whereas the B models define more complex set data structures. So, our approach is based on set constraint solving techniques whereas ioSTS use integer abstract interpretation and constraint solving techniques.

In [SML06], the authors present a test case generation algorithm from B event systems and use cases by refinement. There are three main differences with our approach. Our method reuse abstract B machines and a concretization layer CL dedicated to the functional test generation. Therefore we do not refine the test cases. Moreover, our test purposes are more expressive use cases that contain target state information.

Also, as a difference with the above cited approaches, we have showed in a previous work [MJP+07] how the test purposes can be automatically computed, by modelling some *test needs* as syntactic transformation rules that transform behavioural properties.

We are currently working at identifying and writing such transformation rules, based on the IAS case study. This work needs to be developed by studying many other case studies (for instance, the mini-challenge that proposes to design and verify a POSIX compliant flash-based system [JH07]) in order to produce rules sufficiently generic to be applicable to a variety of examples.

Rules could also be automatically deduced from the syntactic expression of a property, as suggested by [BDGJ06] for properties expressed in JTPL, a temporal logic for JML.

References

[Abr96] Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)

[ADX01] Amman, P., Ding, W., Xu, D.: Using a model checker to test safety properties. In: ICECCS 2001. IEEE Computer Society, Los Alamitos (2001)

[BDGJ06] Bouquet, F., Dadeau, F., Groslambert, J., Julliand, J.: Safety property driven test generation from JML specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 225–239. Springer, Heidelberg (2006)

[BLLP04] Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. Software: Practice and Experience 34(10), 915–948 (2004)

[CJMR07] Constant, C., Jéron, T., Marchand, H., Rusu, V.: Integrating formal verification and conformance testing for reactive systems. IEEE Transactions on Software Engineering 33(8), 558–574 (2007)

[EFHP02] Farchi, E.E., Hartman, A., Pinter, S.S.: Using a model-based test generator to test for standard conformance. IBM Systems Journal 41(1), 89–110 (2002)

[GIX04] GIXEL. Common IAS Platform for eAdministration, Technical Specifications, 1.01 Premium edition (2004), `http://www.gixel.fr`

[JH07] Joshi, R., Holzmann, G.: A mini challenge: build a verifiable filesystem. Formal Aspects of Computing 19(2), 269–272 (2007)

[JJ05] Jard, C., Jéron, T.: TGV: theory, principles and algorithms. Software Tools for Technology Transfer 7(1) (2005)

[JJRZ05] Jeannet, T., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 349–364. Springer, Heidelberg (2005)

[JL07] Jaffuel, E., Legeard, B.: LEIRIOS Test Generator: Automated test generation from B models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 277–280. Springer, Heidelberg (2006)

[JMT08] Julliand, J., Masson, P.-A., Tissot, R.: Generating security tests in addition to functional tests. In: AST 2008. ACM Press, New York (May 2008)

[MJP+07] Masson, P.-A., Julliand, J., Plessis, J.-C., Jaffuel, E., Debois, G.: Automatic generation of model based tests for a class of security properties. In: A-MOST 2007, pp. 12–22. ACM Press, New York (2007)

[SLB05] Satpathy, M., Leuschel, M., Butler, M.: ProTest: An automatic test environment for B specifications. In: MBT 2004. ENTCS, vol. 111, pp. 113–136 (2005)

[SML06] Satpathy, M., Malik, Q.-A., Lilius, J.: Synthesis of scenario based test cases from B models. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 133–147. Springer, Heidelberg (2006)

[UL06] Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science, Amsterdam (2006)

Combining Scenario- and Model-Based Testing to Ensure POSIX Compliance

Frédéric Dadeau, Adrien De Kermadec, and Régis Tissot

Laboratoire d'Informatique – Université de Franche-Comté
LIFC/INRIA CASSIS Project – 16 route de Gray – 25030 Besançon cedex
{dadeau, dekermadec, tissot}@lifc.univ-fcomte.fr

Abstract. We present in this paper a way to produce test suites for the POSIX mini-challenge, based on a formal model of a file system manager, written using a B machine. By this case study, we illustrate the limitations of a fully-automated testing process, which justifies the use of scenarios that complements the classical functional testing approach. Scenarios are expressed through schemas, focusing only on operation chaining. They are played on the model using a symbolic animation engine in order to automatically compute pertinent operation parameter values, based on model coverage criteria such as behavioral or data coverage. We concretize our experimentation by testing the POSIX conformance of two different file systems: a recent Linux distribution, and a customized Java implementation of POSIX used to evaluate the relevance of our approach.

Keywords: B machine, scenarios, Model-Based Testing, symbolic animation, POSIX challenge.

1 Introduction

The use of formal methods becomes crucial when one wants to design a system that requires a high level of correctness. In this context, a mini-challenge is proposed to design and verify a POSIX compliant flash-based system [6]. Whilst this proposal is not the first attempt to put formal methods into a real case study, it illustrates the increasing need for formal modelling and verification that aims at producing safe and secure software. In the process, testing takes an important part which is used to provide witnesses that, first, the system does not contain certain kind of execution errors, and, second, that it behaves as expected.

Our proposal is to consider a model-based testing approach [15], that relies on a formal model of the system. The latter is used to compute the test cases, as a sequence of operation calls intended to cover functional requirements. It also provides the oracle of the test, namely the expected result that will be used to ensure the conformance of the system w.r.t. the model. We have decided to apply this technique in the context of the mini-challenge, and thus, we have designed a formal model of the system, by means of a B machine.

Automated test generation processes suffer from a number of limitations, especially when applied to the domain of security testing (e.g. access control). Indeed,

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 153–166, 2008.

the systematic analysis and partitioning of the model may possibly avoid testing specific configurations of the system that can be more precisely identified by the experience and the know-how of the test engineer, rather than through the automated analysis of models. We therefore propose to extend the functional test cases by scenarios. These latter are expressed by schemas that, roughly speaking, can be assimilated to regular expressions over the alphabet represented by the operations of the system.

This paper presents a schema expression language that makes it possible to describe scenarios with test driving possibilities. This process is based on the use of the symbolic animation of the formal model in order to instantiate the test data to obtain pertinent values. This language has originally been designed during the French RNTL POSÉ project[1], in the context of the security testing. It is now used for a larger purpose.

The paper is organized as follows. Section 2 presents the subject and the scope of the case study. Then, we present an automated test generation approach using the LEIRIOS Test Generator [5] and its limitations in Sect. 3. Section 4 presents a scenario-based paradigm that is used to extend the testing phase. Experimental data are provided in Section 5. Finally, Section 6 concludes and presents future work.

2 Formal Model of a POSIX File System

This section presents the formal model that we have designed from the informal specifications. This model is used in the test generation process. It includes a realistic subset of the functionalities required in the challenge. We designed our model from the Open Group Base Specification of Unix[2]. The data model we have considered is relatively similar to the Z specification of the Unix filing system [10].

2.1 Data Model

The basic elements of the file system are the entities, called *nodes* decomposed into directories and files. They are both associated to a parent directory into which they are located, and a name, which is the user designation of the node in the directory. We also consider file descriptors, used to read from and/or write into a file. Files and directories (resp. names) are atoms that are extracted from a single abstract set of *node ids* (resp. names), named *IDS* (resp. *NAMES*). Each existing node, file or directory, is a member of *IDS* and associated to one of the *NAMES*.

$$nodes \subseteq IDS \land dirs \subseteq nodes \land files \subseteq nodes \land$$
$$dirs \cap files = \emptyset \land dirs \cup files = nodes \land id2name \in nodes \rightarrow NAMES$$

[1] `http://www.rntl-pose.info`
[2] `http://www.unix.org/single_unix_specification/`

The file structure is modelled by associating to each entity its parent directory, through the *id2parent* total function: $id2parent \in nodes \rightarrow dirs$. Notice that we do not manage symbolic links on files or directories.

A special directory representing the root of the file system is also considered, that is its own parent. The root directory is the only existing one at the initial state of the system (e.g. after having formatted the file system).

$$id\_root \in IDS \ \wedge \ id\_root \in dirs \ \wedge n\_root \in NAMES \ \wedge$$
$$(id\_root \mapsto n\_root) \in id2name \ \wedge \ (id\_root \mapsto id\_root) \in id2parent$$

For opened files, we consider a set of file descriptors that are integers, identifying the file descriptor. Each file descriptor is mapped to a node id. In addition, we consider a mapping between a file descriptor and the opening mode that is considered, i.e., read (`r`), write (`w`) or both (`rw`). A constant named *MAX_FD* sets the maximal number of files that can be opened.

$$fd2node \in fd \rightarrow nodes \wedge fd2mode \in fd \rightarrow \{\mathtt{r}, \mathtt{w}, \mathtt{rw}\} \wedge fd \subseteq 0..MAX\_FD$$

2.2 Commands

For the experiment, we have decided to model the behavior of common operations that manipulate the file system structure, and the file content. We added several operations that complete the original requirements of the mini-challenge. The operations that we consider are the following. *Jedi/rmdir*, creates/removes a directory; *chdir*, changes the current directory; *open/close*, opens/closes a file descriptor used to read/write into a file. *read/write*, reads from or writes into a file designated by its opened file descriptor; *rename*, renames a file/directory or possibly moves the file; *(f)truncate*, resizes a given file (descriptor); *(f)stat*, provides informations on the file (descriptor) or directory; *opendir/closedir*, opens/closes a directory for reading.

Figure 1 shows, as an example, the `mkdir` operation that is used to create a directory in the file system. For readability purposes, this operation does not include the access control mechanisms. As for all the operations that require a path to be given as input, we represent this parameter as a triplet composed a path root (the ID of the starting directory), a path target (the ID of the destination directory), and a name (the name of the considered entity). Such a representation makes it possible to rebuild the path when the tests are concretized, may it be absolute (i.e., the starting directory is the root), or relative (i.e., the starting directory is the current directory). This operation returns a status word (`OUT_sw`) indicating a correct termination (`ok`) or an erroneous termination (`ko`) of the operation which updates the `errno` state variable indicating the cause.

2.3 Adding Acces Control Mechanisms

Taking access control mechanisms into account adds a potentially large number of elements, namely: the users, their groups, and the read/write/execute permissions on files and directories. Rules are expressed through a discretionary access

```
OUT_sw, OUT_dir ← mkdir(IN_pathRoot, IN_path, IN_name) ≙
   PRE
      IN_pathRoot ∈ IDS ∧ IN_pathRoot ∈ {current_dir, root_id} ∧
      IN_path ∈ IDS ∧ IN_name ∈ NAMES
   THEN
      IF IN_name = EMPTY_NAME  ∨ IN_path ∉ dirs ∪ files
      THEN  /* the path does not exist */
         errno := ENOENT ‖ OUT_sw := ko
      ELSE
         IF IN_path ∈ files
         THEN   /* pathname points to a regular file*/
            errno := ENOTDIR ‖ OUT_sw := ko
         ELSE
            IF (ids2parent~[{IN_path}] ∩ ids2name~[{IN_name}]) ≠ ∅
            THEN  /* the directory already exists */
               errno := EEXIST ‖ OUT_sw := ko
            ELSE  /* a node that does not exist */
               IF (nodes = IDS) THEN OUT_sw := ko ‖ errno = EMFILE ELSE
                  ANY LOC_id WHERE LOC_id ∈ IDS ∧ LOC_id ∉ nodes THEN
                     OUT_sw := ok ‖ errno := OK ‖ OUT_dir := LOC_id ‖
                     dirs := dirs  ∪ {LOC_id} ‖
                     ids2name := ids2name ∪ {LOC_id ↦ IN_name} ‖
                     ids2parent := ids2parent ∪ {LOC_id ↦ IN_path}
      END END END END END END;
```

Fig. 1. The `mkdir` Operation of the POSIX File System Model

control policy (DAC) [7] , meaning that the access is granted to the owner of the entity, which is able to change the access rights to whoever he wants. POSIX distinguishes three access modes: *read*, *write* or *execute*, for three roles: *user*, *group*, *others*. Each user belongs at least to one group.

Existing users (*user*) are related to existing groups (*groups*), the ownership of a node in terms of user (*node*2*own*) and group (*node*2*gr*) is also considered. Permissions are decomposed into three categories for each node: user permission (*permUser*), group permssion (*permGr*), others permission (*permOth*) by their octal value (read-write-execute bits). The formal data model is given hereafter.

$$user \subseteq UID \wedge groups \subseteq GID \wedge user2gr \in user \leftrightarrow group \wedge$$
$$node2own \in node \rightarrow user \wedge node2gr \in node \rightarrow group \wedge$$
$$permUser \in node \rightarrow 0..7 \wedge permGr \in node \rightarrow 0..7 \wedge permOth \in node \rightarrow 0..7$$

Using this data model, the expression that checks, for example, the *write* access for a given user u on node n is the following.

$$\begin{aligned}
&(node2own(n) = u \Rightarrow permUser(n) \in \{4,5,6,7\}) \vee \\
&(node2own(n) \neq u \wedge node2gr(n) \in \{u\} \lhd user2gr \Rightarrow permGr(n) \in \{4,5,6,7\}) \vee \\
&(node2own(n) \neq u \wedge node2gr(n) \notin \{u\} \lhd user2gr \wedge permOth(n) \in \{4,5,6,7\})
\end{aligned}$$

2.4 Restrictions on the Modeling

Since our objective is to generate tests based on a B model, we use the state-of-the-art LEIRIOS Test Generator (LTG) [5] to automatically compute tests from a B machine. However, due to the limitations of the tool, we have modified our model. Since abstract sets are not authorized by LTG, we have instantiated the sets of node ids (*IDS*) and names (*NAMES*). Moreover, LTG being unable to deal with sequences, and sets of sets, we have decided to abstract the content of the file system, in order to focus on the access control mechanisms. Files are thus considered though different attributes that are their size, their owner, group, permissions, and a version number that is incremented each time the file is modified. We rely on the concretization phase of the tests to translate the effects of file reading/writing commands on the concrete file system.

3 Functional Testing Approach

We present in this section the model based testing mechanisms. First, we introduce the testing criteria automatically applied by LTG. Second, we present the observations that we consider to establish whether the test is passed or failed.

3.1 Model Coverage Criteria

LTG considers a model coverage criterion that consists in activating the behaviors of the B operations. Behaviors are defined as a specific path in the control flow graph of an operation. Each behavior is composed of an activation condition, i.e., a predicate that has to hold for the behavior to be activated. This predicate is called the *test target*. If the behaviors contain a decision, e.g. expressed in B using an IF... THEN... ELSE... END structure, a *decision coverage* criteria, selected by the user, is applied in order to refine the test target.

The composition of LTG test cases are shown in Fig. 2. Each test starts with a *preamble* that is a sequence of operation calls that reaches a state satisfying the test target. The test *body* is the activation of the behavior itself by invoking the operation with the appropriate parameters values. The *identification* is used to perform calls to specific operations, used to observe the system state, and thus, to deduce the result of the test. The optional *postamble* feature makes it possible to reset the system in order to chain the test cases on the system under test.

Example 1 (Test target computation). Consider again the `mkdir` operation from the POSIX case study shown in Fig. 1. The following targets are extracted,

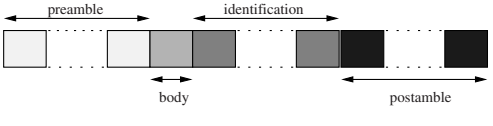


Fig. 2. Composition of an LTG Test Case

under the hypothesis of the operation precondition ($\texttt{IN\_pathRoot} \in IDS \wedge \texttt{IN\_pathRoot} \in \{\texttt{current\_dir, ROOT\_ID}\} \wedge \texttt{IN\_path} \in IDS \wedge \texttt{IN\_name} \in NAMES$).

Target	Predicate
1	$\texttt{IN\_name} = \texttt{EMPTY\_NAME} \vee \texttt{IN\_path} \notin \texttt{nodes}$
2	$\texttt{IN\_name} \neq \texttt{EMPTY\_NAME} \wedge \texttt{IN\_path} \in \texttt{nodes} \wedge \texttt{IN\_path} \in \texttt{files}$
3	$\texttt{IN\_name} \neq \texttt{EMPTY\_NAME} \wedge \texttt{IN\_path} \in \texttt{nodes} \wedge \texttt{IN\_path} \notin \texttt{files} \wedge (\texttt{ids2parent}\tilde{}[\{\texttt{IN\_path}\}] \cap \texttt{ids2name}\tilde{}[\{\texttt{IN\_name}\}]) \neq \emptyset$
4	$\texttt{IN\_name} \neq \texttt{EMPTY\_NAME} \wedge \texttt{IN\_path} \in \texttt{nodes} \wedge \texttt{IN\_path} \notin \texttt{files} \wedge (\texttt{ids2parent}\tilde{}[\{\texttt{IN\_path}\}] \cap \texttt{ids2name}\tilde{}[\{\texttt{IN\_name}\}]) = \emptyset \wedge \texttt{nodes} = \texttt{IDS}$

Since target 1 contains a disjunction, it may be expanded according to 4 rewritings, each one describing a specific coverage criteria, given in the following table, in which $p = (\texttt{IN\_name} = \texttt{EMPTY\_NAME})$ and $q = (\texttt{IN\_path} \notin \texttt{nodes})$. Each rewriting produces additional test targets replacing the initial predicate. Inconsistant targets are detected by the tool and are thus not considered.

	$p \vee q \rightsquigarrow$	Coverage Criteria
1	$p \vee q$	Condition Cov. (CC)
2	p, q	Condition/Decision Cov. (C/DC)
3	$p \wedge \neg q$, $\neg p \wedge q$	Modified Condition/Decision Cov. (MC/DC)
4	$p \wedge \neg q$, $\neg p \wedge q$, $p \wedge q$	Multiple Condition Cov. (MCC)

Each test target produces at least one test, depending on the selected data coverage for the operation parameters (e.g. all values, smart value, boundary values for numerical data). LTG automatically performs the symbolic animation of the B model in order to reach a state satisfying the test target. The result of this computation provides the preamble of the test. It is important to notice that the state exploration algorithm aims at finding the shortest path to the test target, since no parameterization on the preamble length can be performed (e.g. in order to maximize it).

3.2 Test Verdict Definition

The conformance relationship that we consider is based on observing command outputs w.r.t. data given as input. We define two conformance relationships: a heavyweight and a lightweight.

The heavyweight conformance relationship is based on observing if the operation succeeds, and if not, what is the corresponding error code. A test passes if the implementation corresponds to the model for both of these two elements. Due to the possibility of multiple errors and the presence of non-determinism in the POSIX specification, we also consider a lightweight conformance relationship, which only determines, for each step, if the operation call succeeded or failed, without considering the reason of the error.

In practice, the lightweight conformance relationship is sufficient to test a POSIX implementation. Indeed, POSIX operations have the particularity either to succeed and make the system evolve, or they fail and nothing changes in the system (except the `errno` variable). Thus, all the errors can be abstracted.

3.3 Limitations of This Approach

This approach presents the great advantage of performing an exhaustive coverage of the activation contexts of an operation. It thus makes it possible to cover a large variety of use cases of each operation. Nevertheless, LTG does not cover the possibilities of reaching a context. Indeed, the state exploration performed by LTG to build the preamble (i.e., to reach the test target) always chooses the shortest possible path. As a consequence, possibly interesting sequences are simply avoided.

For example, if we consider the test target corresponding to the successful creation of a new directory with the `mkdir` operation, a test will be produced to check the behavior of `mkdir` with a new name as a parameter. Nevertheless, an other interesting test would be to reuse the name of a deleted file or directory. Such a preamble is not taken into account by LTG.

4 Combining Scenario- and Model-Based Testing

Our proposal is to extend the "classical" functional test generation approach with a scenario-based testing approach that uses the symbolic animation of the model to decide the feasability of the test sequences and to compute test data. Another advantage of using scenarios is that they can help reaching "unreached" test targets, especially those related to boundary values coverage. Indeed, consider the file/directory creation operations (e.g. `create` or `mkdir`), automatically computed test target would require the maximal number of nodes in a directory to be reached. Unfortunately, such a target would require numerous successive creation of nodes. As a matter of fact, LTG preamble computation is natively limited in both time and state exploration depth, preventing such a target from being reached. Nevertheless, such a test target can be reached by considering the suitable scenario.

4.1 Principles of Scenario- and Model-Based Testing

Scenario-based testing strengthens a classical functional test generation campaign by generating complementary test cases, that are designed based on the experience of the validation engineer. These tests respond to specific test needs identified by the engineer when reading the informal specifications.

Tool support can be used to ease the test design, such as the Tobias [9] combinatorial testing tool. This latter takes as input combinations of operation sequences and their possible parameters, expressed through finite regular expressions, and combines operations and parameters together. This is a quick

way to produce large test suites, but which may unfortunately be irrelevant, especially when the combination of parameters' values does not satisfy the operation preconditions. Indeed, this declination of scenario-based testing does not use a model. Thus, irrelevant operation calls can be generated and appear in the resulting test suite. Moreover, the engineer is asked to provide the operation parameters values, which might be a complicated task and requires both skills and experience in testing. Also, the validation engineer should have a good knowledge of the system in order to be able to correctly design the operation chaining that reaches a specific system state.

Experiments on combinatorial testing have shown that most of the time, the combinatorial explosion is introduced by the neverending iterations over the possible parameter values. We thus propose to abstract the operation parameters, in order to focus only on the operations chaining.

The principle of the scenario- and model-based testing approach is to combine the essence of combinatorial testing, i.e., being grounded on the know-how of the validation engineer, and the power of the symbolic animation of a formal model [3], in order to compute relevant test data. Scenarios are written using *schemas*, that are now described.

4.2 Schemas Language

In this section, we introduce the language that we have designed to formally express test schemas. We wanted the language to be as generic as possible w.r.t. the modelling language used to formalize the system. It is thus structured in two different layers: the *sequence* layer, and the *test generation directive* layer.

The former describes scenarios in terms of operation calls and state properties to be reached. State properties are expressed in the language of the formal model. In order to describe the multiple behaviors of a schema, the language is based on regular expressions, and introduces notions of choices or iterations. The latter deals with combinatorial issues, by specifying coverage criteria intended to the test generation tool.

Syntax of the Sequence Layer. The concrete syntax of the sequence layer is given hereafter:

```
OP      ::=operation_name
        — $OP
        — $OP "\{" OPLIST "}"

OPLIST::=operation_name
        — operation_name " , " OPLIST
SP      ::=state_predicate

SEQ     ::=OP1 — "(" SEQ ")" — SEQ"⇝(" SP ")"
        —SEQ "." SEQ
        —SEQ REPEAT
        —SEQ CHOICE SEQ

REPEAT::="*" — "+" — "?" — "{" n "}"
        —"{" n ",}" — "{," n "}" — "{" n "," m "}"
```

The SP rule describes the state predicates whereas OP is used to describe operation calls, which may be expressed by: (*i*) an operation name, (*ii*) the $OP tokens meaning any operation, or (*iii*) using $OP\{OPLIST} meaning any operation, except those in list OPLIST.

The SEQ rule describes a sequence of operation calls as a regular expression. A step of sequence is either an operation call, denoted by OP1, or an operation call leading to a state satisfying a state predicate, denoted by SEQ $\rightsquigarrow$(SP). This latter represents the major improvement w.r.t. usual scenario description languages, since it makes it possible to define the target of a sequence of operation, without necessarily enumerating the operations composing the sequence.

Sequences can be composed by the concatenation of two sequences, the repetition of a sequence or the choice between two sequences. We use usual regular expression iteration operators, augmented with bounded repetition operators (exactly n times, at least n times, at most n times, between n and m times).

Syntax of the Test Generation Directive Layer. This part of the language is given hereafter.

CHOICE ::="|" | "$\otimes$"

OP1 ::= OP | "["OP"]"

It allows to specify guidelines for the test generation step. We propose two kinds of directives aiming at reducing the search for instantiations of the test schema.

The `CHOICE` rule introduces two operators, denoted as | and $\otimes$, for covering the branches of a choice. Let S_1 and S_2 be two test schemas. Schema $\mathtt{S_1} \mid \mathtt{S_2}$ specifies that the test generator must generate tests for both schema S_1 and schema S_2. $\mathtt{S_1} \otimes \mathtt{S_2}$ specifies that the test generator must generate tests either for schema S_1 or schema S_2.

The rule `OP1` tells the test generator to cover one of the behaviors of operation `OP`. It is the default option. The test engineer can also perform the coverage of all the behaviors of the operation by surrounding its call with brackets.

Test Schema Example. The test schema hereafter aims at reaching a state with the maximum amount of files opened. Such a target can be generated automatically using a boundary value coverage on the number of currently opened files. In order to compute the preamble, a large number of calls to the *open* operation have to be performed. Since LTG's automated test generation process is natively limited in the number of operations in the preamble, such a target would never be reached.

open$\rightsquigarrow$(id1 $\in$ existing_files) .

(open* $\rightsquigarrow$(card(fd)=MAX_OPEN $\wedge$ ran(fd2ids)={id1}) |

opendir* $\rightsquigarrow$(card(fd)=MAX_OPEN $\wedge$ ran(fd2ids)={ROOT_ID})

) . ([open] | [opendir])

In this example, we assume that the initial state of the system only contains the root directory. The schema is constituted of 3 steps. The first one aims at creating a new file (using the *open* operation). The second opens the maximum amount of file descriptors pointing either on files or on directories. The third and

final step explores the different behaviors of either *open* or *opendir* commands in this state. It unfolds in 4 operation sequences that have to be instantiated. Such an instantiation is now described.

4.3 From Schemas to Test Cases Using Model Symbolic Animation

Symbolic animation is a process that consists in performing the exploration of the states of a model. Instead of considering concrete states, in which the state variables have a defined and concrete value, symbolic animation considers groups of concrete states, named *symbolic states*. These latter are introduced by leaving operation parameters unspecified. Thus, they are replaced by a symbolic value related to the corresponding data domain. Furthermore, all state variables whose evolution depend on such symbolic parameters also become symbolic.

Example 2 (Symbolic Animation of the `mkdir` *Operation).* Consider the sequence:

$$< init; \texttt{mkdir(id\_root,id\_root,}N_1\texttt{)}; \texttt{mkdir(id\_root,id\_root,}N_2\texttt{)} >$$

based on the `mkdir` operation of Fig. 1, in which the name parameters are abstracted by two symbolic variables N_1 and N_2. After the successful execution of these two operations, the $id2name$ state variable becomes symbolic, associated to the following constraints:

$$ids2name = \{id0 \mapsto N_1, id1 \mapsto N_2\} \ \wedge N_1 \in NAMES \wedge N_2 \in NAMES \wedge N_1 \neq N_2$$

In practice, symbolic animation relies on constraint solving techniques that make it possible to instantiate the parameters after the execution of the sequence. In our example, one possible instantiation is N_1=`n01` and N_2=`n02`.

Once unfolded, the tests are played on the model using the symbolic animation engine of LTG (similar as the one presented in [3]). At each step, the parameters are replaced by symbolic values. Once the sequence has been entirely (symbolically) played, the intermediate operation parameters are automatically instantiated by the constraint solvers, so that the resulting test case is *consistent*, meaning that the sequence of operations is feasible (as defined in [1]), and each potential intermediate state predicate holds. It is possible for the validation engineer to select the data coverage he wants to apply. By default, a single value satisfying the constraints is chosen. An advanced test driving mechanism makes it possible to select a boundary value coverage of numerical inputs.

5 Experiments

This section presents the expriments performed, by first giving an overview of the systems under test (SUT) we have considered. Then, we introduce the schemas we have designed and their results on the SUTs.

5.1 Systems Under Test

We have considered two target systems for our tests to be run. The first one is a standard Linux kernel (version 2.6.20-16-generic on an Ubuntu distribution). The second one is a hand-made implementation of POSIX in Java (meant to be later adapted to JavaCard). We considered these targets for two purposes. The Linux distribution makes it possible for us to validate our model. Indeed, even if Linux systems are, as a matter of fact, not POSIX-compliant, the differences with the POSIX standards can not be found in the basic subset of commands that we have considered. Thus, playing our tests on a Linux system, through the C interface, makes it possible to ensure the conformance of our model w.r.t. the standard. Nevertheless, we needed to make sure that our tests were useful. Using our own Java implementation of the POSIX standard makes it possible, first, to check this implementation w.r.t. the validated model, and, second, to easily introduce errors in this implementation in order to measure the fault detection capabilities of our tests, through mutations.

We wrote translators, named *adaptation layers*, that concretize the abstract XML test cases, produced by LTG, to system-dependant test cases. The first translator produces C files containing calls to the corresponding C functions. The second translator produces a JUnit test file containing calls to the corresponding Java methods. The resulting files embed the test oracle, defined as the conformance relationship given in 3.2. Notice that such translators may easily be adapted to test other implementations of POSIX.

5.2 Test Schema and Results

We present here the description of several of the test schemas used to generate the test cases during the experimentation[3].

A functional test campaign has been computed using LTG, producing 78 tests. These tests were completed by the schemas given hereafter. The first two schemas are designed to overcome LTG limitations is reaching boundary values. The two last schemas are designed in order to extend the functional test campaign:

- `maxOpen`: tests the maximal number of file opened (4 tests)
- `maxFileSize`: tests the maximal file size (24 tests)
- `removedRef`: tests the reuse of a (re)moved file name or closed file descriptors (147 tests)
- `multiFd`: tests the multiple use of a file descriptor (64 tests)

We have unfolded the test schemas and instantiated the parameters w.r.t. the model. Thus, we have run our tests on our two SUTs. The functional campaign highlighted 7 non-conformances (on 78) between the Linux kernel and the B model. The analysis of the tests show that they come from a different interpretation of implementation-dependant elements of the specification. The tests issued from the schemas were successfully run on our Linux distribution.

[3] The original model, the schemas, and the resulting tests in C are available at `http://lifc.univ-fcomte.fr/~dadeau/posix/`

Running all the tests on the Java implementation revealed a few bugs (e.g. array indexes out of bounds, null pointer exceptions) but basically no serious non-conformance were found. We introduced several functional errors in this implementation. Among them, we only allowed one file descriptor per system file. This error was not detected by the LTG test suite, but was found using schema `multiFd`. Another error was to incorrectly close an opened file descriptor, so that it is possible to write/read through this file descriptor. Also, this error, not found by LTG, was found using schema `removedRef`. These results illustrate the relevance of using test schemas and the ease of using the schema language.

6 Conclusion and Future Work

We have presented in this paper the combination of scenario- and model-based testing techniques. This approach relies on a schema language, based on regular expressions, that makes it possible to describe symbolic scenarios as operation sequences whose parameters values are computed by a symbolic animation engine. This process is a great improvement of existing techniques. It provides a tool support which relieves the validation engineer from, first, writing repetitive test sequences, and, second, finding consistent test data that make it possible to execute an operations' suite. This latter is an improvement w.r.t. other scenario based testing approaches, even applied to B [13]. In related works, scenario based testing mainly focuses on extracting scenarios from UML diagrams, such as the SCENTOR approach [16] or SCENT [11] using statecharts. The SOOFT approach [14] proposes an object oriented framework for performing scenario-based testing; nevertheless, the scenarios have to be completely described, contrary to our approach that abstracts the difficult task of finding well-suited parameter values. The combination of symbolic techniques and scenarios for generating tests based on models is an original combination that do not seem to have been targeted before.

We applied our technique on a model of the POSIX case study written as a B machine. It includes the most common operations on file systems for manipulating directories and files. We have used this model as an input to the LEIRIOS Test Generator, in order to automatically produce functional test cases. These tests were completed by a combination of scenario- and model-based tests. We also experimented the ProB test generator (ProTest) [12]. Unfortunately, ProB's systematic enumeration of operation parameter values, on the model, suffers from combinatorial explosion and makes it impossible to handle the model. A related but less advanced work has been presented in [4]. This latter used a MBT approach to test the `fcntl` byte range locking APIs of POSIX.

This work is a contribution to the POSIX mini-challenge, for which the purposes are twofold. First, the generated tests can be run on the final implementation in order to make sure that it conforms to the model. Second, if the code of the implementation is automatically generated from the model, the tests that we propose can be played on this latter in order to make sure that it behaves as expected in the requirements of the POSIX standard. Thus, our testing technique

can be employed in the validation phase of the application, either for validating the model, the implementation or both. Therefore, only simple adapters have to be written to translate our tests in the considered target format.

We are now interested in extending our approach. First, we plan to refine our schema language in order to increase its expressiveness. For example, using the Tobias-2 format [8] would make it possible to capture the expressiveness of our language, and would also permit to additionally specify some of the parameter values, which are currently systematically all abstracted. Second, we plan to automate the production of test scenarios based on formal properties expressed on the model. Such an approach would be related to [2] in which we previously used temporal safety properties to produce test cases.

References

1. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Bouquet, F., Dadeau, F., Groslambert, J., Julliand, J.: Safety property driven test generation from JML specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 225–239. Springer, Heidelberg (2006)
3. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B: A constraint solver to animate a B specification. International Journal on Software Tools for Technology Transfer, STTT 6(2), 143–157 (2004)
4. Farchi, E., Hartman, A., Pinter, S.S.: Using a model-based test generator to test for standard conformance. IBM Systems Journal 41(1), 89–110 (2002)
5. Jaffuel, E., Legeard, B.: LEIRIOS Test Generator: Automated Test Generation from B Models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 277–280. Springer, Heidelberg (2006)
6. Joshi, R., Holzmann, G.: A mini challenge: build a verifiable filesystem. Formal Aspects of Computing 19, 269–272 (2007)
7. Lampson, B.W.: Protection. SIGOPS Oper. Syst. Rev. 8(1), 18–24 (1974)
8. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the tobias-2 test generator. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 535–536. ACM, New York (2007)
9. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 281–294. Springer, Heidelberg (2004)
10. Morgan, C., Sufrin, B.: Specification of the UNIX filing system. In: Specification case studies, pp. 91–140 (1987)
11. Ryser, J., Glinz, M.: A practical approach to validating and testing software systems using scenarios (1999)
12. Satpathy, M., Leuschel, M., Butler, M.: ProTest: An Automatic Test Environment for B Specifications. Electronic Notes in Theroretical Computer Science 111, 113–136 (2005)
13. Satpathy, M., Malik, Q.A., Lilius, J.: Synthesis of Scenario Based Test Cases from B Models. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 133–147. Springer, Heidelberg (2006)

14. Tsai, W.T., Saimi, A., Yu, L., Paul, R.: Scenario-based object-oriented testing framework. qsic, 00:410 (2003)
15. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach, 550 pages. Elsevier Science, Amsterdam (2006)
16. Wittevrongel, J., Maurer, F.: Scentor: Scenario-based testing of e-business applications. In: WETICE 2001: Proceedings of the 10th IEEE International Workshops on Enabling Technologies, Washington, DC, USA, 2001, pp. 41–48. IEEE Computer Society, Los Alamitos (2001)

UseCase-Wise Development: Retrenchment for Event-B

Richard Banach

School of Computer Science,
University of Manchester,
Manchester, M13 9PL, UK
banach@cs.man.ac.uk

Abstract. UseCase-wise Development, the introduction of functionality into an application in stages, with each stage being carried through to (ideally) implementation before the next is considered, is examined with a view to its being treated via an Event-B methodology. The need to modify top level behaviour in a non-skip way precludes its naive treatment via Event-B refinement, and paves the way for the use of retrenchment in Event-B. The details of an Event-B formulation of retrenchment, aligned to the practical details of the Rodin toolset, are described. The details of refinement/retrenchment interworking needed to handle UseCase-wise development are outlined, and a simple case study is given.

Keywords: Event-B, UseCase-wise Development, Incremental Development, Refinement, Retrenchment, Tower Pattern.

1 Introduction

One of the notable things about the move from traditional B [1] to Event-B [2,3], is the way that the re-engineered refinement theory of Event-B has managed to encompass many 'low hanging fruit' issues, for the handling of which, retrenchment has been advanced in more conventional refinement frameworks in the past. One can mention: the introduction of new events at successive development levels (within certain restrictions); the emphasis on guards (rather than preconditions) and their strengthening during refinement; the migration of information between I/O variables and state variables (since in Event-B there is generally no separate category of I/O variables to worry about); and so on. All of this is beneficial, in bringing such issues under more rigorous control than when using other development techniques (or when using retrenchment).

Nevertheless, because in Event-B (as in every other rigorous refinement framework), the development strategy and the notion of correctness is fixed *ab initio* —and yet the world is richly and subtly structured— it is almost inevitable that sooner or later an application scenario will arise in which the demands of Event-B will prove to be a less than ideal fit for the application in question. It is to help accomodate situations like these that retrenchment was originally conceived, so it is natural to ask what retrenchment amounts to in the Event-B context, and how the notions of Event-B refinement and Event-B retrenchment would interact. Fortunately, since the original introduction of retrenchment [4], we have accumulated a good deal of experience and evidence on which to base the answer (see eg. [5]).

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 167–180, 2008.

In this paper we examine retrenchment in the Event-B context, by looking at a small case study developed using a UseCase-wise development methodology. UseCase-wise development is our name for a development strategy in which increments of functionality are added in stages, with the introduction of each resulting in a usable application before the next is considered. Such an approach is at odds with the more traditional waterfall model with which typical formal development approaches are frequently aligned. We view the exploration of alternative strategies as a good motivation for studying how retrenchment should be formulated in Event-B, a question which is of independent interest in any case.

The rest of this paper is as follows. In Section 2 we describe UseCase-wise development, contrasting it with conventional Event-B development. Section 3 briefly reviews Event-B and discusses the details of retrenchment for Event-B. In Section 4 we cover retrenchment/refinement interworking and the Tower Pattern. The preceding ingredients are then applied to a small case study, illustrating a good fit between the UseCase-wise approach and Event-B correctness when retrenchment is available. Section 6 concludes.

2 UseCase-Wise Development

In Event-B there is a strong emphasis on getting the requirements correct (or as near correct as is achievable) at the outset. One then analyses the requirements and determines the most appropriate order in which to take them into account within a sequence of refinements. The refinements themselves, mix the accretion of requirements issues as identified during requirements analysis, with data refinements, as appropriate. As the models get more detailed, sound decomposition techniques are available to split models into components, allowing further refinements to be done independently. This TopDown (TD) approach, proceeding as it does in an essentially linear manner, shows that the Event-B approach can be viewed as a formal interpretation of a fairly traditional waterfall strategy.

By UseCase-wise (UCw) development, we mean an approach to system development that proceeds by taking one or more of the UseCases identified during requirements analysis, and completes the development of those first, from the abstract models down to implementation, giving a usable system (with limited functionality). Subsequently further UseCases are incorporated, with all the elements of the development getting suitably enhanced, and yielding another working system, this time with greater functionality. The process is repeated until all the UseCases identified during requirements analysis have been developed, yielding a system with all the functionality desired. UCw development can be seen as a member of the 'Agile Methods' family of system development techniques.[1]

[1] We coined the term 'UseCase-wise development' to avoid confusion. It is enough to glance at http://en.wikipedia.org/wiki/Agile_software_development and the acronym blizzard one finds there, with the same term having different meanings in different settings, to realise what dangers lurk in the casual use of terms invented in this field. What we call UseCase-wise development is also called 'incremental development' in other places, but that term is so laden with possibilities for misinterpretation, that we thought it safest to invent a fresh name, inevitably causing yet more terminological proliferation.

Fig. 1. Illustrating TopDown versus UseCase-wise development strategies

Fig. 1 illustrates the TD versus UCw distinction. On the left we see a development proceeding TD in layers, while on the right, we see additional slices of functionality being added UCw to an initially developed system. It is important to realise that the TD vs. UCw distinction refers to the *dynamics* of the process by which the system is built. Even though a system may be built using a UCw process, one which is superficially unsympathetic to Event-B perspectives, there is no reason why the end result should not be a collection of models which enjoy the levels of mutual consistency characteristic of Event-B. Thus, even though one might argue that introducing a UCw approach into Event-B would be a retrograde step for Event-B, it is hard to dispute that introducing Event-B's criteria for correctness into the UCw approach would be a positive step for UCw development. This begs the question of how one might incorporate Event-B correctness into the UCw process. This will be dealt with in Section 4.

3 Event-B Machines, Refinement and Retrenchment for Event-B

In this section, we review Event-B machines, refinements, and against this background, we formulate retrenchment in a way that will permit the smoothest possible cooperation between the two techniques.

3.1 Event-B Machines

In a nutshell, an Event-B MACHINE has a *name*, it SEES one or more *static contexts*, and it owns some VARIABLES; these are allowed to be updated via EVENTS, but are required to always satisfy the INVARIANTS. The events can declare their own *parameters* (which are bound variables acting as carriers of input values) — each event has one or more *guards*, and one or more *actions* which are specified via *before-after predicates* (or notations such as assignment for simpler cases). Among the events there is an *INITIALISATION*, whose guard must be true.

The semantics of Event-B machines and of the refinement relationship between machines, is expressed via a number of proof obligations (POs). These must be provable in order for the machine or refinement in question to be well defined. We quote the main ones of interest to us, mentioning the others more briefly. See [2,3] for full details.

For a machine A to be well defined the *initialisation* and *correctness* POs must hold:

$$Init_A(u') \Rightarrow I(u') \tag{1}$$

$$I(u) \wedge G_{Ev_A}(i,u) \wedge Ev_A(u,i,u') \Rightarrow I(u') \tag{2}$$

In (1), $Init_A$ is the initialisation event and (1) says that the value u' of A's state variable u established by $Init_A$ satisfies A's invariant I. Likewise, (2) says that for an event Ev_A of A, if A's invariant I, and Ev_A's guard $G_{Ev_A}(i,u)$, both hold in the before-state of the event, and Ev_A's before-after relation $Ev_A(u,i,u')$ also holds, then the after-state will satisfy the invariant I once more. In (1) and (2) we have suppressed mention of the details of the static contexts seen by A, but we have singled out Ev_A's input variables i for later convenience. For closer conformance to [2,3] we have not mentioned any output variables, though it would be trivial to include them in the before-after relation $Ev_A(u,i,u')$ and in (2). Aside from (1) and (2), Event-B machines must satisfy *feasibility* POs for the initialisation and for all events, and also a *deadlock freedom* PO for non-terminating systems; see [2,3].

3.2 Event-B Refinement

Suppose that as well as machine A, we have another machine C, with state variable w, input variable k, initialisation event $Init_C$, and typical event Ev_C, with guard $G_{Ev_C}(k,w)$ and before-after relation $Ev_C(w,k,w')$. If C is a refinement of A, its invariant $K(u,w)$ will be a relation over both u and w, and the counterparts of (1) and (2) are:

$$Init_C(w') \Rightarrow (\exists\, u' \bullet Init_A(u') \wedge K(u',w')) \tag{3}$$

$$I(u) \wedge K(u,w) \wedge G_{Ev_C}(k,w) \wedge Ev_C(w,k,w') \Rightarrow (\exists\, i,u' \bullet G_{Ev_A}(i,u) \wedge Ev_A(u,i,u') \wedge K(u',w')) \tag{4}$$

where Ev_C is an event that is supposed to refine Ev_A and we have amalgamated the *guard strengthening* and *correctness* POs in (4) for later convenience. In (3), each $Init_C(w')$ intialisation must be witnessed by some $Init_A(u')$ intialisation that establishes the joint invariant $J(u',w')$. Likewise, (4) says that when both invariants hold, each $Ev_C(w,k,w')$ event is witnessed by some $Ev_A(u,i,u')$ event that re-establishes the joint invariant. Aside from (3) and (4) there are also *feasibility* POs for the initialisation and for all events, *variant decrease* POs for 'new' C events not declared to be refinements of any event of A, and also an overall *relative deadlock freedom* PO. See [2,3] for full details.

We give a small example of Event-B refinement. It builds a directed graph from a finite universe of possible nodes contained in a set *NSet* held in a context *NCtx*.

Machine *Nodes* is concerned with the requirement of assigning nodes to the graph, picking them out of the set *NSet* using the event *AddNode*, starting with the empty set. Machine *Edges* refines Nodes, and addresses the requirement of having edges between some of the graph nodes. In typical Event-B fashion, it simply accumulates the new model elements, leaving the preceding ones unchanged. So *Edges* just contains *Nodes* in its body. The new requirement is handled by adding a new variable *edg* and a new event *AddEdge*. *AddEdge* acts like skip on the existing variable *nod*, as required for such 'new' events. Also since *AddEdge* does not refine any existing event (unlike *AddNode* which refines itself and is thus 'ordinary'), it must be 'convergent', which means that each invocation of *AddEdge* decreases the $\mathbb{N}$-valued VARIANT $\mathrm{card}(NSet \times NSet - edg)$, ensuring relative deadlock freedom. (We suppress the WHICH IS clauses below.)

MACHINE *Nodes*
SEES *NCtx*
VARIABLES *nod*
INVARIANTS
$nod \in \mathbb{P}(NSet)$
EVENTS
INITIALISATION
WHICH IS ordinary
BEGIN $nod := \varnothing$ END
AddNode
WHICH IS ordinary
ANY n
WHERE $n \in NSet - nod$
THEN $nod := nod \cup \{n\}$
END
END

MACHINE *Edges*
REFINES *Nodes*
SEES *NCtx*
VARIABLES nod, edg
INVARIANTS
$nod \in \mathbb{P}(NSet)$
$edg \in \mathbb{P}(NSet \times NSet)$
EVENTS
INITIALISATION
WHICH IS ordinary
BEGIN $nod := \varnothing$ END
AddNode
WHICH IS ordinary
REFINES *AddNode*
ANY n
WHERE $n \in NSet - nod$
THEN $nod := nod \cup \{n\}$
END
AddEdge
WHICH IS convergent
ANY n, m
WHERE $\{n, m\} \subseteq nod$
$n \mapsto m \in NSet \times NSet - edg$
THEN $edg := edg \cup \{n \mapsto m\}$
END
VARIANT $\mathrm{card}(NSet \times NSet - edg)$
END

3.3 Retrenchment for Event-B

We now formulate retrenchment for Event-B against the preceding background. The objective of retrenchment is to offer a flexible relationship between machines or system models that can capture situations in which all the detailed criteria of some species of refinement cannot be met, but where the two models in question are deemed nevertheless (and especially by domain experts rather than refinement specialists) to belong to the same development activity. The focus of retrenchment is on a simulation-like criterion, with the added aim of convenient interworking with refinement. Retrenchment is therefore formulated as a modification of the main POs of the refinement notion, with the incorporation of suitable additional predicates to enhance expressivity.[2]

For the specific context of Event-B, retrenchment is a relationship that is to hold between top level machines. When a retrenchment involving a refinement machine is needed, one must quantify away the dependence on the higher level abstractions to get a self-contained top level machine using the technique described in Chapter 11 of [1].

Unlike refinement in Event-B, in which the refinement data (essentially just the joint invariant and some bookkeeping details, as in our example) are incorporated into the syntax of the refining machine, retrenchment is an independent syntactic construct, as befits the weaker relationship between machines that it expresses, and especially, the desire that none of the details of retrenchment interfere in any way with any refinement that any machine involved in a retrenchment might also be involved in. Notationally this departs from the scheme in [4] and agrees with the line taken in [6,7,8].

[2] Thus the modification of the relevant refinement POs constitutes the sense in which the *simulation-like criterion* is intended; suitable pairs of transitions in the two models should satisfy an appropriate generalisation of (4).

Suppose we have top level machines A (having the elements mentioned earlier) and B, and B's state and input variables are v, j, the invariant is $J(v)$ and the other pieces can be imagined. Here is a schematic syntax for the retrenchment construct, intended as a good fit for Event-B as currently implemented in the Rodin toolset [9]:

```
RETRENCHMENT Identifier Ret_{A,B}
FROM Identifier A TO Identifier B
[ SEES IdentifierList ]
[ RETRIEVES Predicate R(u, v) ]
[ EVENTS
   [ RAMIFICATIONS Identifier Ev_A [ TO Identifier Ev_B ]
      [ WITHIN Predicate W_{Ev_A,Ev_B}(i, j, u, v) ]
      [ CONCEDES Predicate C_{Ev_A,Ev_B}(u', v', i, j, u, v) ]
      END
   ]+
]
END
```

The construct has a name $Ret_{A,B}$, and is FROM machine A TO machine B. It can SEE static contexts as can a machine or refinement. There is a RETRIEVES relation $R(u, v)$ between the two state spaces, and for each pair of retrenchment-related events in A and B, eg. Ev_A and Ev_B (where one can omit mentioning Ev_B if it has the same name as Ev_A), there are the RAMIFICATIONS, consisting of the WITHIN relation $W_{Ev_A,Ev_B}(i, j, u, v)$ and the CONCEDES relation $C_{Ev_A,Ev_B}(u', v', i, j, u, v)$.

The semantics of retrenchment is given by its POs. These are:

$$Init_B(v') \Rightarrow (\exists\, u' \bullet Init_A(u') \wedge R(u', v')) \tag{5}$$

$$I(u) \wedge R(u, v) \wedge J(v) \wedge W_{Ev_A,Ev_B}(i, j, u, v) \wedge Ev_B(v, j, v') \Rightarrow (\exists\, i, u' \bullet Ev_A(u, i, u') \wedge (R(u', v') \vee C_{Ev_A,Ev_B}(u', v', i, j, u, v))) \tag{6}$$

where there is an instance of (6) for each ramifications-related pair Ev_A and Ev_B. We see that the intialisation PO is standard, while the correctness PO permits considerable deviation from refinement-like behaviour by virtue of the presence of the within and concedes relations. In addition to the above, we demand for each Ev_A/Ev_B pair that:

$$W_{Ev_A,Ev_B}(i, j, u, v) \Rightarrow G_{Ev_A}(i, u) \wedge G_{Ev_B}(j, v) \tag{7}$$

which is the criterion that ensures smooth retrenchment/refinement interworking. Note however, that the other POs of Event-B refinement, variant decrease and relative deadlock freedom, do not have counterparts in retrenchment; we want to be able to relate machines with significantly different behaviour as regards these aspects, if desired.

Although we do not have space here to fully examine the arguments why the above design is a good one for retrenchment, we can make the following remarks. Firstly, the aim of a notion that *departs from* refinement or *desires to accommodate inability to satisfy* refinement, must amount to a weakening of refinement — there is clearly no point in doing the opposite. The proposal we have given above does this, since the occurrences of $W_{Ev_A,Ev_B}(i, j, u, v)$ and $C_{Ev_A,Ev_B}(u', v', i, j, u, v)$, in the hypotheses and conclusions respectively, of (6), clearly weaken (4). Secondly, we want this weakening to be as general as possible so as not to have to invent a different notion of non-refinement for every conceivable departure from refinement that might arise. Again, (6)

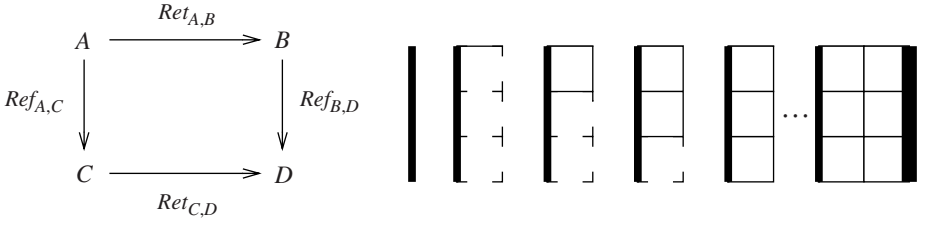


Fig. 2. The basic structure for the Tower Pattern on the left, and on the right, its use in constructing a UCw development whose outcome enjoys the rigour of an Event-B development

achieves this since $W_{Ev_A,Ev_B}(i,j,u,v)$ and $C_{Ev_A,Ev_B}(u',v',i,j,u,v)$ must be specified on a per-event-pair basis. Thirdly, we would want the departure from refinement to be quantified in some way. Again, (6) achieves this, at least indirectly, since $W_{Ev_A,Ev_B}(i,j,u,v)$ and $C_{Ev_A,Ev_B}(u',v',i,j,u,v)$ must actually be specified by the user in each particular case of retrenchment — in doing this the precise details of how refinement fails to hold is made clear by the user in the details of these (otherwise unconstrained) relations. Lastly, we would want good interworking with refinement. This important topic is the subject of the next section. See [5] for more extensive discussion of generalities such as these concerning retrenchment.

4 Retrenchment and Event-B Interworking: The Tower Pattern

A definite *sine qua non* of retrenchment is that the use of retrenchment should not spoil the rigour achievable via refinement. The best results are obtained when the two notions work closely together, with retrenchment being used to connect together otherwise incompatible refinement strands. The more tightly such different refinement strands are coupled via retrenchment, the more restraint is exercised over retrenchment's otherwise extreme permissiveness.

The paradigmatic arrangement of retrenchments and refinements, that achieves both the tight coupling that restricts retrenchment and the non-interference with the rigour of refinement, is the *Tower Pattern*, an epithet that summarises a host of square completion and factorisation results in Jeske's thesis [10]. The left hand side of Fig. 2 shows a commuting square of retrenchments and refinements among four system models A, B, C, D, with the retrenchments horizontal and the refinements vertical (and the data that characterises these retrenchments and refinements implicit). The results of [10] show that whenever you start with two adjacent sides of such a square, the square can be completed by building the missing system model and its impinging retrenchment and refinement out of the existing elements in a canonical way, and that the result is indeed a commuting square. (Section 5 is concerned with an explicit example of this construction.) Such commuting squares are the fundamental building blocks of the tower, which itself is just an arrangement of such squares into a suitable grid pattern appropriate to the development at hand.

The tower construction has by now had substantial vindication. In the formal development of the Mondex Electronic Purse [11], there were a number of requirements

issues that were handled less than ideally in the formal modelling. These have all been handled convincingly via retrenchment, mostly using the tower [12,13,14].

Although [10] was done in the context of Z refinement to directly serve the needs of the Mondex work, the approach advocated in Section 3 and discussed more widely in [5], ensures that there is a wide commonality between retrenchment formulations for different variants of refinement. The insistence that retrenchment is confined to the initialisation and correctness POs helps here, since most notions of refinement have initialisation and correctness POs that are either identical to, or extremely close to, (5) and (6). In particular, this applies to Event-B and Z refinements, so, since the composition of refinements and retrenchments is defined via their initialisation and correctness POs, these compositions (which yield retrenchments), will be identically defined for both Z and Event-B. See [15].

For our purposes, we need a suitable analogue of the Postjoin Theorem from [10], which states that if we have three systems A, B, C, as in Fig. 2, the square can be completed in a canonical way with a system D, and a connecting retrenchment $Ret_{C,D}$ and refinement $Ref_{B,D}$, so that the two retrenchment+refinement compositions round it are equal. What impedes the immediate application of the theorem from [10] is: (a) its ferocious technical complexity; (b) its detailed Z dependence as regards 'non-correctness' POs (i.e. the POs that in Z replace guard strengthening and relative deadlock freedom). Fortunately the same remedy overcomes both problems. The ferocity of the postjoin construction comes from wanting to deal with the most general situation possible, which means allowing the relations that comprise the given retrenchment and refinement to be as unrestricted as possible. The postjoin construction then has to extract those parts of these constituent entities that compose smoothly, and it attempts to do so in the most general manner achievable — this generates formidable complexity. However, if we are dealing with a situation in which the constituents are well behaved to start with, most of the complexity simplifies drastically, and a situation that is 'obviously sensible' emerges. The same applies to the non-correctness POs; in a well behaved context, these do not cause problems either. Further discussion of these points is best given in the context of an example, so we pick up this thread again near the end of Section 5.

What does any of this have to do with reconciling the TD and UCw strategies? Well, a single step of the UCw strategy takes the pre-existing development, and incorporates a new UseCase of functionality. We can imagine that the pre-existing development has been captured within a sequence of Event-B refinements, starting with the most abstract formulation of the pre-existing functionality, and descending into more concrete levels of description, perhaps aggregating additional events into the description as we go in the usual Event-B way. We can represent this pre-existing development by the thick vertical line in the middle of Fig. 2.

The incorporation of the new functionality may well require the introduction of new events at the top level, a reworking of the top level invariant, reworked top level guards, and so on. As such it will not generally fit into the preceding refinement sequence, not least because the new top level functionality will usually not manipulate the top level state in a skip-like fashion. (These of course are the crucial reasons why one cannot, in general, capture such increments of functionality using Event-B refinements.) However, the new functionality can be related to the existing development via a retrenchment.

(We can say the latter with confidence since we show in [5] that *any* two system models can be related via a retrenchment, the potential vacuousness of such a statement being alleviated by the observation that the various relations that comprise a retrenchment help to *quantify* the difference between the models, as noted above. In a well-controlled situation, such as the introduction of new functionality, the difference between the two models will not be capriciously arbitrary (despite not necessarily conforming to Event-B refinement desiderata), and so the retrenchment between them will, in fact, be able to say quite a lot.)

Depicting the retrenchment from previous top level model to new top level model horizontally, we arrive at the Γ shape given by the solid part of the next piece of Fig. 2.

Now the tower construction can take over, and complete a sequence of refinements from the new top level via the requisite sequence of postjoin square completions, working downwards, as illustrated in the next part of Fig. 2. The new bottom level will be at the right level of abstraction to correspond with the pre-existing bottom level model. Thus one bout of UseCase introduction has been achieved via the tower. Successive bouts follow the same route. In each case we draw up the retrenchment that takes us to the new top level model, and allow the tower to do the rest. Finally, the right hand column of the last bout yields a pristine Event-B development of the full functionality, shown as an even thicker line on the right of Fig. 2.

5 A Simple Case Study

We tackle a toy distributed allocation problem. It is carried out in the way done here only for purposes of illustrating our techniques. In reality, one would only apply the machinery discussed in this paper to significantly more substantial examples.

Elements are to be allocated. At the most abstract level, there is a large (potentially infinite) set, *ASet*, whose elements are to be allocated, and at a low level this is replaced by a much smaller finite subset *DSet*. Also, at low enough levels of abstraction, *ASet* and *DSet* are statically partitioned into $ASet1, ASet2$ and $DSet1, DSet2$ (the latter being subsets of the former) for allocation to two individual agents. These static facts are captured in the context *Ctx*:

CONTEXT *Ctx*
SETS $ASet, DSet, ASet1, ASet2, DSet1, DSet2$
AXIOMS
$axm1 : ASet1 \cup ASet2 = ASet$
$axm2 : ASet1 \cap ASet2 = \varnothing$
$axm3 : DSet \subset ASet$
$axm4 : DSet1 = DSet \cap ASet1$
$axm5 : DSet2 = DSet \cap ASet2$
END

5.1 Four Machines

Below are four machines, *A*, *B*, *C*, *D*, deliberately arranged as in Fig. 2. The left hand column treats only one UseCase, that of allocation. Machine *A*, the most abstract one, simply models the allocation of an element from *ASet* at the global level. Machine *A*

is refined to machine C, in which two agents can allocate from their statically assigned partitions, each agent allocation refining the global allocation event.

MACHINE A
SEES Ctx
VARIABLES x
INVARIANTS $inv1 : x \in \mathbb{P}(ASet)$
EVENTS
  INITIALISATION
    BEGIN $act1 : x := \varnothing$ END
  AddEl
    ANY el
    WHERE $grd1 : el \in ASet - x$
    THEN $act1 : x := x \cup \{el\}$
    END
END

MACHINE B
SEES Ctx
VARIABLES y
INVARIANTS $inv1 : y \in \mathbb{P}(ASet)$
EVENTS
  INITIALISATION
    BEGIN $act1 : y := \varnothing$ END
  AddEl
    ANY el
    WHERE $grd1 : el \in ASet - y$
    THEN $act1 : y := y \cup \{el\}$
    END
  SubEl
    ANY el
    WHERE $grd1 : y \neq \varnothing$
          $grd2 : el \in y$
    THEN $act1 : y := y - \{el\}$
    END
END

MACHINE C
REFINES A
SEES Ctx
VARIABLES $x1, x2$
INVARIANTS $inv1 : x1 \in \mathbb{P}(ASet1)$
           $inv2 : x2 \in \mathbb{P}(ASet2)$
           $inv3 : x = x1 \cup x2$
EVENTS
  INITIALISATION
    BEGIN $act1 : x1 := \varnothing$
          $act2 : x2 := \varnothing$
    END
  *AddEl*1
    REFINES *AddEl*
    ANY el
    WHERE $grd1 : el \in ASet1 - x1$
    THEN $act1 : x1 := x1 \cup \{el\}$
    END
  *AddEl*2
    REFINES *AddEl*
    ANY el
    WHERE $grd1 : el \in ASet2 - x2$
    THEN $act1 : x2 := x2 \cup \{el\}$
    END
END

MACHINE D
REFINES B
SEES Ctx
VARIABLES $y1, y2$
INVARIANTS $inv1 : y1 \in \mathbb{P}(DSet1)$
           $inv2 : y2 \in \mathbb{P}(DSet2)$
           $inv3 : y = y1 \cup y2$
EVENTS
  INITIALISATION
    BEGIN $act1 : y1 := \varnothing$
          $act2 : y2 := \varnothing$
    END
  *AddEl*1
    REFINES *AddEl*
    ANY el
    WHERE $grd1 : el \in DSet1 - y1$
    THEN $act1 : y1 := y1 \cup \{el\}$
    END
  *AddEl*2
    . . . . . .
  *SubEl*1
    REFINES *SubEl*
    ANY el
    WHERE $grd1 : y1 \neq \varnothing$
          $grd2 : el \in y1$
    THEN $act1 : y1 := y1 - \{el\}$
    END
  *SubEl*2
    . . . . . .
END

The right hand column introduces the deallocation UseCase. Machine B is like machine A, except that (aside from variable renaming for clarity) it has a *SubEl* event as well as an *AddEl* one. Machine B is refined to machine D. In machine D, the allocation and deallocation events are refined into their agent-wise counterparts (the ones for agent 2 are just like the ones for agent 1, and so are suppressed to save space). Also machine D introduces the use of $DSet$ and its partition into $DSet1, DSet2$.

Let us consider the relationships between these various machines. The A to C refinement is a normal Event-B refinement, as is the B to D refinement. However there is a

difference between the two. In the A to C refinement, the static set $ASet$ stays the same, whereas in the B to D refinement, we are able to replace $ASet$ by $DSet$. The reason we are able to do this in the case of the B to D refinement but not the A to C refinement is connected with the details of Event-B refinement POs. One of these, the relative deadlock freedom PO, demands that the disjunction of the guards of all the abstract events implies the disjunction of the guards of all the concrete ones. Consider then the state in which all $DSet$ elements have been allocated. If we used $DSet$ instead of $ASet$ in machine C, then, whereas the machine A $AddEl$'s guard would be true (since there are plenty of elements left in $ASet - DSet$) the disjunction of the machine C $AddEl1$ and $AddEl2$ guards would be false (since by definition, $(DSet1 - x1) \cup (DSet2 - x2)$ is empty in this state). So the disjunction of the abstract guards would not imply the disjunction of the concrete ones, and the refinement would fail. The same is not true of the B to D refinement. There, when all the $DSet$ elements have been allocated, the disjunction of the abstract guards is true as before, but now, at the concrete level, even though $AddEl1$ and $AddEl2$ are disabled as in machine C, we have the $SubEl1$ and $SubEl2$ events enabled, so the disjunction of the concrete guards is true as well, and the refinement succeeds.[3]

The relationship from machine A to machine B cannot be an Event-B refinement since machine B's $SubEl$ event manipulates the machine A state in a non-skip manner (and furthermore, the relationship cannot be a converse Event-B refinement since then machine B's $SubEl$ event would not be refined by anything). To capture this relationship we need retrenchment, and the trivial retrenchment $Ret_{A,B}$ that follows will do:[4]

```
RETRENCHMENT Ret_{A,B}
FROM A TO B
SEES Ctx
RETRIEVES ret1 : x = y
EVENTS
  RAMIFICATIONS AddEl
    WITHIN wth1 : true
    CONCEDES con1 : false
    END
END
```

```
RETRENCHMENT Ret_{C,D}
FROM C TO D
SEES Ctx
RETRIEVES ret1 : x1 = y1
          ret2 : x2 = y2
EVENTS
  RAMIFICATIONS AddEl1
    WITHIN wth1 : true
    CONCEDES con1 : false
    END
  RAMIFICATIONS AddEl2
    ... ...
END
```

Alongside $Ret_{A,B}$, we have $Ret_{C,D}$, the retrenchment required to relate machine C to machine D. Note that neither retrenchment needs to say anything about the initialisation events, since they are required to work just as in refinement. $Ret_{C,D}$ looks just as trivial as $Ret_{A,B}$ but in fact it is less so. In the Rodin toolset, there is a convention that when one event refines another, any parameters that are identically named in the two events are in fact equal, and the relevant equalities are automatically factored in to the automated reasoning. We have availed ourselves of a similar convention for retrenchments, and it applies in both $Ret_{A,B}$ and $Ret_{C,D}$. In $Ret_{A,B}$ this has little impact, since the only place

[3] The success can be attributed to the fact that we are using the *weak* relative deadlock freedom PO rather than the strong one (see [3], Deliverable D3). The strong version demands that for *each* abstract event, its guard implies the disjunction of the corresponding concrete guard with all the 'new event' guards. Such a PO would fail here, a circumstance that could be overcome with a more extensive use of retrenchment.

[4] One could introduce syntax to deal with such trivial event retrenchments more succinctly.

where it applies (the parameters of the machine A and machine B *AddEl* events), the assumptions pertaining to the two events' parameters are identical. In $Ret_{C,D}$ however, the same situation is less trivial, since machine C's *AddEl*1 *el* is selected from *ASet* while machine D's *AddEl*1 *el* is selected from *DSet*. If we temporarily rename the parameters in these two events by adding subscripts, the real within relation between the *AddEl*1 events in $Ret_{C,D}$ becomes:

$$el_C = el_D \wedge el_C \in ASet \wedge el_D \in DSet \tag{8}$$

which enforces an additional constraint on el_C. So, despite appearances, the within relation of *AddEl*1 has some real work to do. (Note that a similar thing is silently accomplished in the course of the B to D refinement. And if we had taken name identity even further, and avoided renaming the A/C variables $x, x1, x2$, to the B/D variables $y, y1, y2$, we could have simplified $Ret_{A,B}$ and $Ret_{C,D}$ even more by trivialising the retrieve relations.)

5.2 A, B, C, D and the Tower

Machines A, B, C, D, (and the various retrenchments and refinements that relate them), form a commuting square and an instance of the Postjoin Theorem. It is time to pick up the discussion left over from Section 4 regarding this. If we follow a state element from A through the A to B retrieve relation and then through the B to D joint invariant, we arrive at the same set of possibilities as if we had first gone through the A to C joint invariant and then the C to D retrieve relation, i.e. the relevant relational compositions are equal (a claim easy enough to check by hand in this simple example), and they constitute the retrieve relation for the composed retrenchment. The rest depends on the events. Of these, the initialisations behave straightforwardly of course; assuming the truth of the component initialisation POs enables the truth of the composed initialisation PO to be proved, given the composed retrieve relation.

For the other events, we note that machine A's *AddEl* event is going to be retrenched to both *AddEl*1 and *AddEl*2 in machine D, by tracing the square via B or C. Since *AddEl*1 and *AddEl*2 are so similar, it will be sufficient for us to discuss *AddEl*1 and to leave *AddEl*2 to the reader. To discuss *AddEl*1, we first need the within relation for *AddEl* and *AddEl*1. This can be obtained in one of two ways. One can compose the within relation of the A to B retrenchment with the conjunction of the joint invariant and WITNESS relations[5] of the B to D refinement, or one can compose the joint invariant and WITNESS relations of the A to C refinement with the within relation of the C to D retrenchment. Since the square commutes, these two calculations agree, as they must, and as the reader can check.

The concedes relation for *AddEl* and *AddEl*1 is determined similarly. One way round, the concedes relation of the A to B retrenchment is composed with the joint invariant

[5] In Rodin, when an event and its refinement have different parameters, the refined event has a WITNESS clause to say how any abstract parameters not occurring in the refinement are to be related to the refined ones. This is like the within relation of a retrenchment and goes beyond what is documented in [3] Deliverable D3. See the Rodin User Manual at [9]. When there are no such abstract parameters, the witness relation trivialises.

and witness relations of the *B* to *D* refinement for the before-state and input parameters, and another copy of the *B* to *D* refinement joint invariant is used for the after-state. The other way round, the witness relation and two copies of the joint invariant of the *A* to *C* refinement are composed with the concedes relation of the *C* to *D* retrenchment. Again, either way round the square yields the same result. See [15] for more detailed calculations and proofs regarding the general case.

Altogether, we get the composed retrenchment $Ret_{A,D}$, in which the familiar facts hold for the common *el* parameter of *AddEl* and *AddEl*1:

RETRENCHMENT $Ret_{A,D}$
FROM A TO D
SEES Ctx
RETRIEVES $ret1 : x = y1 \cup y2$
EVENTS
  RAMIFICATIONS *AddEl* TO *AddEl*1
    WITHIN $wth1$: true
    CONCEDES $con1$: false
    END
  RAMIFICATIONS *AddEl* TO *AddEl*2
    ... ...
END

The above sketches a confirmation that machine *D* (which we pulled out of a hat) has the right characteristics to be the desired square completion. In general, when machines are constructed to solve plausible problems, their interrelationships are benign, and it is normally transparent what the square completion should look like, without resorting to the general theory. Benign situations are characterised by the fact that the state (and other) spaces partition into equivalence classes, which the various relations in play treat in an 'all or nothing' manner. In other words, the relations involved and their needed compositions are all *regular* [16]. In such cases one can confidently eschew the forbidding complexity of the results in [10], or as here, their Event-B analogues, and work by hand.

6 Conclusions

Assuming that bringing the correctness achievable using techniques like Event-B to today's 'Agile Methods' would be a good thing, we argued that, in general, Event-B's insistence that levels of abstraction be complete at the point of introduction blocks its ready deployment in such methodologies. Thus, in the specific UCw approach, one might want to introduce new top level events that manipulate the state in non-skip ways, and perhaps to make even more drastic modifications. We then showed that retrenchment, which we reformulated in a manner suitable for Event-B, and for Rodin, could address such situations via the *Tower Pattern*, which we illustrated 'by hand'.

The same technical constructions also enlarge the scope for Event-B to tackle a wider variety of 'real-world' applications. For example, Event-B's insistence that all data types are discrete (at least) inhibits its application in real-world scenarios in which the intrinsic variable types are continuous. Of course in all such cases, the continuous variables must eventually be reduced to discrete ones in order to implement digital controllers, but carrying out the argument to justify this replacement within a retrenchment context allows it to make real contact with the formal development, whereas otherwise,

it would have to be expelled completely from the formal considerations. Other ways in which retrenchment might capture the 'grey areas' surrounding a formal development using Event-B could be easily imagined.

It would of course be desirable to mechanise the technology introduced here. For this, as well as the obvious tool development, it would be necessary to formulate an Event-B version of the theorems of [10]. This would need to focus on the useful cases of the tower in a manner that allowed for ready mechanisation of the whole square completion process. These aspects remain as work for the future.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.R.: Event-B (to be published)
3. Rodin. European Project Rodin (Rigorous Open Development for Complex Systems) IST-511599, `http://rodin.cs.ncl.ac.uk/`
4. Banach, R., Poppleton, M.: Retrenchment: An Engineering Variation on Refinement. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 129–147. Springer, Heidelberg (1998)
5. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. 67, 301–329 (2007)
6. Banach, R., Fraser, S.: Retrenchment and the BToolkit. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 203–221. Springer, Heidelberg (2005)
7. Fraser, S., Banach, R.: Configurable Proof Obligations in the Frog Toolkit. In: Proc. Fifth IEEE International Conference on Software Engineering and Formal Methods, pp. 361–370. IEEE Computer Society Press, Los Alamitos (2007)
8. Fraser, S.: Mechanized Support for Retrenchment. PhD thesis, School of Computer Science, University of Manchester (2008)
9. The Rodin Platform, `http://sourceforge.net/projects/rodin-b-sharp/`
10. Jeske, C.: Algebraic Integration of Retrenchment and Refinement. PhD thesis, University of Manchester (2005)
11. Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory (2000)
12. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Retrenching the Purse: Finite Sequence Numbers, and the Tower Pattern. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 382–398. Springer, Heidelberg (2006)
13. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: Finite Exception Logs, and Validating the Small. In: Proc. IEEE/NASA SEW30-06, pp. 234–245 (2005)
14. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties. In: Proc. IEEE ISOLA-06 (to appear, 2006)
15. Banach, R., Jeske, C., Poppleton, M.: Composition Mechanisms for Retrenchment. J. Log. Alg. Prog. 75, 209–229 (2008)
16. Banach, R.: On Regularity in Software Design. Sci. Comp. Prog. 24, 221–248 (1995)

Towards Modelling Obligations in Event-B

Juan Bicarregui[1], Alvaro Arenas[1], Benjamin Aziz[1],
Philippe Massonet[2], and Christophe Ponsard[2]

[1] e-Science Centre, STFC Rutherford Appleton Laboratory, UK
[2] Centre of Excellence in Information and Communication Tech. (CETIC), Belgium
{j.c.bicarregui,a.e.arenas,b.aziz}@rl.ac.uk, {phm,cp}@cetic.be

Abstract. We propose a syntactic extension of Event-B incorporating a limited notion of obligation described by triggers. The trigger of an event is the dual of the guard: when a guard is not true, an event must not occur, whereas when a trigger is true, the event must occur. The obligation imposed by a trigger is interpreted as a constraint on when the other events are permitted. For example, the simplest trigger *next*, which states that the event must be the next one to be executed when the trigger becomes true, is modelled as an extra guard on each of the other events which prohibits their execution at this time. In this paper we describe the modelling of triggers in Event-B, and analyse refinement and abstract scheduling of triggered events.

1 Introduction

In Event-B, a system is defined as a *state* consisting of a set of *variables* and some *events* that cause the state to change by updating the values of the variables as defined by the *generalised substitution* of the event.

Each event is guarded by some condition, which when satisfied implies that the event is permitted in the current state. However, the guard is not an obligation to perform the event as an event may be delayed as a result of, for example, the interleaving with other permitted events. The choice to schedule permitted events is made non-deterministically.

In this paper, we introduce a dual of guards which we call *triggers*. The trigger of an event expresses an *obligation* on when the event must be executed. This is useful in a number of modelling situations, for example to ensure that if a request for a service is made, then the service will eventually be delivered, or that if a hazard state is encountered an alarm will be promptly raised. Often there is a caveat to the obligation, for example, if the receiver remains ready to receive the service, or if the alarm system is in working order.

Triggers model such obligations as constraints on when other events are permitted. For example, the simplest trigger is *next* which states that the event must be the next one executed when the trigger becomes true. This is in effect an extra guard on each of the other events which prohibits their execution at this time.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 181–194, 2008.

Event-B also incorporates a refinement methodology which is used to incrementally develop a model of the system. Our model of triggers enables the abstract specification of certain constraints on the ordering of events. In refinement, further constraints can be added as these reduce the non-determinism inherent in the choice of events. Thus triggers can be strengthened in refinement.

This work is motivated by a desire to close the gap between requirements and specifications, in particular, when using the KAOS goal-oriented requirements methodology for describing requirements [15] and Event-B for specifying software systems. In KAOS, system goals are refined into requirements under the responsibility of agents. Agents performs operations in order to fulfill requirements. Operations are specified by pre- and post-conditions, which represent state transitions in the usual way, and a trigger condition, which captures an obligation to perform the operation when the condition becomes true provided the domain precondition is true.

The structure of the paper is as follows. Triggers are introduced to Event-B in section 2. The use of triggers is demonstrated and compared with classical Event-B on a motivating exemplar in section 3. The way obligations are interpreted in Event-B is fully described in section 4. Refinement with triggers is discussed in section 5. Then, section 6 discusses abstract schedulability, as triggers introduce constraints on the order of event which may introduce deadlocks for which extra proof obligations are required. Section 7 explores some related work. Finally the paper summarises main results and highlights future work in section 8.

2 Events in Event-B

An Event-B model describes number of *events* which manipulate the state. Events are defined by the following syntax:

$$ev ::= \quad \textbf{EVENT}\ e\ \textbf{WHEN}\ G\ \textbf{THEN}\ S\ \textbf{END}$$

Where G is the guard, expressed as a first-order logical formula in the state variables, and S is the generalised substitution, defined by the syntax of Figure 1.

$S ::=$	$\textbf{SKIP}$	Do nothing
	$\mid\ x := E(var)$	Deterministic substitution
	$\mid\ \textbf{ANY}\ t\ \textbf{WHERE}\ P(t, var)$	
	$\textbf{THEN}\ x := F(t, var)\ \textbf{END}$	Non-deterministic substitution
	$\mid\ S \parallel S'$	Parallel substitution

Fig. 1. The syntax of generalised substitutions

For a comprehensive description of the Event-B language and its formal meaning, we refer the reader to more detailed references such as [14].

2.1 Events with Triggers

As mentioned above, in standard Event-B systems the next event to be executed is chosen non-deterministically from all those whose guards are true. If a particular order of execution of events is required this must be described explicitly through the guards by incorporating any required flags or other protocol in the model of the state In this paper, we introduce the ability to implicitly model a particular form of obligation, which we believe gives some of the benefits of richer expressibility without changing the underlying semantic framework of Event-B.

For this purpose, we introduce a new syntactic construct to Event-B which we define within the standard semantics by extending the model. This syntactic sugar provides a way to abstractly describe requirements on the order of execution of events without explicitly detailing a model of how the scheduling is performed. Methodologically, triggers have the advantage of associating with each event any obligation as to when it is performed, but the disadvantage is that they implicitly impose constraints on other events which may be unwelcome.

Note that the obligation imposed by a trigger is similar to a partial correctness guarantee: it ensures that *if* something happens, it will be the right thing but it does not guaranteed that anything will happen at all. That is, it does not guarantee that the system will not deadlock.

The new construct replaces the guard with a trigger and is indicated by changing the **THEN** keyword to **NEXT**. In the simplest case it forces the event to be the next event which happens

$$ev ::= \quad \textbf{EVENT}\ e\ \textbf{WHEN}\ T\ \textbf{NEXT}\ S\ \textbf{END}\ .$$

A weaker form requires the event to happen some time in the future

$$ev ::= \quad \textbf{EVENT}\ e\ \textbf{WHEN}\ T\ \textbf{EVENTUALLY}\ S\ \textbf{END}\ .$$

Both of these are special cases of the **WITHIN** construct which gives an upper bound to the number of other events which may occur before the triggered event

$$ev ::= \quad \textbf{EVENT}\ e\ \textbf{WHEN}\ T\ \textbf{WITHIN}\ n\ \textbf{NEXT}\ S\ \textbf{END}$$

where n is zero for **NEXT** and n is unboundedly non-deterministically chosen for **EVENTUALLY**.

The above syntax introduces a trigger condition, T, into the specification of an event. This condition is a predicate on states which defines those states which, if reached, oblige a particular behaviour to follow. This behaviour can be seen as a bounded form of the `leads-to` modality [12]. Let $\Box$ denote the *always* temporal operator and $\Diamond$ denote the *eventually* operator. Given a certain predicate P defined on the states variables of an evolving system, then $\Box P$ means that P always holds whatever the evolution of the system. The statement $\Diamond P$ means that P holds at system start up or that it will certainly hold after system start up whatever the evolution of the system. Given predicates P and Q, the statement P `leads-to` Q means that it is always the case that once P holds then Q

holds eventually, which is formalised as $\Box(P \Rightarrow \Diamond Q)$. Our triggered events are modelling the behaviour $\Box(P \Rightarrow \Diamond_{\leq n}(P \Rightarrow Q))$, meaning that once P holds then $(P \Rightarrow Q)$ will occur before at most n time units. So the **WITHIN** event introduced above models the behaviour $\Box(T \Rightarrow \Diamond_{\leq n}(T \Rightarrow e))$.

Our triggers model a class of queuing behaviour which are common in resource management but also occur in other situations such as in telephone services or ticket controlled supermarket counters. If on entering the queue, the requester is ready to be served, and thereafter remains ready to be served, the service will eventually be delivered. But if the requester leaves the queue, the request is cancelled. In the next section, we introduce a very simple example to motivate the most basic form of such requirement, when the service must be delivered in the next cycle.

3 Motivating Example

We illustrate the use of triggers with a very small example, which although simplified to the point of triviality still illustrates some of the advantages and disadvantages of triggers. The example, taken from [11], is about the sump in a mine which is used to control the drain water out of the main areas. In this system, water seeps into the sump from the mine and the level of water is kept within bounds by operating a pump. Additionally, an alarm must be immediately sounded if methane is detected in the sump. The requirements on the system are as follows:

1. The pump must be activated when the water level reaches a high water sensor in order to keep the mine dry.
2. The pump must be deactivated when the water level reaches a low water sensor in order to avoid the pump running dry which would damage it.
3. If methane is detected in the sump then the pump must be deactivated immediately in order to avoid the risk of explosion, and an alarm sounded in the main areas in order to warn of the eventual risk of flooding.

A partial specification in Event-B is given in Figure 2 in two versions, one using triggers and the other without. Note how the use of the trigger allows the specification of the events to closely reflect the description of the problem given in the requirements in that it captures the immediacy specified in the third requirement alongside other aspects of that requirement. On the other hand, it has the disadvantage of being rather implicit. In order to fully understand the behaviour of the pump, the reader will now have to consider the specification of the methane event.

Note also how the conflict between requirements 1 and 3, in the case where high water and methane are both detected, is handled. The extra guard in the event which switches the pump on ensures that requirement 3 is met. We will show in section 4.1 that the two specifications are equivalent by definition.

In this simple example, both versions can be easily created and understood but as we will see later, the situation is not so simple when there are several

```
INVARIANTS
  lowwater : Bool
  highwater : Bool
  methane : Bool
  pump : {ON, OFF}
  bell : {ON, OFF}

EVENTS
high_water_detected
  WHEN highwater = true
  THEN pump := ON
  END

low_water_detected
  WHEN lowwater = true
  THEN pump := OFF
  END

methane_detected
  WHEN methane = true
  NEXT pump := OFF || bell := ON
  END
```

```
INVARIANTS
  lowwater : Bool
  highwater : Bool
  methane : Bool
  pump : {ON, OFF}
  bell : {ON, OFF}

EVENTS
high_water_detected
  WHEN highwater = true
      AND not (methane = true)
  THEN pump := ON
  END
low_water_detected
  WHEN lowwater = true
      AND not (methane = true)
  THEN pump := OFF
  END
methane_detected
  WHEN methane = true
  THEN pump := OFF || bell := ON
  END
```

Fig. 2. A simple example with and without the use of triggers

more complex timing requirements. In these situations, triggers can be used to raise the level of abstraction by formalising requirements concerning the order of execution of events without explicitly elaborating a model which exhibits them.

4 The Interpretation of Triggered Events

4.1 NEXT Events

Let us consider the interpretation of triggers for the simplest case, that is, when a trigger forces an event to be the next one executed. Consider a system with two events e and f, as shown in the upper box of Figure 3.

EVENT e **WHEN** G **THEN** S **END**
EVENT f **WHEN** T **NEXT** R **END**

EVENT e **WHEN** $G \wedge \neg T$ **THEN** S **END**
EVENT f **WHEN** T **THEN** R **END**

Fig. 3. Simple case of **NEXT** trigger and its interpretation

In this case, whenever T become true, then e must be prohibited so that the only remaining possibility is that f is the next event, representing the obligation $\Box(T \Rightarrow \circ f)$, where $\circ$ denote the *next* temporal operator. This can be modelled by extending the guard on e with the negation of T as shown in the lower box of Figure 3. Thus the trigger in f can be considered as a syntactic sugar for an extra guard on e which ensures that e will be disabled when trigger T is true. It is clear that if G is always false when T is false, that is if $G \Rightarrow T$, then the un-triggered event will never be executed.

The case where there are several triggered events is given in Figure 4. Here all other events must be disabled when any trigger becomes true so if more than one trigger becomes true simultaneously, the machine will be "deadlocked".

EVENT e_1 **WHEN** G_1 **THEN** S_1 **END**
EVENT e_2 **WHEN** G_2 **THEN** S_2 **END**
EVENT f_1 **WHEN** T_1 **NEXT** R_1 **END**
EVENT f_2 **WHEN** T_2 **NEXT** R_2 **END**

EVENT e_1 **WHEN** $G_1 \wedge \neg T_1 \wedge \neg T_2$ **THEN** S_1 **END**
EVENT e_2 **WHEN** $G_2 \wedge \neg T_1 \wedge \neg T_2$ **THEN** S_2 **END**
EVENT f_1 **WHEN** $T_1 \wedge \neg T_2$ **NEXT** R_1 **END**
EVENT f_2 **WHEN** $T_2 \wedge \neg T_1$ **NEXT** R_2 **END**

Fig. 4. The interpretation of **NEXT** triggers as extra guards on other events

This will show up in the following deadlock-freeness condition which must be shown alongside the usual one that at least one guard (or trigger) must always be true.

Law 1 (Deadlock-free of NEXT events) *Let M be a system with k next events **EVENT** e_i **WHEN** T_i **NEXT** S_i, for $i = 1 \cdots k$. System M is deadlock-free with relation to its **NEXT** events if all the trigger conditions associated with the **NEXT** constructor are pairwise disjoint, i.e. $\neg(T_i \wedge T_j)$ for $i \neq j$.*

A more general discussion of this law, including the general form of this proof obligation is presented in section 6.

4.2 WITHIN Events

A generalisation of the **NEXT** constructor is the **WITHIN** constructor. In this case, if the trigger becomes true the triggered event must be executed before at most n other events are executed (provided the trigger remains true). If the trigger becomes false within these n steps, the obligation is cancelled.

Again let us consider the simple case of a system with just two events, one of them being a triggered one, as shown in the upper box of Figure 5.

To "un-sugar" this system, as shown in lower box of Figure 5, we must extend the state with a counter for event f. We add an integer valued $counter_f$,

EVENT e **WHEN** G **THEN** S **END**
EVENT f **WHEN** T **WITHIN** n **NEXT** R **END**

$Inv \triangleq \ldots \wedge 0 \leq counter_f \leq n$
$Init \triangleq \ldots \parallel counter_f := n$
EVENT e **WHEN** $G \wedge (\neg T \vee counter_f > 0)$ **THEN**
$S \parallel counter_f := ((counter_f - 1) \triangleleft T \triangleright n)$ **END**
EVENT f **WHEN** T **THEN** $R \parallel counter_f := n$ **END**

Fig. 5. Simple case of **WITHIN** trigger and its interpretation

which is set with the value n whenever T becomes true, and is decremented each time e is executed whilst T remains true. Here, we borrow the conditional operator from UTP [9], and write $x := exp1(var) \triangleleft b(var) \triangleright exp2(var)$ to denote the substitution **ANY** z **WHERE** $((b(var) \Rightarrow z = exp1(var)) \wedge (\neg b(var) \Rightarrow z = exp2(var)))$ **THEN** $x := z$ **END** . If $counter_f$ reduces all the way to zero, then e becomes disabled and consequently f becomes obliged. If T becomes false while the counter is active, then it is reset to n. Here we are modelling obligation $\Box(T \Rightarrow \Diamond_{\leq n} f)$, which corresponds to a bounded version of the `leads-to` modality.

The case where there are several triggered events is given in Figure 6. Here the state is extended with a counter for each triggered event and each event is extended with extra clauses in the guards and substitutions to manipulate these counters.

It is clear that this model will quickly become quite complex if there are several triggers. In fact the analysis of deadlock for such systems is not trivial as we will see in section 6.

The **NEXT** trigger corresponds to the particular case of **WITHIN** with n equal to 0.

Theorem 1 (Relation between NEXT and WITHIN). *The event* ***EVENT*** e ***WHEN*** T ***NEXT*** R *is equivalent to the event* ***EVENT*** e ***WHEN*** T ***WITHIN*** 0 ***NEXT*** R.

The proof which is omitted relies on the counter being always zero.

4.3 Events with Guards and Triggers

So far, our examples of triggered events have not included guards. We have interpreted this as the guard being the same as the trigger, that is the event is triggered exactly when it is permitted. Another possibility is that the guard of the triggered event is always true. In the most general case, a triggered event can have specified a guard indicating the states in which the event is permitted, as well as a trigger indicating when it is obliged

$ev ::=$ **EVENT** e **WHEN** $(Trigger\, T, Guard\, G)$ **WITHIN** n **NEXT** S **END**

$$\begin{array}{l}\textbf{EVENT } e_1 \textbf{ WHEN } G_1 \textbf{ THEN } S_1 \textbf{ END}\\ \textbf{EVENT } e_2 \textbf{ WHEN } G_2 \textbf{ THEN } S_2 \textbf{ END}\\ \textbf{EVENT } f_1 \textbf{ WHEN } T_1 \textbf{ WITHIN } n_1 \textbf{ NEXT } R_1 \textbf{ END}\\ \textbf{EVENT } f_2 \textbf{ WHEN } T_2 \textbf{ WITHIN } n_2 \textbf{ NEXT } R_2 \textbf{ END}\end{array}$$

$$\begin{array}{l} Inv \triangleq \ldots \wedge 0 \leq counter_{f_1} \leq n_1 \wedge 0 \leq counter_{f_2} \leq n_2\\ Init \triangleq \ldots \parallel counter_{f_1}, counter_{f_2} := n_1, n_2\\ \textbf{EVENT } e_1 \textbf{ WHEN } G_1 \wedge (\neg T_1 \vee counter_{f_1} > 0) \wedge (\neg T_2 \vee counter_{f_2} > 0) \textbf{ THEN}\\ \quad S_1 \parallel counter_{f_1} := ((counter_{f_1} - 1) \triangleleft T_1 \triangleright n_1)\\ \qquad \parallel counter_{f_2} := ((counter_{f_2} - 1) \triangleleft T_2 \triangleright n_2) \textbf{ END}\\ \textbf{EVENT } e_2 \ldots\\ \textbf{EVENT } f_1 \textbf{ WHEN } T_1 \wedge (\neg T_2 \vee counter_{f_2} > 0) \textbf{ THEN}\\ \quad R_1 \parallel counter_{f_1} := n_1\\ \qquad \parallel counter_{f_2} := ((counter_{f_2} - 1) \triangleleft T_2 \triangleright n_2) \textbf{ END}\\ \textbf{EVENT } g_1 \textbf{ WHEN } T_2 \wedge (\neg T_1 \vee counter_{f_1} > 0) \textbf{ THEN}\\ \quad R_2 \parallel counter_{f_2} := n_2\\ \qquad \parallel counter_{f_1} := ((counter_{f_1} - 1) \triangleleft T_1 \triangleright n_1) \textbf{ END}\end{array}$$

Fig. 6. The interpretation of **WITHIN** triggers introduces a counter for each trigger

In this case the following healthiness condition, which relates triggers and guards would apply: if an event is obliged, then surely it must be permitted.

Definition 1 (Well formedness of triggers). *For all events* ***EVENT*** e ***WHEN*** $(Trigger\ T, Guard\ G)$ ***WITHIN*** n ***NEXT*** S ***END*** *, we have that* $T \Rightarrow G$.

On the other hand, the classical definition of an event in Event-B corresponds to an event with false trigger.

Theorem 2 (Relating un-triggered and triggered events). *The event* ***EVENT*** e ***WHEN*** G ***THEN*** R *is equivalent to* ***EVENT*** e ***WHEN*** $(false, G)$ ***WITHIN*** n ***NEXT*** R*, where* n *is any arbitrary integer greater than or equal to 0.*

The proof is omitted for brevity.

4.4 EVENTUALLY Events

The unbounded case, described by **WHEN** T **EVENTUALLY** S, is modelled by **WITHIN** with an unbounded non-deterministic choice of n. Note that in this approach, the choice of n is made when the trigger becomes true and so the deadline would be set at that time although it would be only known internally.

5 Refinement with Triggers

Refinement allows one to build a model incrementally by making it more and more precise, that is closer to the reality. In this section we analyse refinement with

triggers. We use notation $e \sqsubseteq f$ to indicate that abstract event e is refined by concrete event f, meaning that feasibility, guard and invariant refinement laws hold between e and f, as stated in the Event-B manual [14, pp. 11, Fig. 20].

5.1 Refinement of Duration

It is clear that the addition of triggers to a system restricts its possible behaviours by strengthening its guards and so constitutes a refinement of that system. This is formalised in the theorem below. On the other hand, it may of course introduce the possibility of deadlock which is considered in the next section.

Theorem 3 (Refinement of duration). *Let P be a predicate on states, S be a substitution and let $0 \leq n \leq m$ be integers. Then we have:*

$$\begin{array}{ll} & \textbf{\textit{EVENT}}\ e\ \ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{EVENTUALLY}}\ S \\ \sqsubseteq & \textbf{\textit{EVENT}}\ e_1\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{WITHIN}}\ m\ \textbf{\textit{NEXT}}\ S \\ \sqsubseteq & \textbf{\textit{EVENT}}\ e_2\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S \\ \sqsubseteq & \textbf{\textit{EVENT}}\ e_3\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{NEXT}}\ S \end{array}$$

5.2 Refinement of the Trigger Predicate

As mentioned above, guards can be strengthened in refinement and so, by duality, we would expect that triggers can be weakened in refinement[8]. To motivate this, consider the abstract obliged behaviours. These are a minimal set of behaviours necessary for the requirement to be satisfied. During refinement we would expect to ensure that the set of obliged behaviours does not decrease as this could invalidate a requirement.

This can also be understood mechanistically as the trigger is interpreted by adding its negation to the other guards, weakening a trigger is in effect strengthen ing the other guards.

Theorem 4 (Refinement of trigger predicates). *Let M be a system deadlock-free of* ***NEXT*** *events (as defined in Law 1), which includes abstract event* $\textbf{\textit{EVENT}}\ e_a\ \textbf{\textit{WHEN}}\ T_a\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S$. *If system M is deadlock-free of* ***NEXT*** *events when event e_a is replaced by event* $\textbf{\textit{EVENT}}\ e_c\ \textbf{\textit{WHEN}}\ T_c\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S$ *and $T_a \Rightarrow T_c$, then we have that*

$$\begin{array}{ll} & \textbf{\textit{EVENT}}\ e_a\ \textbf{\textit{WHEN}}\ T_a\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S \\ \sqsubseteq & \textbf{\textit{EVENT}}\ e_c\ \textbf{\textit{WHEN}}\ T_c\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S \end{array}$$

Proof. The proof is straightforward by the usual refinement of guards.

5.3 Removing Triggers

From the above we see that we have $T_a \Rightarrow T_c \Rightarrow G_c \Rightarrow G_a$. That is, during refinement, triggers will get ever closer to guards. There are three limiting cases. A false trigger is the degenerate case where nothing is obliged and the specification

reverts to a standard Event-B semantics. A true trigger means that the event is always obliged and may therefore block the execution of any other event. The third limiting case is when the trigger becomes equal to the guard. At this point there is no choice left and the permitted behaviours are equal to the obliged ones. Depending on the form of the obligation we have modelled and type of concurrency in the system, this may mean that only one event can execute at any given time and therefore that we have, in effect, partitioned the states by the possible events.

5.4 Implementing Triggers

We have seen how the definition of refinement can be extended to incorporate triggers and how this ensures that obligations are preserved during refinement. However, it is still required, at the end of the refinement process, to ensure that the most concrete specification does indeed implement the triggers and so satisfies the obliged behaviors all the way back up the refinement chain.

Whilst the usual refinement process will ensure the model developed implicitly using triggers is necessarily correct in this sense, it is perhaps unlikely that this model will yield a satisfactory implementation. So we expect to have to build into the implementation a mechanism for scheduling the events which has the desired properties. This concrete model, which itself will have no triggers, is then shown to be correct against the triggered version in the usual way. Thus we do not expect to allow triggers in an implementation but instead develop a model ourselves which implements the required behaviour.

6 Scheduling

The interpretation of triggered events with counters in Event-B is an example of the inclusion of abstract scheduling in a specification, as advocated in [2]. Such techniques have been used in the past for modelling dynamic contraints in B [1] or to specify abstract scheduling of real-time programs [3].

In this section we consider the scheduling of triggered events and develop a sufficient condition for schedulablity. We begin with the case where an event is triggered immediately.

6.1 Deadlock Freeness for NEXT

As stated earlier it is clear that the system will deadlock if two "**WITHIN** 0" triggers become true at any one time. Let us define an active counter to be one whose corresponding trigger is currently true, then it is clear that there must be at most one active counter whose value is equal to zero.

Definition 2 (Active Counter). *For all events,*
EVENT e ***WHEN*** T ***WITHIN*** n ***NEXT*** S*, we say that* e *has an* **active counter** *if* T *is true in the current state.*

Definition 3 (Deadlock-free for NEXT). *A system is deadlock-free for* ***NEXT*** *if at all times there is at most one active counter whose value is equal to zero.*

This condition must be true in addition to the usual condition that at least one guard is true to ensure that the system is not currently in deadlock. It is slightly more general than the disjointness of triggers for next events given earlier as it also requires that any "**WITHIN** n" event which may have been triggered earlier does not clash with a "**WITHIN** 0" event just triggered.

6.2 Schedulability

To generalise the above notion of deadlock for triggered events with non-zero counters, we develop some properties related to the schedulability of triggered events.

Definition 4 (Schedulability of a WITHIN event).
Event ***EVENT*** e ***WHEN*** T ***WITHIN*** n ***NEXT*** S *is schedulable if whenever* T *becomes true, there are at most* n *other active counters whose value is less than or equal to* n*.*

This says that whenever a "**WITHIN** n" event is triggered, there are not too many other events already triggered for the next $n+1$ slots. This is not actually a sufficient condition to guarantee that the system will not deadlock within this period as it is possible that more events with shorter within clauses will be triggered whilst this counter is active. Neither is it a necessary condition, as some triggers which are currently true may become false before their event is executed and therefore liberate some of the slots. It simply states that as far as we can tell at the moment, it is not going to be impossible to schedule this event.

Definition 5 (Schedulability of a system with WITHIN events). *An event system is schedulable if all its* ***WITHIN*** *events are schedulable at all times.*

Schedulability is not easy to prove in general, as it is not at all easy to characterise which counters are active in any given state as this depends on the history of the trace to this point, that is, on which triggers have been true in the past.

Given the above definitions, however, we can give the following characterisation of schedulability.

Theorem 5 (System schedulability). *An event system with* ***WITHIN*** *events is schedulable iff at all times, for all* n*, there are at most* $n+1$ *active counters whose value is less than or equal to* n*.*

This condition can be considered to be an invariant of a well defined system and can therefore be added as an extra proof obligation for each event which, if true, ensures that the system is deadlock-free. It states that, for any execution

of any event, if the system is schedulable beforehand, it must still be schedulable afterwards. This then becomes an inductive prove of deadlock-freeness.

Note that we have assumed that the events meet the healthiness condition given earlier, that is, that the guards are true whenever the triggers are true. This is necessary so that if the scheduling of events requires that an event will be next, we can be sure that it is permitted at this point. The healthiness condition ensures this since, if the guard were false, then so would be the trigger, and so the counter would become inactive and the event removed from the queue.

7 Related Work

There has been several proposals to model obligations in event-based languages like B. In their seminal paper on dynamic constraints in B [1], Abrial and Mussat propose modelling the `leads-to` and the `until` modalities in B. Given P and Q state predicates, the leads-to modality $\Box(P \Rightarrow \Diamond Q)$ means that it is always the case that once P holds then Q holds eventually. They model this modality, for a particular set of events, as a loop which is selected when condition P holds and then iterates, executing one of the events until condition Q becomes true. A variant condition guarantees termination of the loop. By contrast, our triggered events model a bounded leads-to modality $\Box(P \Rightarrow \Diamond_{\leq n} Q)$, which means that once that once P holds then Q will occcur before at most n other events. So, Abrial and Mussat's model can be seen as a general case of our trigger model, but there are some importance differences. For our triggered events, P is an additional predicates on states and Q is the generalised substitution of the event. When P becomes true, a counter is set running which ensures that no more than n *other* events can occur before the triggered event is executed.

In [13], Méry and Merz propose an event language with deontic concepts such as permissions, right and obligations, and develop a stepwise refinement method. Their approach is close to ours in that their notion of obligation corresponds to our trigger condition: a predicate associated to an event indicating the liveness property that when the predicate is true it may lead to the occurrence of the associated event. However, we interpret our triggers through an extension of the usual event-B model rather than introducing a more complex semantic framework.

In [8], Fiadeiro and Maibaum propose a relationship between deontic logic structures, which use the notions of permissions and obligations, and temporal logic through the definition of a consequence operator. This relationship then permits the derivation of normative behaviours of systems, which could include both safety and liveness properties, as well as the reasoning on the relationship between normative states and normative trajectories that could lead to non-normative states, e.g. the performing of permitted actions that lead to obligations that cannot be fulfilled. Our work could be considered as the application of one aspect of their framework, namely the description of a particular class of deontic property, to Event-B systems.

Other attempts to deal with liveness properties in B include [5], which presents a proposal of specification and proof of liveness properties in Event-B. Here proof obligations are defined in terms of weakest preconditions, inspired by the UNITY logic.

Our work is also related to extensions of Event-B to deal with real-time. In [6], the authors present a refinement method that allows refined events to be guarded by time constraints using the concept of active times. The main difference from our current work is that active times are a form of guards and thus do not express any obliged behaviour. Colin *et al.* describe in [7] the alternative approach of extending the semantic model of B with the duration calculus in order to deal with real-time issues.

More recent work on CSP‖B allows designers to add *control flow annotations* to machine operations [10]. One of their possible annotations is NEXT which introduces the set of operations that should be enabled after an operation is executed. There is an interesting relation between this annotation and our. Interpretation of our **NEXT** event can correspond in some cases to annotating *other* events with NEXT annotations although this correspondence is not straightforward. However, we focus on identifying circumstances when an event will be executed next, rather that defining directly the order in which events must occur.

8 Conclusion

This paper has presented a syntactic extension to Event-B to model the notion of obligation throughout the use of triggers. The obligation imposed by a trigger is interpreted as a constraint on when other events can be permitted. We have analysed issues related to the refinement and schedulability of triggered events.

There are some limitations in our proposal that we plan to address as future work. One restriction is related to the abstract scheduling of events through counters, which could make it difficult to incorporate other scheduling policies into the model. One potential solution could be the use of the VARIANT clause in the model, as advocated in [1]. There are some open questions in relation to eventuality and scheduling, since the selection of n must not lead to deadlock. We also plan to develop a more complete proof method for Event-B with obligations, which will allow one to proof event properties without need to expand events into the classical Event-B.

As mentioned before, our motivation is to link KAOS requirements with Event-B specifications. Triggered events as presented here are suitable for modelling the KAOS *achieve* pattern [16]; we would like to investigate the representation of other modalities as events, so that we can model other KAOS patterns such as *maintain* and *cease*. Finally, we would like to model and reason about obligation policies in our framework. Initial work on this line has been reported in [4].

Acknowledgement

This work is funded by the European Commission under the FP6 IST project GridTrust (project reference number 033817).

References

1. Abrial, J.R., Mussat, L.: Introducing Dynamic Constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393. Springer, Heidelberg (1998)
2. Apt, K.R., Olderog, E.-R.: Proof Rules and Transformations Dealing with Fairness. Science of Computer Programming 3(1), 65–100 (1983)
3. Arenas, A.E.: An Abstract Model for Scheduling Real-Time Programs. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 204–215. Springer, Heidelberg (2002)
4. Arenas, A.E., Aziz, B., Bicarregui, J.C., Matthews, B.: Managing Conflicts of Interests in Virtual Organisations. In: STM 2007, ERCIM Workshop on Security and Trust Management. Electronic Notes in Theoretical Computer Science, vol. 197, pp. 45–56. Elsevier, Amsterdam (2008)
5. Ruíz Barradas, H., Bert, D.: Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335. Springer, Heidelberg (2002)
6. Cansell, D., Mery, D., Rehm, J.: Time Constraint Patterns for Event B Development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 140–154. Springer, Heidelberg (2006)
7. Colin, S., Mariano, G., Poirriez, V.: Duration Calculus: A Real-Time Semantic for B. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 431–446. Springer, Heidelberg (2005)
8. Fiadeiro, J., Maibaum, T.: Temporal Reasoning over Deontic Specifications. Journal of Logic Computation 1(3), 357–395 (1991)
9. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall Series in Computer Science (1998)
10. Ifill, W., Schneider, S., Treharne, H.: Augmenting B with Control Annotations. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 34–48. Springer, Heidelberg (2006)
11. Joseph, M.: Real-Time Systems: Specification, Verification and Analysis. Prentice Hall International, Englewood Cliffs (1996)
12. Manna, Z., Pnueli, A.: The Reactive Behavior of Reactive and Concurrent System. Springer, Heidelberg (1992)
13. Méry, D., Merz, S.: Event Systems and Access Control. In: Gollmann, D., Jürjens, J. (eds.) 6th Intl. Workshop Issues in the Theory of Security, Vienna, Austria, pp. 40–54. IFIP WG 1.7, Vienna University of Technology (2006)
14. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language. Rodin Deliverable D3.2 (2005)
15. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Fifth IEEE International Symposium on Requirements Engineering (2001)
16. van Lamsweerde, A., Letier, E.: Deriving Operational Software Specifications from System Goals. In: Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, pp. 119–128. ACM, New York (2002)

A Practical Single Refinement Method for B

Steve Dunne and Stacey Conroy

School of Computing, University of Teesside
Middlesbrough, TS1 3BA, UK
s.e.dunne@tees.ac.uk

Abstract. We propose a single refinement method for B, inspired directly by Gardiner and Morgan's longstanding single complete rule for data refinement, and rendered practical by application of the current first author's recent first-order characterisation of refinement between monotonic computations.

1 Introduction

In this paper we describe a method for verifying arbitrary refinements between B machines, in the absence of unbounded nondeterminism, in a single step rather than having to find an intermediate backward refinement of the "abstract" machine which is itself then forward-refined by the "concrete" machine. The idea of a single complete refinement rule is by no means new: such a rule for data refinement in a predicate-transformer setting was described as long ago as 1993 by Gardiner and Morgan [11], and it is indeed fundamentally their idea which we exploit in this paper. Gardiner and Morgan themsleves appear to have regarded their rule as of theoretical interest only; it seems they didn't seek to exploit it in practice. We will show that a slightly extended version of B provides a suitable setting for practical exploitation of Gardiner and Morgan's rule. Like Gardiner and Morgan we interpret a computation as a weakest-precondition (wp) predicate transformer from sets of final states to sets of starting states [6], and call it *monotonic* if its corresponding wp predicate-transformer is monotonic. A monotonic computation can exhibit both demonic and angelic nondeterminism. *Conjunctivity* and *disjunctivity* are special cases of monotonicity: a conjunctive computation can exhibit only demonic nondeterminism, while a disjunctive computation can exhibit only angelic nondeterminism [7].

Our contribution here is the formulation of a pair of simple first-order proof obligations for verifying refinements between monotonic computations, which renders such verifications amenable to mechanisation in a similar way to that which B already uses for refinements between conjunctive computations [1].

The remainder of the paper is structured as follows: in Section 2 we describe Gardiner and Morgan's single complete rule for data refinement and in Section 3 we take some mathematical insight from [2] to explain why an arbitrary data refinement can always be "factored" into a succession of backward and forward refinements. In Section 4 we summarise the relevant properties of extended substitutions which we subsequently exploit to develop our new single refinement

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 195–208, 2008.

method for B in Section 5; in Section 6 we illustrate the use of our new method on an example refinement scenario; in Section 7 we compare our single complete method for B with that formulated for Z in [4] before finally relating it to other relevant recent work and drawing some conclusions in Section 8.

2 Gardiner and Morgan's Rule for Data Refinement

Gardiner and Morgan [11] significantly advanced our understanding of data refinement when they showed that forward and backward refinement could be subsumed into a single complete refinement rule in which the traditional retrieve relation between abstract and concrete states is superseded by a monotonic predicate transformer of sets of abstract states to sets of concrete states. Such a predicate transformer can be regarded as characterising in terms of its wp semantics a heterogeneous monotonic computation from concrete states to abstract states called a *representation operation*. Our intuition is that in a particular refinement context such an operation "computes" for any given concrete state an abstract state which that concrete state can be said to "represent".

2.1 Cosimulation

For a pair of abstract data types *Adt* and *Cdt* with respective state spaces $Astate$ and *Cstate*, and respective initialisations ainit and cinit, finalisations afin and cfin, and repertoires of operations aop_i and cop_i for $i \in I$, then a monotonic representation operation rep from *Cstate* to $Astate$ is a *cosimulation* if the following hold:

$$\mathsf{ainit} \sqsubseteq \mathsf{cinit} ; \mathsf{rep}$$

$$\mathsf{rep} ; \mathsf{aop}_i \sqsubseteq \mathsf{cop}_i ; \mathsf{rep} \qquad \text{for each } i \in I$$

$$\mathsf{rep} ; \mathsf{afin} \sqsubseteq \mathsf{cfin}$$

The significance of the existence of such a cosimulation is that it establishes that *Cdt* refines *Adt*. In the special case where rep is disjunctive then *Cdt* is a *forward refinement* of *Adt*, while if rep is conjunctive then *Cdt* is a *backward refinement* of *Adt*. There is, however, an important qualification on the completeness of Gardiner and Morgan's single rule, namely that the abstract operations and the representation operation itself must only be at most *boundedly* nondeterministic.

In prominent formal modelling methods such as B [1], Z [16] and VDM [12] finalisations are invariably just projections onto the global space, so the finalisation condition is trivially met providing that rep is total (*i.e.* everywhere feasible).

3 Factorising an Arbitrary Refinement

For any relation $R \in X \leftrightarrow Y$ Back and von Wright [2] define two particular computations from X to Y. They call these respectively the *demonic* and *angelic*

relational updates on R, and denote them respectively by $[R]$ and $\{R\}$. The former is characterised by a conjunctive wp predicate transformer, the latter by a disjunctive one. If x and y range respectively over X and Y, R is expressed as predicate $R(x, y)$ and $Q(y)$ is any postcondition predicate, we have

$$\begin{aligned} \mathrm{wp}([R], Q) &=_{df} \forall y \,.\, R \Rightarrow Q \\ \mathrm{wp}(\{R\}, Q) &=_{df} \exists y \,.\, R \wedge Q \end{aligned}$$

In [2] it is shown that for any monotonic computation comp from X to Y an intermediate state space Z can be constructed with relations $R_1 \in X \leftrightarrow Z$ and $R_2 \in Z \leftrightarrow Y$ such that $\mathsf{comp} = \{R_1\} ; [R_2]$. This explains why an arbitrary refinement of a data type *Adt* by another *Cdt* can always be factored into a backward refinement of *Adt* by some intermediate data type *Bdt* and then a forward refinement of that by *Cdt*. In these refinements the relations R_1 and R_2 play the familiar role of retrieve relations between the concrete and abstract states.

3.1 Traditional Representation of Refinements in B

Currently in both classical and Event-B refinement the retrieve relation concerned is of course subsumed along with the concrete machine's state invariant into what is known as the "gluing" invariant. The concrete machine is therefore not explicitly exhibited in the refinement component which is actually presented, although it is always inferrable from the latter. It is important to appreciate that this is a merely the way the original architects of the B method chose to represent refinements, rather than being fundamental to the concept of refinement itself in B. Other possibilities for representing refinements in B are quite imaginable. For example, in [3] a new RETRENCHMENT construct is proposed which refers to a pair of existing machines to express the existence of a *retrenchment* relation between them. In the same way B might have had a REFINEMENT construct which refers to a pair of existing machines and provides an appropriate retrieve relation between them.

4 Extended Substitutions

In [10] B's generalised substitution language is extended by the introduction of angelic choice, and a theory of so-called *extended* substitutions is developed. In particular, the bounded angelic and demonic choice operators are denoted respectively by "$\sqcup$" and "$\sqcap$". Like ordinary generalised substitutions [1,8], extended substitutions can naturally express heterogeneous computations (those whose starting and final state spaces are distinct). This merely requires that their *passive* (read frame) variables are all associated with the starting state space, while their *active* (write frame) variables are all associated with the final state space[1]. The significance here is that extended substitutions provide a means of

[1] In the theory of generalised substitutions in [8] and of extended substitutions in [10] the active frame of a substitution is simply called its frame.

expressing a heterogeneous monotonic representation operation in B. We note that the read frame of an operation includes its input parameters, while its write frame includes its output parameters.

4.1 Relational Characterisation of an Extended Substitution

Extended substitutions have several important associated characteristic predicates. For our purpose here the most significant of these is the so-called before-after *power co-predicate*[2] $\mathrm{cod}(S)$, defined for an extended substitution S with frame s as follows:

$$\mathrm{cod}(S) \quad =_{df} \quad [S]\, s \in u$$

Here the atomic variable u is assumed fresh, and ranges over sets of final states, where each such final state is denoted by a tuple whose components correspond to the individual variables of the final state in lexical order of their names, while the frame variable s is interpreted here as a similar tuple whose whose components collectively denote a starting state of the computation characterised by S. Thus $\mathrm{cod}(S)$ is a relational predicate whose free variables are those comprising s together with the fresh variable u.

For example, if S is $x, y := 7, x + 1 \sqcap x := 8$ then since $\mathrm{frame}(S) = x, y$ the s in the definition of $\mathrm{cod}(S)$ above is interpreted here as the tuple (x, y), so we have that

$$\begin{aligned} \mathrm{cod}(S) \quad &= \quad [x, y := 7, x + 1 \sqcap x := 8]\,(x, y) \in u \\ &= \quad [x, y := 7, x + 1]\,(x, y) \in u \ \wedge \ [x := 8]\,(x, y) \in u \\ &= \quad (7, x + 1) \in u \ \wedge \ (8, y) \in u \end{aligned}$$

Thus here $\mathrm{cod}(S)$ relates each starting state (x, y) to every corresponding set u of final states which includes states $(7, x+1)$ and $(8, y)$, and *inter alia*, therefore, to the minimal set of final states $\{(7, x + 1), (8, y)\}$. Notice that the variable u in $\mathrm{cod}(S)$ is just a placeholder for sets of possible final states of the monotonic computation characterised by S, in the same way that the primed frame variables in the before-after predicate $\mathrm{prd}(T)$ of an ordinary generalised substitution T in [8] are collectively just a placeholder for individual possible final states of the conjunctive computation characterised by T.

4.2 Refinement of Extended Substitutions

An ordinary generalised substitution is characterised by its frame, its termination predicate trm and its before-after predicate prd [8], whereas in contrast an extended substitution is characterised by its frame and its cod alone without

[2] It is called a power co-predicate to distinguish it from its dual the *power predicate* $\mathrm{pod}(S) \ =_{df} \ \neg\, [S]\, s \notin u$ also defined in [10], which with the frame s provides an alternative full characterisation of S.

need of its trm. Indeed [10, Prop 5.6] establishes the following important first-order characterisation of refinement between extended substitutions S and T with the same frame, where v denotes the list of all free variables of $\mathrm{cod}(S)$ and $\mathrm{cod}(T)$ –including of course the special atomic variable u used in the definition of cod:

$$S \sqsubseteq T \quad \Leftrightarrow \quad \forall v \,.\, \mathrm{cod}(S) \Rightarrow \mathrm{cod}(T)$$

5 A Complete Single Refinement Rule for B

We exploit the above formulation of extended-substitution refinement to re-express Gardiner and Morgan's single complete refinement rule described in Section 2 by replacing its explicit occurrences of the refinement symbol $\sqsubseteq$. This yields the following complete first-order characterisation of the refinement of one B machine *Amach*, with initialisation ainit and operations aop_i for $i \in I$, by another *Cmach* with initialisation cinit and corresponding operations cop_i for $i \in I$. Such a refinement is verified if a representation operation rep can be specified from *Cmach*'s states to *Amach*'s states, expressed as a total boundedly nondeterministic extended substitution, such that

$$\forall v \;.\; \mathrm{cod}(\mathsf{ainit}) \;\Rightarrow\; \mathrm{cod}(\mathsf{cinit} \,;\, \mathsf{rep})$$

$$\forall v \;.\; \mathrm{cod}(\mathsf{rep} \,;\, \mathsf{aop}_i) \;\Rightarrow\; \mathrm{cod}(\mathsf{cop}_i \,;\, \mathsf{rep}) \qquad \text{for each } i \in I$$

where v again signifies the list of all free variables of the cods concerned here.

5.1 Nature of a First-Order Characterisation

The above pair of obligations represent a first-order characterisation of refinement since they can be re-written to eliminate first all the references to cod by applying its definition and then the resulting substitutions by applying them as wp predicate transformers. This will result in a finite collection of proof obligations expressed only in first-order logic with set-membership and equality, and therefore eminently amenable to manual or machine-assisted proof.

The fact that an extended substitution is characterised by its frame and cod alone without need of trm conveniently serves to limit the number of proof obligations so generated. This is in contrast to traditional classical B refinement [1] which generates two proof obligations for each operation, one essentially concerned with before-after effects and one concerned with termination. In the following section we will illustrate our refinement method with an example.

6 Schrődinger's Cat Revisited

The trio of machines below is almost the same as the *Schrődinger's Cat* example given in [9] as one of several examples of "intuitively obvious" co-refinements

which nevertheless can only be proved in one direction but not the other by B's traditional forward refinement method.

Our *ACat* and *BCat* machines each model from an external perspective the scenario of putting a cat into an opaque box, and then later taking it out and thereupon discovering whether it has survived or died during its confinement, its fate having been dealt nondeterministically.

6.1 The Abstract and Concrete Specifications

First we introduce our *GivenSets* machine declaring relevant types:

MACHINE *GivenSets*

SETS
$BOXSTATE = \{empty, full\}$
$CATSTATE = \{alive, dead\}$

END

In the *Acat* machine below the cat's fate is actually sealed when it is placed in the box, because it is then that the state variable *cat* is nondeterministically assigned its relevant value *alive* or *dead* which will subsequently be reported when that cat is taken out of the box:

MACHINE *Acat*

SEES *GivenSets*

VARIABLES $acat, abox$

INVARIANT $abox \in BOXSTATE \ \wedge \ acat \in CATSTATE$

INITIALISATION $abox := empty \ || \ acat :\in CATSTATE$

OPERATIONS

put $\widehat{=}$ PRE $abox = empty$
THEN $abox := full \ || \ acat :\in CATSTATE$
END ;

$rr \longleftarrow$ take $\widehat{=}$
PRE $abox = full$
THEN $abox, rr := empty, acat$
END

END

On the other hand, in the *BCat* machine below the cat's fate isn't sealed until it is taken out of the box, because only then is the report variable *rr* nondeterministically assigned its value *alive* or *dead*:

MACHINE *Bcat*

SEES *GivenSets*

VARIABLES $bbox$

INVARIANT $bbox \in BOXSTATE$

INITIALISATION $bbox := empty$

OPERATIONS

$\mathsf{put} \;\widehat{=}\;$ PRE $bbox = empty$
THEN $bbox := full$
END ;

$rr \longleftarrow \mathsf{take} \;\widehat{=}$
PRE $bbox = full$
THEN $box := empty \;||\; rr :\in CATSTATE$
END

END

Clearly an external observer must remain entirely oblivious of this fine distinction between these machines' respective internal workings concerning just when the cat's fate is actually determined. From his perspective the machines behave identically. With a complete refinement method we ought to be able to prove both that $Acat \sqsubseteq Bcat$ and $Bcat \sqsubseteq Acat$. With B's standard refinement method we can only prove that $Bcat \sqsubseteq Acat$, but not that $Acat \sqsubseteq Bcat$. In [9] we developed a counterpart *backward* refinement, but even that doesn't allow us to prove directly here that $Acat \sqsubseteq Bcat$, since this isn't purely a backward refinement either[3].

6.2 Proof of Refinement

We will now prove directly that $Acat \sqsubseteq Bcat$ using our new single complete refinement method. For this we deem $Acat$ as the abstract datatype while $Bcat$ is the concrete one.

Representation Operation. First, we specify an appropriate representation operation:

$\mathsf{rep} \;\widehat{=}\;$ IF $bbox = empty$
THEN $abox, acat := empty, alive \;\sqcup\; abox, acat := empty, dead$
ELSE $abox, acat := full, alive \;\sqcap\; abox, acat := full, dead$
END

We note that rep employs both demonic choice "$\sqcap$" and angelic choice "$\sqcup$" so it is non-trivially monotonic.

[3] The original $Acat$ in [9] is subtly different from the one here: in addition to assigning values to $abox$ and rr its version of take also nondeterministically assigns either $alive$ or $dead$ to $acat$. This has no effect on the externally observable behaviour of the machine, but turns $Acat \sqsubseteq Bcat$ into a purely backward refinement which can be proved directly by [9]'s backward refinement method.

Initialisation Labelling the abstract ($Acat$) initialisation as ainit and the concrete ($Bcat$) one as binit, we have to prove that

$$\mathrm{cod}(\mathsf{ainit}) \Rightarrow \mathrm{cod}(\mathsf{binit}\,;\mathsf{rep})$$

Proof:

$\mathrm{cod}(\mathsf{ainit})$
$= \{$ defn of cod $\}$
$[\mathsf{ainit}]\,(abox, acat) \in u$
$= \{$ body of ainit $\}$
$[abox := empty \;\; || \;\; acat :\in CATSTATE]\,(abox, acat) \in u$
$= \{$ rewrite $||$ $\}$
$[abox, acat := empty, alive \sqcap abox, acat := empty, dead]\,(abox, acat) \in u$
$= \{$ apply substitution $\}$
$(empty, alive) \in u \;\wedge\; (empty, dead) \in u$ (1)

whereas

$\mathrm{cod}(\mathsf{binit}\,;\mathsf{rep})$
$= \{$ defn of cod $\}$
$[\mathsf{binit}\,;\mathsf{rep}]\,(abox, acat) \in u$
$= \{$ defn of ; $\}$
$[\mathsf{binit}]\,[\mathsf{rep}]\,(abox, acat) \in u$
$= \{$ body of rep $\}$
$[\mathsf{binit}]\,[\text{IF ... END}]\,(abox, acat) \in u$
$= \{$ appln of IF ... END $\}$
$[\mathsf{binit}]\,((bbox = empty \Rightarrow (empty, alive) \in u \;\vee\; (empty, dead) \in u)$
$\qquad \wedge \;\; (bbox = full \Rightarrow (full, alive) \in u \;\wedge\; (full, dead) \in u))$
$= \{$ body of binit $\}$
$[bbox := empty]\,((bbox = empty \Rightarrow (empty, alive) \in u \;\vee\; (empty, dead) \in u)$
$\qquad \wedge \;\; (bbox = full \Rightarrow (full, alive) \in u \;\wedge\; (full, dead) \in u))$
$= \{$ apply substitution, logic $\}$
$(empty, alive) \in u \;\vee\; (empty, dead) \in u$ (2)

whence it can be seen that (1) $\Rightarrow$ (2)

□

The put Operation. To differentiate the abstract and concrete versions of put we label the former as aput and the latter as bput. We have to prove that

$$\mathrm{cod}(\mathsf{rep}\,;\mathsf{aput}) \Rightarrow \mathrm{cod}(\mathsf{bput}\,;\mathsf{rep})$$

Proof:

$\mathrm{cod}(\mathsf{rep}\,;\mathsf{aput})$
$= \{$ defn of cod $\}$

$$
\begin{array}{l}
[\mathsf{rep}\,;\,\mathsf{aput}]\,(abox, acat) \in u \\
\quad = \{\ \text{defn of } ;\ \} \\
[\mathsf{rep}]\,[\mathsf{aput}]\,(abox, acat) \in u \\
\quad = \{\ \text{body of } \mathsf{aput}\ \} \\
[\mathsf{rep}]\,[abox = empty \mid (abox := full \ \ || \ \ acat :\in CATSTATE)]\,(abox, acat) \in u \\
\quad = \{\ \text{defn of } \mid\ \} \\
[\mathsf{rep}]\,(abox = empty \ \wedge\ [abox := full \ \ || \ \ acat :\in CATSTATE]\,(abox, acat) \in u \\
\quad = \{\ \text{rewrite } ||\ \} \\
[\mathsf{rep}]\,(abox = empty \ \wedge \\
\qquad [abox, acat := full, alive \sqcap abox, acat := full, dead]\,(abox, acat) \in u \\
\quad = \{\ \text{apply substitution, logic}\ \} \\
[\mathsf{rep}]\,(abox = empty \ \wedge\ (full, alive) \in u \ \wedge\ (full, dead) \in u) \\
\quad = \{\ \text{body of } \mathsf{rep}\ \} \\
[\text{IF ... END}]\,(abox = empty \ \wedge\ (full, alive) \in u \ \wedge\ (full, dead) \in u) \\
\quad = \{\ \text{apply IF ... END, logic}\ \} \\
bbox = empty \ \wedge\ (full, alive) \in u \ \wedge\ (full, dead) \in u \qquad (3)
\end{array}
$$

whereas

$$
\begin{array}{l}
\text{cod}(\mathsf{bput}\,;\,\mathsf{rep}) \\
\quad = \{\ \text{defn of cod}\ \} \\
[\mathsf{bput}\,;\,\mathsf{rep}]\,(abox, acat) \in u \\
\quad = \{\ \text{defn of } ;\ \} \\
[\mathsf{bput}]\,[\mathsf{rep}]\,(abox, acat) \in u \\
\quad = \{\ \text{body of } \mathsf{rep}\ \} \\
[\mathsf{bput}]\,[\text{IF ... END}]\,(abox, acat) \in u \\
\quad = \{\ \text{apply IF ... END}\ \} \\
[\mathsf{bput}]\,(bbox = empty \Rightarrow (empty, alive) \in u \ \vee\ (empty, dead) \in u) \ \wedge \\
\qquad\qquad (bbox = full \Rightarrow (full, alive) \in u \ \wedge\ (full, dead) \in u) \\
\quad = \{\ \text{body of } \mathsf{bput}\ \} \\
[bbox = empty \mid bbox := full] \\
\qquad\qquad (bbox = empty \Rightarrow (empty, alive) \in u \ \vee\ (empty, dead) \in u) \ \wedge \\
\qquad\qquad (bbox = full \Rightarrow (full, alive) \in u \ \wedge\ (full, dead) \in u) \\
\quad = \{\ \text{apply substitution, logic}\ \} \\
bbox = empty \ \wedge\ (full, alive) \in u \ \wedge\ (full, dead) \in u \qquad (4)
\end{array}
$$

whence it can be seen that (3) = (4)

□

The take Operation. To differentiate the abstract and concrete versions of take we label the former as atake and the latter as btake. Since they share the output variable *rr* this appears in both their frames. We have to prove that

$$\text{cod}(\mathsf{rep}\,;\,\mathsf{atake}) \Rightarrow \text{cod}(\mathsf{btake}\,;\,\mathsf{rep})$$

We note that the relevant frame tuple u here is $(abox, acat, rr)$.
Proof:

$$
\begin{array}{l}
\mathrm{cod}(\mathsf{rep}\,;\mathsf{atake}) \\
\quad = \{\ \text{defn of cod}\ \} \\
[\mathsf{rep}\,;\mathsf{atake}]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{defn of}\ ;\ \} \\
[\mathsf{rep}]\,[\mathsf{atake}]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{body of atake}\ \} \\
[\mathsf{rep}]\,[abox = full \mid abox, rr := empty, acat]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{apply substitution}\ \} \\
[\mathsf{rep}]\,(abox = full \ \wedge\ (empty, acat, acat) \in u) \\
\quad = \{\ \text{body of rep}\ \} \\
[\text{IF ... END}]\,(abox = full \ \wedge\ (empty, acat, acat) \in u) \\
\quad = \{\ \text{apply IF ... END, logic}\ \} \\
bbox = full \ \wedge\ (empty, alive, alive) \in u \ \wedge\ (empty, dead, dead) \in u
\end{array}
\tag{5}
$$

whereas

$$
\begin{array}{l}
\mathrm{cod}(\mathsf{btake}\,;\mathsf{rep}) \\
\quad = \{\ \text{defn of cod}\ \} \\
[\mathsf{btake}\,;\mathsf{rep}]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{defn of}\ ;\ \} \\
[\mathsf{btake}]\,[\mathsf{rep}]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{body of rep}\ \} \\
[\mathsf{btake}]\,[\text{IF ... END}]\,(abox, acat, rr) \in u \\
\quad = \{\ \text{apply IF ... END}\ \} \\
[\mathsf{btake}]\,((bbox = empty \Rightarrow (empty, alive, rr) \in u \ \vee\ (empty, dead, rr) \in u) \ \wedge \\
\qquad\qquad (bbox = full \Rightarrow (full, alive, rr) \in u \ \wedge\ (full, dead, rr) \in u)) \\
\quad = \{\ \text{body of btake}\ \} \\
[bbox = full \mid bbox := empty \ \ ||\ \ rr :\in CATSTATE] \\
\qquad ((bbox = empty \Rightarrow (empty, alive, rr) \in u \ \vee\ (empty, dead, rr) \in u) \ \wedge \\
\qquad\qquad (bbox = full \Rightarrow (full, alive, rr) \in u \ \wedge\ (full, dead, rr) \in u)) \\
\quad = \{\ \text{rewrite}\ \ ||\ \ \} \\
[bbox = full \mid bbox, rr := empty, alive \sqcap bbox, rr := empty, dead] \\
\qquad ((bbox = empty \Rightarrow (empty, alive, rr) \in u \ \vee\ (empty, dead, rr) \in u) \ \wedge \\
\qquad\qquad (bbox = full \Rightarrow (full, alive, rr) \in u \ \wedge\ (full, dead, rr) \in u)) \\
\quad = \{\ \text{apply substitution, logic}\ \} \\
bbox = full \ \wedge\ ((empty, alive, alive) \in u \ \vee\ (empty, dead, alive) \in u) \ \wedge \\
\qquad ((empty, alive, dead) \in u \ \vee\ (empty, dead, dead) \in u)
\end{array}
\tag{6}
$$

whence it can be seen that (5) $\Rightarrow$ (6)

□

7 Comparison with Single Complete Refinement in Z

In [4] Derrick gives a single complete refinement rule for Z, which he expresses within an appropriate relational framework although it is inspired by the older technique of *possibility mappings* first proposed in [15]. In place of a simple retrieve relation between abstract and concrete states, his rule employs a *powersimulation*, *i.e.* a relation from sets of abstract states to individual concrete states. There is in fact a close correspondence between Derrick's method and ours: specifically, his powersimulation when inverted should yield the power copredicate of our cosimulation as embodied by our representation operation.

7.1 Derrick's Example Translated into B

The single complete rule in [4] is illustrated there on an example refinement which is neither a forward nor backward one, and therefore unamenable to a direct single-step proof using alone either the forward refinement rules or backward refinement rules in [16], although of course since these rules are jointly complete it would be possible to prove this as indeed any valid refinement by using them in combination via an intermediate refinement.

We applied our method to the same example, after first translating this from Z to B to obtain the following pair of machines:

MACHINE $Amach$
VARIABLES xx
INVARIANT $xx \in 0..5$
INITIALISATION $xx := 0$
OPERATIONS
  one $\widehat{=}$ PRE $xx = 0 \lor xx = 1$ THEN $xx = 0 \Longrightarrow xx := 1 \sqcap xx = 1 \Longrightarrow xx := 0$ END ;
  two $\widehat{=}$ PRE $xx = 0$ THEN $xx := 2 \sqcap xx := 3$ END ;
  three $\widehat{=}$ PRE $xx = 2 \lor xx = 3$ THEN $xx = 2 \Longrightarrow xx := 4 \sqcap xx = 3 \Longrightarrow xx := 5$ END
END

MACHINE $Cmach$
VARIABLES yy
INVARIANT $yy \in \{0, 2, 4, 5\}$
INITIALISATION $yy := 0$
OPERATIONS
  one $\widehat{=}$ PRE $cc = 0$ THEN $yy := 0$ END ;
  two $\widehat{=}$ PRE $yy = 0$ THEN $yy := 2$ END ;
  three $\widehat{=}$ PRE $yy = 2$ THEN $yy := 4 \sqcap yy := 5$ END
END

7.2 Verification of Derrick's Refinement Example

Our experience of verifying Derrick's refinement example was interesting. First, we constructed the following representation operation rpn corresponding directly to the powersimulation given by Derrick in [4] for the same example:

$$\begin{aligned} \mathsf{rpn} \;\widehat{=}\; & (yy = 0 \mid (xx := 0 \sqcap xx := 1) \sqcup xx := 0 \sqcup xx := 1) \\ & \sqcup\ (yy = 4 \vee yy = 5 \mid (xx := 4 \sqcap xx := 5)) \end{aligned}$$

We were then unexpectedly perplexed to find that this rpn was ineffective for proving $Amach \sqsubseteq Cmach$ by our method. On the other hand, we found *were* able to verify this refinement by means of a different representation operation rpr, where

$$\begin{aligned} \mathsf{rpr} \;\widehat{=}\; & (yy = 0 \mid (xx := 0 \sqcup xx := 1)) \\ & \sqcup\ (yy = 2 \mid (xx := 2 \sqcap xx := 3) \\ & \sqcup\ (yy = 4 \vee yy = 5 \mid (xx := 4 \sqcap xx := 5)) \end{aligned}$$

We omit here the proofs involved, which are similar to those already given for Schrődinger's cat. We note that our representation operation rpr corresponds to the powersimulation r, defined in Z terms by

$$\begin{array}{|l} r : \mathbb{P}\, Astate \leftrightarrow Cstate \\ \hline r = \{\{\langle xx \rightsquigarrow 0\rangle\} \mapsto \langle yy \rightsquigarrow 0\rangle, \\ \quad \{\langle xx \rightsquigarrow 1\rangle\} \mapsto \langle yy \rightsquigarrow 0\rangle, \\ \quad \{\langle xx \rightsquigarrow 2\rangle, \langle xx \rightsquigarrow 3\rangle\} \mapsto \langle yy \rightsquigarrow 2\rangle, \\ \quad \{\langle xx \rightsquigarrow 4\rangle, \langle xx \rightsquigarrow 5\rangle\} \mapsto \langle yy \rightsquigarrow 4\rangle, \\ \quad \{\langle xx \rightsquigarrow 4\rangle, \langle xx \rightsquigarrow 5\rangle\} \mapsto \langle yy \rightsquigarrow 5\rangle\} \end{array}$$

rather than the r defined in [4]. We subsequently alerted [4]'s author to this discrepancy between his and our powersimulations. He obliged us by undertaking his own investigation which resulted in his diagnosing a printer's error in [4]; moreover, he confirmed that the correct powersimulation for the example is indeed our r above rather than that given in [4]. We take this as a significant vindication of our single refinement method for B: not only has it proved effective in independently verifying this refinement example; it also directly led us to detect a previously unsuspected mistake in the original powersimulation given in [4] for verifying the same example by Derrick's rule.

8 Related Work and Conclusions

In [5] model-checking is employed to generate retrieve relations for both forward and backward refinements. Presumably this technique could be extended to generate powersimulations for arbitrary refinements, although this is not discussed in [5]. On the other hand [14] does describe automatic verification of arbitrary refinements in B using the ProB model checker [13]. That technique uses ProB

to construct a relation from concrete states to sets of abstract states which is in effect the power co-predicate of a cosimulation for the refinement, so this complements our refinement proof method rather well.

Our single refinement method is applicable to classical B and Event-B alike. In particular, Event-B's characteristic introduction of new events during refinement raises no particular issues for the new method. The key to our method is the construction of an effective monotonic representation operation. Our experience indicates that the flexibility afforded by the extended substitution language's syntax to arbitrarily interleave demonic and angelic choices greatly assists the developer in such an exercise.

The example refinements on which we have demonstrated our single refinement method are necessarily rather trivial, although they do nevertheless illustrate all the principles of the method so we hope that they may have served sufficiently to demonstrate that our method is amenable to the sort of mechanisation provided by both the classical B and Event-B development support environments. Indeed we hope to explore the possible provision of a suitable plug-in for the Rodin platform for the generation of the proof obligations of our method. Two extensions to core B are needed by our method, namely support for extended substitutions and also for arbitrary tuples. Fortunately, we believe neither of these should pose any particular difficulty for support tool implementors.

Acknowledgements

We are grateful for the points raised by the anonymous reviewers, which we have endeavoured to address in this final version of the paper.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, New York (1998)
3. Banach, R., Fraser, S.: Retrenchment and the B-Toolkit. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 203–221. Springer, Heidelberg (2005)
4. Derrick, J.: A single complete refinement rule for Z. Journal of Logic and Computation 10(5), 663–675 (2000)
5. Derrick, J., Smith, G.: Using model checking to automatically find retrieve relations. In: International Refinement Workshop (Refine 2007). Electronic Notes in Theoretical Computer Science, vol. 201, pp. 155–175. Elsevier, Amsterdam (2008)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall International, Englewood Cliffs (1976)
7. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Berlin (1990)
8. Dunne, S.E.: A theory of generalised substitutions. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 270–290. Springer, Heidelberg (2002)

9. Dunne, S.E.: Introducing backward refinement into B. In: Bert, D., Bowen, J.P., King, S., Walden, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 178–196. Springer, Heidelberg (2003)
10. Dunne, S.E.: Chorus Angelorum. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 19–33. Springer, Heidelberg (2006)
11. Gardiner, P.H.B., Morgan, C.: A single complete rule for data refinement. Formal Aspects of Computing 5, 367–382 (1993)
12. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall, Englewood Cliffs (1990)
13. Leuschel, M., Butler, M.J.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
14. Leuschel, M., Butler, M.J.: Automatic refinement checking for B. In: Lau, K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 345–359. Springer, Heidelberg (2005)
15. Lynch, N.A.: Multivalued possibility mappings. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 519–543. Springer, Heidelberg (1990)
16. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall, Englewood Cliffs (1996)

The Composition of Event-B Models

Michael Poppleton

School of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK
mrp@ecs.soton.ac.uk

Abstract. The transition from classical B [2] to the Event-B language and method [3] has seen the removal of some forms of model structuring and composition, with the intention of reinventing them in future. This work contributes to that reinvention. Inspired by a proposed method for state-based decomposition and refinement [5] of an Event-B model, we propose a familiar parallel event composition (over disjoint state variable lists), and the less familiar event fusion (over intersecting state variable lists). A brief motivation is provided for these and other forms of composition of models, in terms of feature-based modelling. We show that model consistency is preserved under such compositions. More significantly we show that model composition preserves refinement.

1 Introduction

1.1 Historical Context

Early work on the composition of specifications and programs such as [14,1] indicated the importance of composition as a key mechanism for the scalability of Formal Methods in software development. Various compositional mechanisms were developed for classical B as defined in [2] and elaborated in [22]. These mechanisms - denoted INCLUDES, EXTENDS, USES, etc. - are syntactic in nature, and concerned with the visibility or inclusion of the text of one machine by another. A variety of visibility and usage rules and constraints are defined. These mechanisms were designed with the scalability of automated proof obligation (PO) generation and proof at least as much in mind as modelling utility. Perhaps unsurprisingly, they are not very intuitive, are dissimilar to inclusion mechanisms in other languages, and not straightforward to use. Later work [23] revealed further unsuspected modelling limitations in the composition of B machines.

Recently completed EU Framework VI project RODIN[1] saw the definition of the Event-B language [19] and the creation of the rich RODIN toolkit [3] for formal modelling, animation, verification, and proof with Event-B. Project RODIN is succeeded by project DEPLOY[2] which will, driven by industrial deployments, further develop the RODIN toolset and Event-B methods.

[1] RODIN - Rigorous Open Development Environment for Open Systems: EU IST Project IST-511599, http://rodin.cs.ncl.ac.uk

[2] DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity: FP VII Project 214158 under Strategic Objective IST-2007.1.2.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 209–222, 2008.

The classical B compositional mechanisms have been excised from Event-B to make way for their reinvention in future. The high aspirations of [4], which demonstrated the modelling power that could be unleashed by implementing the rich generic axiomatic structuring of set theory at the metalanguage (as opposed to object language) level, will be realized to some degree by the schedule for project DEPLOY.

The motivation for model decomposition is to reduce model size and proof complexity; there is the bonus of enabling the distribution of development work. Two methods for decompositional working through refinement are on the DEPLOY schedule. Methodologically, they work similarly: a single, "abstract" model M is developed and decomposed - or abstracted - into component models $\{\mathrm{N}_i\}$. The components are refined to more "concrete" versions $\{\mathrm{NR}_i\}$; these concrete refinements are then recomposed into model MR in a *particular way* that guarantees that MR refines M.

[19,5] propose the state-based decomposition (called "type A" decomposition, after Abrial) of a model: here the state variables $\{v_j\}$ of M are initially partitioned across the $\{\mathrm{N}_i\}$. The events $\{e_k\}$ follow variables they act on into the $\{\mathrm{N}_i\}$. Provided all events acting on a variable *v* are located in its component machine N_i, that variable is *local*, or *internal* to that machine and needs no special treatment. In general at least one variable *w* is *shared* between two given component machines that act on it; such a variable is also called *external* to each. If this is not the case, then we simply have disjoint and unrelated developments.

Of course, the refinement of *M* by *MR* only decomposes provided the gluing invariant decomposes conjunctively in the right way. More significantly, [19][3] shows that external variables must be refined by a common, functional gluing invariant; internal variables are not so constrained. The functional constraint is required by the proof of the construction. The part of the gluing invariant concerning say, external *v* refined by *w*, can be written $v = h(w)$, and this equality enables certain existential quantifications to be simplified with the existential one-point rule.

The second proposal is for event-based decomposition (called "type B" decomposition, after Butler) from [11,15]. Since "Event-B machines have the same semantic structure and refinement definitions as action systems" [Op.cit.], this is precisely the reverse of the composition proposal of [10], where it was posed in an action systems [7] setting. Here, an abstract model M is refined in a manner that facilitates the partition of events between component models. The refinement of M to a single model MR decomposes the state variables (by adding new ones), such that MR is expressible as a parallel composition of component models $|| \{\mathrm{NR}_i\}$ over the partitioned variables. Each event accessing variables in more than one NR_i is decomposed into a set of events each accessing only a local variable, that communicate by message-passing. The semantic correspondence of action systems and CSP is used to proved monotonicity of this process w.r.t. refinement.

Both the above proposals elaborate the traditional "top-down" development process; it remains canonical to start from the most general, and concise abstraction, and then to elaborate through refinement. Such top-down approaches are not natuarally receptive to reuse, where one might want to draw on a database of models and model elements, at

[3] Note that [5] make the stronger requirement that external variables are not data refined, purely to simplify their exposition.

various levels of abstraction and genericity. This work is motivated by the desire to facilitate such working with reuse, i.e. to produce a refinement-preserving compositional method, which reuses existing models. We demonstrate that Event-B models can indeed be so composed, in a manner analogous to the inverse of type A decomposition. Unlike A- and B-decomposition however, *new* events are constructed in the composite machine by a version of the event *fusion* of Butler and Back [15,8].

This introduction continues with a précis of specification (section 1.2) and refinement (section 1.3) in Event-B, and ends with some remarks (section 1.4) motivating feature-based composition as a form of reuse. Section 2 then defines our form of model composition, including the mechanism of event fusion. We show that model consistency is preserved under such composition. Section 3 proves that fusion preserves refinement, an essential property for scalable working. In conclusion section 4 considers related work, and describes future work.

1.2 Event-B Basics

This section is a précis of parts of [19], the Event-B language definition.

Event-B is designed for long-running *reactive* hardware/software systems that respond to stimuli from user and/or environment. The set-theoretic language in first-order logic (FOL) takes as semantic model a transition system labelled with event names. The correctness of a model is defined by an invariant property, i.e. a predicate, or constraint, which every state in the system must satisfy. More practically, every event in the system must be shown to preserve this invariant; this verification requirement is expressed in a number of *proof obligations* (POs). In practice this verification is performed either by model checking or theorem proving (or both).

For modelling in Event-B the two units of structuring are the *machine* of dynamic variables, events and their invariants, and the *context* of static data of sets, constants and their axioms. Every machine *sees* at least one context. The unit of behaviour is the *event*. An event e acting on (a list of) state variables v, subject to enabling condition, or *guard* $G(v)$ and *action*, or *assignment* $E(v)$, has syntax

$$e \mathrel{\widehat{=}} \textsf{ when } G(v) \textsf{ then } E(v) \textsf{ end} \tag{1}$$

That is, when the state is such that the guard is true, this enables the action, or state transition defined by $E(v)$. Next we give a more general syntax for a nondeterministic event. We give the guard, whose meaning is obvious from the before-after predicate for the event: the guard is precisely the statement that there exists an after-state defined by the before-after predicate, i.e. that the latter is *feasible*.

event syntax:	$\textsf{any } t \textsf{ where } Q(t,v) \textsf{ then } v := F(t,v) \textsf{ end}$	(2)
guard:	$\exists t \bullet Q(t,v)$	(3)
before-after predicate:	$\exists t \bullet (Q(t,v) \wedge v' = F(t,v))$	(4)

Note the shorthand syntax: since v above is in general a variable list, $F(t,v)$ is an expression list. (2-4) define a t-indexed nondeterministic choice between those transitions

$v' = F(t, v)$ for which $Q(t, v)$ is true[4]. t is interpreted as an input from the environment. Syntactic sugar is available: parallel ($||$) is used to enumerate multiple single-variable assignments. In the any form, the event guard is not stated explicitly since it is constructed automatically from the where clause $Q(t, v)$. The following useful property always holds for the guard G_e and before-after predicate E_e of an any -defined event e:

$$E_e \Rightarrow G_e \tag{5}$$

For the sake of completeness it is worth defining a more general event syntax that specifies an after-state in terms of a predicate it satisfies, called $x :| P(x, x', y)$. The equality-based event definition of (2-4) is usually sufficiently expressive and forms the basis of this work.

event syntax:	any t where $P(x, t, y)$ then $x := t$ end	(6)
guard:	$\exists x' \bullet P(x, x', y)$	(7)
before-after predicate:	$P(x, x', y)$	(8)

An event e works in a model (comprising a machine and at least one context) with constants c and sets s subject to *axioms* (properties) $P(s, c)$ and an *invariant* $I(s, c, v)$. Thus the event guard G and assignment with before-after predicate E take s, c as parameters. Two of the consistency proof obligations [5] (POs) for event e are FIS (feasibility preservation) and INV (invariant preservation). For an event defined as (2-4), FIS clearly discharges trivially.

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \Rightarrow \exists v' \bullet E(s, c, v, v') \quad \text{FIS} \tag{9}$$

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge E(s, c, v, v') \Rightarrow I(s, c, v') \quad \text{INV} \tag{10}$$

1.3 Refinement

The refinement of a context is simply its elaboration, by the addition of new sets, constants and axioms. The refinement of a machine includes both data and algorithm refinement: all variables v are replaced by new ones w, some simply by renaming - i.e. of the same type and meaning - and others by variables of different type. Existing events are transformed to work on the new variables, and new events can be defined; that is, the behaviour of an abstract event e can be refined by some sequence of e and new events. The new behaviour will usually reduce nondeterminism. When model $N(w)$ refines $M(v)$, it also has an invariant $J(s, c, v, w)$ which can include M's variables v. This "gluing invariant", or refinement relation, has the function of relating abstract variables v to concrete ones w mathematically.

In Fig. 1, M sees C, N refines M and D refines C, then N sees D. It is also possible for C not to be refined (i.e. to be identity-refined), in which case N sees C.

[4] The deterministic assignment is simply written $v := F(v)$, without an any variable or where clause.

[5] See [19] for the others.

As for simple machines, there are proof obligations for refinement. We assume axioms $P(s,c)$, and abstract, concrete invariants $I(s,c,v)$ and $J(s,c,v,w)$ respectively. An abstract event with guard $G_A(s,c,v)$ and before-after predicate $E_A(s,c,v,v')$ is refined by a concrete event with guard $G_C(s,c,w)$ and before-after predicate $E_C(s,c,w,w')$. The following obligations state that the concrete event is feasible (FIS_REF), the concrete guard strengthens the abstract one (GRD_REF), and that every concrete step is correct (simulates) w.r.t. some abstract step (INV_REF):

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge G_C(s,c,w) \Rightarrow \exists w' \bullet E_C(s,c,w,w') \quad \text{FIS\_REF} \quad (11)$$

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge G_C(s,c,w) \Rightarrow G_A(s,c,v) \quad \text{GRD\_REF} \quad (12)$$

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge G_C(s,c,w) \wedge E_C(s,c,w,w') \Rightarrow \exists v' \bullet (E_A(s,c,v,v') \wedge J(s,c,v',w')) \quad \text{INV\_REF} \quad (13)$$

[19] defines further refinement obligations, for nondivergence of new events introduced, and "relative deadlockfreeness" to ensure a concrete model cannot deadlock more often than the abstract one. We do not pursue these matters in this work.

1.4 Model Reuse with Features

A useful way to analyse reuse - provided by the Software Product Line (SPL) community, e.g. [20] - is in terms of data and behavioural variability [12] between system versions. The concepts are as applicable in software reuse through evolution as they are in SPLs. Event-B deals with static data variability by separating - in a B model - the dynamic *machine* from the static *context*. However, there is no mechanism to deal with behavioural variability. It is straightforward to generate variant versions of a B development that differ only in configuration, or static data: simply switch the

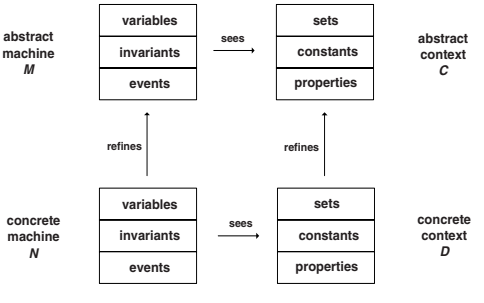


Fig. 1. Machine and context refinements (from [19])

required contexts into the refinement tree. In [21] we proposed the notion of a feature model, as a fine level of granularity for B specification, with composition of such as a mechanism for behavioural variability in development, thus contributing to the "Roadmap for Enhanced Languages and Methods to Aid Verification" [18]. A *feature* is defined simply to be a B model which is (largely) atomic with respect to composition. "Atomic", in the sense that no syntactically partial, or incomplete, model will (at this time) be input to reuse in modelling. "Largely", in the sense that certain obvious refactorings, such as systematic renamings of certain identifiers, or text insertions such as strengthening of predicates, will be allowed.

By way of brief motivation for feature-based composition, imagine a database of Event-B features for some application domain, such as resource management for distributed computing. Imagine a model M_1 with variables x, y and an event

$$e = \text{ any } t \text{ where } Q_1(t,x) \text{ then } x := F_1(t,x,y) \text{ end}$$

where the event specifies the allocation of some resource x such as a virtual circuit, subject to some QoS requirement given by (Q_1, F_1). We might wish - subject to suitable systematic variable and event renaming - to compose M_1 with some other model M_2 to allow specification of other resources through other events f, g in M_2. If both variable lists and event lists are disjoint, composition is a trivial matter with no extra proof obligations arising. Should the variable lists overlap, it will be necessary to show that M_1 events preserve the invariant of M_2 and vice versa.

A more interesting case is where we wish to *fuse* an event e from M_1 with some event f from M_3, say. It may be that M_3 specifies different requirements of virtual circuits, such as fault-tolerance/redundancy. We may wish to select in x a virtual circuit satisfying QoS (Q_1, F_1), *and* supporting fault-tolerance as specified by f.

In the next section we demonstrate that Event-B models (or features) can indeed be composed, in a manner analogous to the inverse of state-based decomposition, in a way that preserves refinement. We will focus in particular on the case of event fusion.

2 Model and Event Fusion

Consider two models M_1 and M_2 which we propose to *fuse* by combining variables and events. That is, we concatenate the variable lists and events, conjoin those events with common names (in a manner to be defined) in a new model M. The variable list v in M_1 comprises the list x of actioned variables and the list y of skipping variables for each event[6]. Similarly variables w in M_2 comprise actioned z and skipping a. We define $xz = x \cap z$, the common actioned variables, and $ya = y \cap a$, the common skipping variables. Note that the other intersecting variable lists yz and xa are both empty, to enable meaningful composition definitions. Since the context axioms P_1, P_2 of the two models do not influence the proofs we assume they share sets and constants s, c without loss of generality.

[6] Strictly speaking v should be partitioned into (x_e, y_e) for each event e. We do not need this decoration since only one event in each model is considered.

$M_1 : v = x \cup y$
$s, c, P_1(s,c)$ context
$v, I_1(s,c,v)$ invariant
event:
$e =$ any α where $Q_1(\alpha, v)$
then $x := F_1(\alpha, v)$
Thus
$G_e \mathrel{\widehat{=}} \exists \alpha \bullet Q_1(\alpha, v)$
$E_e \mathrel{\widehat{=}} \exists \alpha \bullet (Q_1(\alpha, v) \wedge x' = F_1(\alpha, v)) \wedge y' = y$

$M_2 : w = z \cup a$
$s, c, P_2(s,c)$ context
$w, I_2(s,c,w)$ invariant
event:
$f =$ any β where $Q_2(\beta, w)$
then $z := F_2(\beta, w)$
Thus
$G_f \mathrel{\widehat{=}} \exists \beta \bullet Q_2(\beta, w)$
$E_f \mathrel{\widehat{=}} \exists \beta \bullet (Q_2(\beta, w) \wedge z' = F_2(\beta, w)) \wedge a' = a$

Next we define the fused model M, distinguishing clearly in the before-after predicate between actioned variables $< x - xz >$ exclusive to M_1, common actioned variables xz, and actioned variables $< z - xz >$ exclusive to M_2. We write the fusion of events e and f as $e \odot f$. The fused model is then specified in the obvious way:

$M : v, w = x \cup z \cup y \cup a$
$s, c, P_1(s,c) \wedge P_2(s,c)$ context
$v, w, I_1(s,c,v) \wedge I_2(s,c,w)$ invariant
$e \odot f =$ any α, β where $Q_1(\alpha, v) \wedge Q_2(\beta, w)$
then $x := F_1(\alpha, v) \parallel z := F_2(\beta, w)$
end

The usual existence proof obligation for a machine context - i.e. $P_1 \wedge P_2$ - arises here.

The meaning of the above syntax - i.e. the use of $\parallel$ over intersecting variable lists, undefined as yet in the Event-B language - is given by the fused guard and before-after predicate definitions[7]:

$$G_{e\odot f} \mathrel{\widehat{=}} \exists \alpha, \beta \bullet (Q_1(\alpha, v) \wedge Q_2(\beta, w) \wedge F_1(\alpha, v) = F_2(\beta, w)) \tag{14}$$

$$E_{e\odot f} \mathrel{\widehat{=}} \exists \alpha, \beta \bullet (Q_1(\alpha, v) \wedge Q_2(\beta, w) \wedge < x - xz >' = F_1(\alpha, v) \wedge xz' = F_1(\alpha, v) \wedge xz' = F_2(\beta, w) \wedge < z - xz >' = F_2(\beta, w)) \wedge y' = y \wedge a' = a \tag{15}$$

Clearly, there must be sufficient nondeterminism in these definitions to satisfy $G_{e\odot f}$ for meaningful state values v, w.

The following useful properties are obvious:

$$G_{e\odot f} \Rightarrow G_e \wedge G_f E_{e\odot f} \Rightarrow E_e \wedge E_f \tag{16}$$

Theorem 1. Event consistency (9-10) is preserved under model fusion.

Proof. Assume $P_1 \wedge P_2 \wedge I_1 \wedge I_2 \wedge G_{e\odot f} \wedge E_{e\odot f}$. From (16) the hypotheses of INV(e) and INV(f) are made available, and it follows that $I_1(s,c,v') \wedge I_2(s,c,w')$. **QED**

[7] $G_{e\odot f}, E_{e\odot f}$ definitions are given in shorthand; F_1, F_2 are expression *lists*, each list being partitioned according to the variable sublists in use at that point in the definitions. Thus (14-15) should be read in terms of the appropriate sublists. In particular, (14) refers only to the the sublists of F_1, F_2 assigning to common actioned variables xz.

We make some observations:

1. The fusion of two models clearly requires sufficient nondeterminism in the fusing events' actions over shared variables, in order for the fused event to be feasible (and the fused guard not vacuously false). A natural way in which this might arise is as follows. Event $e(v_1, v_2, v_3)$, say, assigns v_1 nondeterministically to $F_1(\alpha, v_1, v_2, v_3)$ for some α, v_2 to anything in its type V_2, and skips on v_3. Event $f(v_1, v_2, v_3)$ assigns v_1 to anything in its type V_1, v_2 nondeterministically to $F_2(\beta, v_1, v_2, v_3)$ for some β, and skips on v_3. This represents the compositional modelling, from prior component models, of the requirement to perform F_1 on v_1 and F_2 on v_2, in the manner suggested in section 1.4.

   Methodologically it is desirable that the fusion of two events should refine each of them, and this is indeed the case, as we show below.
2. **Theorem 1a.** Theorem 1 applies for two models with disjoint variable lists and composing events by the same reasoning. This is a parallel composition of models, where each composed event represents the product of all transitions on all variables from the component models.
3. **Theorem 1b.** Theorem 1 applies for two models with disjoint variable and event lists; this is the embedding of each model in a larger one, where each event acts on variables from its own model and skips on those from the other model. The POs discharge trivially since each event is the identity refinement of its abstract counterpart: for event e acting on v in composed model M, INV discharges by noting that $I_1(s, c, v')$ follows from INV(e) in M_1, and that $I_2(s, c, w)$ follows from skip in M_2.

Theorem 2. The fusion $e \odot f$, in model M, of two events e and f refines each of those events in their respective models.

Proof. We discharge the refinement obligations (11-13) for $e \sqsubseteq e \odot f$; the f case is treated identically. FIS_REF (11) follows trivially in the same way that FIS does for events of the form (2-4), since $G_{e\odot f} \Rightarrow \exists\, v', w' \bullet E_{e\odot f}$. GRD_REF (12) follows trivially since $G_{e\odot f} \Rightarrow G_e$. For INV_REF, for clarity we rename abstract variables in M_v v_0, and assume

$$P_1 \wedge P_2 \wedge I_1(v_0) \wedge v_0 = v \wedge I_1(v) \wedge I_2(w) \wedge G_{e\odot f}(v, w) \wedge E_{e\odot f}(v, w, v', w')$$

We must prove

$$\exists\, v'_0 \bullet (E_e(v_0, v'_0) \wedge v' = v'_0 \wedge I_1(v'_0) \wedge I_2(w'))$$

that is, removing the identical-copy abstract variables v_0, v'_0

$$E_e(v, v') \wedge I_1(v') \wedge I_2(w'))$$

Since $E_{e\odot f} \Rightarrow E_e$, and we have the second two conjuncts from INV(e) and INV(f) resp. we are done. **QED**

3 Preservation of Refinement by Event Fusion

We show that the fusion of refined events refines the fusion of the original events. Consider the compositional arrangement of models in Fig. 2. Since this construction is

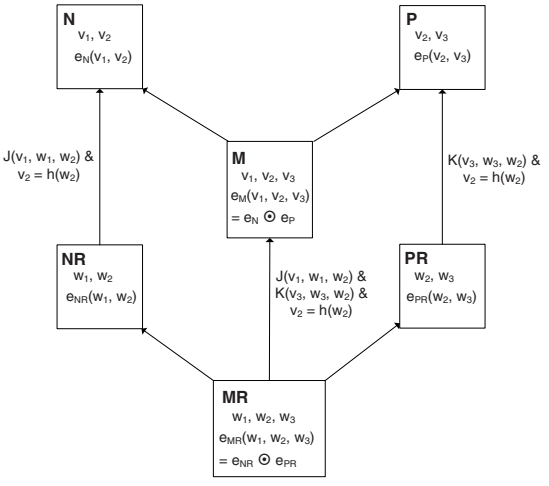


Fig. 2. Refinement of event fusion

inspired by the state-based decomposition construction of [19] (as discussed in section 1.1), the diligent reader will see that the gluing invariants here are precisely those of [Op.cit.].

Model N has variables v_1, v_2 and event $e_N(v_1, v_2)$ with guard G_N and before-after predicate E_N. Model P has variables v_2, v_3 and event $e_P(v_2, v_3)$ with guard G_P and before-after predicate E_P. v_2 is thus the shared variable between N and P. Model M over variables v_1, v_2, v_3 with event $e_M(v_1, v_2, v_3) \mathrel{\widehat{=}} e_N \odot e_P$ is the fusion[8] of N and P. The guard and before-after predicate of e_M are named G_M, E_M respectively.

Next we have two models NR, PR which refine N, P respectively. NR has variables w_1, w_2 and event $e_{NR}(w_1, w_2)$ with guard G_{NR} and before-after predicate E_{NR}. e_{NR} refines e_N with gluing invariant

$$J(v_1, w_1, w_2) \wedge v_2 = h(w2) \tag{17}$$

Similarly, PR has variables w_2, w_3 and event $e_{PR}(w_2, w_3)$ with guard G_{PR} and before-after predicate E_{PR}. e_{PR} refines e_P with gluing invariant

$$K(v_3, w_3, w_2) \wedge v_2 = h(w2) \tag{18}$$

Note the requirement that the shared variable is refined in the same functional manner in both machines; this satisfies the intuition that a shared variable should be treated "in the

[8] Contrast this construction with that of [19] as outlined in sec. 1.1: in that case events e_N and e_P, both acting on external variable v_2, *both* appear independently in M. Then, in N the external effect of e_P on v_2 must be modelled by a new *external* event e_{Px}; in N e_{Px} abstracts e_P in M. The new event is required in order that M refines N. In our scheme the fusion of events removes their independence of behaviour, thus removing the need for external events. The proof of the construction is simplified, but still requires the functional gluing invariant, for the same reason as [Op.cit.].

same way" in each sharing refinement chain, before the refinements are fused. The local variables in component machines may be defined more generally and independently of each other, while allowing the reference to the concrete shared variable.

Finally, model MR over variables w_1, w_2, w_3 has event $e_{MR}(w_1, w_2, w_3) \mathrel{\widehat{=}} e_{NR} \odot e_{PR}$ which is the fusion of e_{NR} and e_{PR}. We say that e_{MR} has guard G_{MR} and before-after predicate E_{MR}. We must now show that MR refines M w.r.t. gluing invariant

$$J(v_1, w_1, w_2) \wedge v_2 = h(w2) \wedge K(v_3, w_3, w_2) \tag{19}$$

Theorem 3. Given that $\underline{e_N \sqsubseteq e_{NR}}$:

$$\begin{aligned} &J(v_1, w_1, w_2) \wedge v_2 = h(w_2) \wedge G_{NR}(w_1, w_2) \wedge E_{NR}(w_1, w_2, w_1', w_2') \\ &\Rightarrow G_N(v_1, v_2) \wedge \exists\, v_1', v_2' \bullet (E_N(v_1, v_2, v_1', v_2') \wedge J(v_1', w_1', w_2') \wedge v_2' = h(w_2')) \end{aligned} \tag{20}$$

and $\underline{e_P \sqsubseteq e_{PR}}$:

$$\begin{aligned} &K(v_3, w_3, w_2) \wedge v_2 = h(w_2) \wedge G_{PR}(w_2, w_3) \wedge E_{PR}(w_2, w_3, w_2', w_3') \\ &\Rightarrow G_P(v_2, v_3) \wedge \exists\, v_2', v_3' \bullet (E_P(v_2, v_3, v_2', v_3') \wedge K(v_3', w_3', w_2') \wedge v_2' = h(w_2')) \end{aligned} \tag{21}$$

we must show[9] $\underline{e_M = e_N \odot e_P \sqsubseteq e_{MR} = e_{NR} \odot e_{PR}}$:

$$\begin{aligned} &J(v_1, w_1, w_2) \wedge K(v_3, w_3, w_2) \wedge v_2 = h(w_2) \wedge \\ &\quad G_{MR}(w_1, w_2, w_3) \wedge E_{MR}(w_1, w_2, w_3, w_1', w_2', w_3') \\ &\Rightarrow G_M(v_1, v_2, v_3) \wedge \exists\, v_1', v_2', v_3' \bullet (E_M(v_1, v_2, v_3, v_1', v_2', v_3') \wedge \\ &\quad J(v_1', w_1', w_2') \wedge K(v_3', w_3', w_2') \wedge v_2' = h(w_2')) \end{aligned} \tag{22}$$

Proof. is straightforward and uses the fusion definitions (14, 15), i.e.

$$G_N \mathrel{\widehat{=}} \exists\, \alpha \bullet Q_N(\alpha, v_1, v_2) \tag{23}$$
$$E_N \mathrel{\widehat{=}} \exists\, \alpha \bullet (Q_N \wedge v_1' = F_N^1(\alpha, v_1, v_2) \wedge v_2' = F_N^2(\alpha, v_1, v_2)) \tag{24}$$
$$G_P \mathrel{\widehat{=}} \exists\, \beta \bullet Q_P(\beta, v_2, v_3) \tag{25}$$
$$E_P \mathrel{\widehat{=}} \exists\, \beta \bullet (Q_P \wedge v_2' = F_P^2(\beta, v_2, v_3) \wedge v_3' = F_P^3(\beta, v_2, v_3)) \tag{26}$$
$$G_{NR} \mathrel{\widehat{=}} \exists\, \gamma \bullet Q_{NR}(\gamma, w_1, w_2) \tag{27}$$
$$E_{NR} \mathrel{\widehat{=}} \exists\, \gamma \bullet (Q_{NR} \wedge w_1' = F_{NR}^1(\gamma, w_1, w_2) \wedge w_2' = F_{NR}^2(\gamma, w_1, w_2)) \tag{28}$$
$$G_{PR} \mathrel{\widehat{=}} \exists\, \delta \bullet Q_{PR}(\delta, w_2, w_3) \tag{29}$$
$$E_{PR} \mathrel{\widehat{=}} \exists\, \delta \bullet (Q_{PR} \wedge w_2' = F_{PR}^2(\delta, w_2, w_3) \wedge w_3' = F_{PR}^3(\delta, w_2, w_3)) \tag{30}$$

and thus

$$G_M \mathrel{\widehat{=}} \exists\, \alpha, \beta \bullet (Q_N(\alpha, v_1, v_2) \wedge Q_P(\beta, v_2, v_3) \wedge F_N^2 = F_P^2) \tag{31}$$
$$E_M \mathrel{\widehat{=}} \exists\, \alpha, \beta \bullet (Q_N \wedge Q_P \wedge v_1' = F_N^1 \wedge v_2' = F_N^2 \wedge v_2' = F_P^2 \wedge v_3' = F_P^3) \tag{32}$$

[9] [19] states that, instead of discharging (11-13), it suffices to prove the composite statement (22).

$$G_{MR} \mathrel{\widehat{=}} \exists \gamma, \delta \bullet (Q_{NR} \land Q_{PR} \land F^2_{NR} = F^2_{PR}) \tag{33}$$

$$E_{MR} \mathrel{\widehat{=}} \exists \gamma, \delta \bullet (Q_{NR} \land Q_{PR} \land w'_1 = F^1_{NR} \land w'_2 = F^2_{NR} \land w'_2 = F^2_{PR} \land w'_3 = F^3_{PR}) \tag{34}$$

We rewrite the theorem in expanded form, omitting redundant guard expressions by (5), i.e.

Given that $\underline{e_N \sqsubseteq e_{NR}}$:

$$J(v_1, w_1, w_2) \land v_2 = h(w_2) \land$$
$$\exists \gamma \bullet (Q_{NR}(\gamma, w_1, w_2) \land w'_1 = F^1_{NR}(\gamma, w_1, w_2) \land w'_2 = F^2_{NR}(\gamma, w_1, w_2)) \tag{35}$$
$$\Rightarrow \exists v'_1, v'_2 \bullet (\exists \alpha \bullet (Q_N(\alpha, v_1, v_2) \land v'_1 = F^1_N(\alpha, v_1, v_2) \land v'_2 = F^2_N(\alpha, v_1, v_2)) \land J(v'_1, w'_1, w'_2) \land v'_2 = h(w'_2))$$

... where the RHS can be simplified to ...

$$\exists \alpha \bullet (Q_N(\alpha, v_1, v_2) \land J(F^1_N(\alpha, v_1, v_2), w'_1, w'_2) \land F^2_N(\alpha, v_1, v_2) = h(w'_2)) \tag{36}$$

and $\underline{e_P \sqsubseteq e_{PR}}$:

$$K(v_3, w_3, w_2) \land v_2 = h(w_2) \land$$
$$\exists \delta \bullet (Q_{PR}(\delta, w_2, w_3) \land w'_2 = F^2_{PR}(\delta, w_2, w_3) \land w'_3 = F^3_{PR}(\delta, w_2, w_3)) \tag{37}$$
$$\Rightarrow \exists v'_2, v'_3 \bullet (\exists \beta \bullet (Q_P(\beta, v_2, v_3) \land v'_2 = F^2_P(\beta, v_2, v_3) \land v'_3 = F^3_P(\beta, v_2, v_3)) \land K(v'_3, w'_3, w'_2) \land v'_2 = h(w'_2))$$

... where the RHS can be simplified to ...

$$\exists \beta \bullet (Q_P(\beta, v_2, v_3) \land K(F^3_P(\beta, v_2, v_3), w'_3, w'_2) \land F^2_P(\beta, v_2, v_3) = h(w'_2)) \tag{38}$$

we must show $\underline{e_M \sqsubseteq e_{MR}}$:

$$J(v_1, w_1, w_2) \land K(v_3, w_3, w_2) \land v_2 = h(w_2) \land$$
$$\exists \gamma, \delta \bullet (Q_{NR}(\gamma, w_1, w_2) \land Q_{PR}(\delta, w_2, w_3) \land w'_1 = F^1_{NR}(\gamma, w_1, w_2) \land w'_2 = F^2_{NR}(\gamma, w_1, w_2) \land w'_2 = F^2_{PR}(\delta, w_2, w_3) \land w'_3 = F^3_{PR}(\delta, w_2, w_3)) \tag{39}$$
$$\Rightarrow \exists v'_1, v'_2, v'_3 \bullet (\exists \alpha, \beta \bullet (Q_N(\alpha, v_1, v_2) \land Q_P(\beta, v_2, v_3) \land v'_1 = F^1_N(\alpha, v_1, v_2) \land v'_2 = F^2_N(\alpha, v_1, v_2) \land v'_2 = F^2_P(\beta, v_2, v_3) \land v'_3 = F^3_P(\beta, v_2, v_3)) \land J(v'_1, w'_1, w'_2) \land K(v'_3, w'_3, w'_2) \land v'_2 = h(w'_2))$$

... where the RHS can be simplified to ...

$$\exists \alpha, \beta \bullet (Q_N(\alpha, v_1, v_2) \land Q_P(\beta, v_2, v_3) \land F^2_N(\alpha, v_1, v_2) = F^2_P(\beta, v_2, v_3) \land J(F^1_N(\alpha, v_1, v_2), w'_1, w'_2) \land K(F^3_P(\beta, v_2, v_3), w'_3, w'_2) \land F^2_P(\beta, v_2, v_3) = h(w'_2)) \tag{40}$$

Assuming the hypothesis (39) for the refinement of e_M, we can partition its terms into separate quantifications over γ and δ, and thus infer the hypotheses (35, 37) for the refinements of e_N, e_P respectively. The consequents of the component refinements

(36, 38) follow, and then we infer the result (40) directly by recombining the terms under a joint quantification over α, β. **QED**

4 Conclusion and Related Work

(De-)Compositional approaches to modelling and verification have been extensively studied for obvious reasons, and continue to be developed. We discuss only those most relevant to Event-B; whilst contemporary work on component- and service-based composition such as [6] is interesting, its application to Event-B remains for the future.

Following earlier work on temporal property verification on labelled transition systems (LTS) inspired by B [9,13], Kouchnarenko and Lanoix [16,17] investigated compositional verification in that LTS setting. For their expressive "constraint synchonized product" composition of components, preservation of both local and global invariants is shown, as well as compositionality of refinement. With stuttering behaviour allowed and non-increasing of deadlocks in refinement, their work is of interest to the Event-B community. While this work does not deal with these behavioural aspects of refinement - leaving that for the future - it does allow for intersecting state spaces, i.e. communication through shared variables. Kouchnarenko and Lanoix have disjoint state spaces but their work may be extensible to a message-passing composition like that of Butler [11].

Patterns and techniques for compositional/decompositional working with Event-B are in their infancy, reflecting the fact that they remain to be implemented in what is still a very recent language and method. Although the decompositional techniques of section 1 have been known for some years, and paper-based case studies have been published, e.g. [11], these techniques remain to be implemented by tools. Some progress is expected in this regard during project DEPLOY. For these more established techniques, and certainly for newer proposals such as ours, case study work is required to validate their utility, followed by prototype tool development to implement them.

In the short term we will investigate the extensibility of the results of this work. Obvious questions are (i) does the construction work for full behavioural Event-B refinement (as mentioned in sec. 1.3), (ii) under what conditions can features expressed in the more general syntax (6-8) be composed, and (iii) can we compose subject to less constrained gluing invariants than (17-18)? Beyond that we anticipate the proposal of more elaborate patterns of composition. Our proposal (per Fig. 2) gives a simple one-to-one feature refinement pattern, in general inadequate for elaborating an architectural model of the system. More flexibility is required in the elaboration of the modular arrangement of refinements. In the figure, for example, we can imagine different depths in the feature refinement chains, or feature decompositions: say that PR is refined by event-based decomposition into PR_{21}, PR_{22}, and each of these is further refined into PR_{31}, PR_{32} before fusing, together with NR, into MR.

Significant further tool infrastructure will ultimately be required to support reuse, i.e. the construction of system variants from different arrangements of feature composition and refinement. This includes inter alia system variant identification (in terms of components), feature refactoring, and proliferation of feature changes.

References

1. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. 15(1), 73–132 (1993)
2. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
4. 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)
5. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae 77(1-2) (2007)
6. Attiogbé, C., André, P., Ardourel, G.: Checking component composability. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 18–33. Springer, Heidelberg (2006)
7. Back, R.-J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distributed Computing 3(2), 73–87 (1989)
8. Back, R.J.R., Butler, M.: Fusion and simultaneous execution in the refinement calculus. Acta Informatica 35, 921–949 (1998)
9. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Ready-simulation is not ready to express a modular refinement relation. In: Maibaum, T.S.E. (ed.) FASE 2000. LNCS, vol. 1783, pp. 266–283. Springer, Heidelberg (2000)
10. Butler, M.: Stepwise refinement of communicating systems. Science of Computer Programming 27, 139–173 (1996)
11. Butler, M.: An approach to the design of distributed systems with B AMN. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 223–241. Springer, Heidelberg (1997)
12. Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. IEEE Software, 37–45 (November/December 1998)
13. Darlot, C., Julliand, J., Kouchnarenko, O.: Refinement preserves PLTL properties. In: Bert, D., P. Bowen, J., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 408–420. Springer, Heidelberg (2003)
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619 (1983)
15. Jones, C.B. (ed.): Intermediate report on methodology. Technical Report Deliverable 19, EU Project IST-511599 - RODIN (August 2006), `http://rodin.cs.ncl.ac.uk`
16. Kouchnarenko, O., Lanoix, A.: Refinement and verification of synchronized component-based systems. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 341–358. Springer, Heidelberg (2003)
17. Kouchnarenko, O., Lanoix, A.: Verifying invariants of component-based systems through refinement. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 289–303. Springer, Heidelberg (2004)
18. Abrial, J.R., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C.B., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., Leavens, G.T., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Proc. 5th Int. Conf. Generative Programming and Component Engineering, Portland, Oregon (2006)
19. Métayer, C., Abrial, J.-R., Voisin, L.: Event-B Language. Technical Report Deliverable 3.2, EU Project IST-511599 - RODIN (May 2005), `http://rodin.cs.ncl.ac.uk`

20. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
21. Poppleton, M.R.: Towards Feature-Oriented Specification and Development with Event-B. In: Sawyer, P., Paech, B., Heymans, P. (eds.) REFSQ 2007. LNCS, vol. 4542, pp. 367–381. Springer, Heidelberg (2007)
22. Potet, M.-L.: Spécifications et développements structurés dans la méthode B. Technique et Science Informatiques 22, 61–88 (2003)
23. Potet, M.-L., Rouzaud, Y.: Composition and refinement in the B-method. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 46–65. Springer, Heidelberg (1998)

Reconciling Axiomatic and Model-Based Specifications Reprised

Ken Robinson

[1] School of Computer Science & Engineering
The University of New South Wales
Sydney NSW 2052 Australia
k.robinson@unsw.edu.au
[2] National ICT Australia

Abstract. This paper is a reprise of a paper presented at ZB2000 that attempted to reconcile the worlds of model-based and axiomatic specification. The new paper uses the same problem, but treats it very differently in Event B. The development also serves as a short tutorial example in Event B.

Keywords: formal specification, axiomatic, model-based, refinement, classical B, Event B.

1 Introduction

An earlier paper [6] was based on a small example —that of a simple stack— used to explore the differences between axiomatic and model based specification using the original **B** [1], now referred to as Classical B. The exercise was carried out using the B-Toolkit[3]. In truth that exercise broke the usual rules of Classical B by refining a constant. This required a simple modification to the B-Toolkit.

The reprise of that same exercise seems appropriate for the ABZ08 conference, especially in regard to the AB part of the conference.

The reprise is carried out in Event B [2] using the Rodin toolkit[5]. Thus, the exercise is also a tutorial in Event B and a demonstration of the Rodin toolkit. As with many exercises that use Event B the final product —in the experience of the author— is more abstract, and in the case of this example the final model is much more axiomatic than its earlier attempt.

While the abstract modelling presented here may not generally be required in modelling in B, there are many aspects of the modelling of datatypes that are relevant to the more conventional modelling for which B is normally used. In that respect, the paper can be seen as a tutorial exercise in datatype modelling using Event B.

1.1 Axiomatic vs Model-Based Specification

Both axiomatic and model-based specification use axioms. The difference is one of degree. Most model-based specification use structures that are perceived to

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 223–236, 2008.

possess behaviours that readily support the required behaviours of the construct being modelled. This can mean that some behaviours of the model depend on assumed, and hence unproven, behaviour of the chosen structures.

This can be illustrated by considering the example to be pursued in the remainder of this paper. We will model something very simple: a stack. Commonly a stack would be modelled using a sequence, it being very clear that adding and removing elements to and from the same end of a sequence produces a last-in first-out behaviour that should yield a stack; but there is no abstract modelling of that behaviour itself.

In the rest of this paper we will first give a brief introduction to Event B and the Rodin toolkit. Then we introduce an abstract stack in which $STACK$ is an opaque set of stacks. There is no visible structure to the stack; instead we model all required behaviour of the stack with axioms.

We then refine the abstract stack to a concrete stack using a sequence. The relationship between the abstract and concrete stack is defined by more axioms.

The whole development is carried out on the Rodin toolkit and the presentation in this paper consists of markup of those machines interspersed with commentary. Thus the formal text in this paper is actual Event B text that has been analysed by the Rodin toolkit. The toolkit has also generated proof obligations and the discharge of those POs has been addressed.

Finally, the paper discusses some of the experiences of using Rodin.

2 Brief Overview of Event B

Event B is the most recent development of the B Method by Abrial. The original B is now referred to as Classical B. Event B models consist of the following components:

Context machines: Context machines contain the declaration and definitions of sets and constants. The definitions are given using axioms and common behaviours can be presented using theorems. Context machine can be *extended* by other context machines.

Machines: Machines can see context machines and contain variables and an invariant, modelling a machine state. The behaviour of a machine is specified by events, which are autonomous actions controlled by guards that decide when an event *may* fire. Machines can also contain theorems that use the invariants as axioms.

Refinements: Refinement machines present a refined state and refinements of the events of an abstract machine. During refinement, new events can be specified for realising a more concrete version of the refined event, or even for adding extra functionality that is "orthogonal" to that of the initial abstract events.

The mathematical toolkit of Event B is essentially the same as for Classical B, but the language is severely pared down making it feel more abstract than its parent.

The Rodin toolkit developed by the Rodin project is implemented on the Eclipse platform.

3 The Abstract Stack

A context machine, *StackAxiom*, is used to define the sets and constant functions that will be required for our abstract stack.

3.1 Stack Sets, Constants, Axioms and Theorems

The StackAxiom context machine contains declarations of $STACK$ the set of all stacks and $DATA$ the set of all data than can be placed on a stack.

CONTEXT : StackAxiom

SETS:

:	$STACK$	The set of stacks
:	$DATA$	Data that can be stored on the a stack

Constants consist of *emptystack* the empty stack, and functions *push*, *pop*, *tos* and *stackdepth* that abstractly model the expected stack behaviour.

CONSTANTS:

:	*emptystack*	The empty stack
:	*STACK1*	The set of non-empty stacks
:	*push*	The function that pushes an item of type DATA onto a stack
:	*pop*	The function that pops the topmost item on the stack
:	*tos*	A function that yields the value on top of the stack
:	*stackdepth*	A function that displays the depth of a stack

AXIOMS:

stkaxm1 :: $emptystack \in STACK$

stkaxm2 :: $STACK1 = STACK \setminus \{emptystack\}$

stkaxm3 :: $push \in STACK \times DATA \rightarrowtail\!\!\!\!\!\!\rightarrow STACK1$

stkaxm4 :: $pop \in STACK1 \rightarrow STACK$

stkaxm5 :: $tos \in STACK1 \rightarrow DATA$

stkaxm6 :: $\forall st, d \cdot st \in STACK \land d \in DATA \Rightarrow pop(push(st \mapsto d)) = st$

stkaxm7 :: $\forall st, d \cdot st \in STACK \land d \in DATA \Rightarrow tos(push(st \mapsto d)) = d$

$$stkaxm8 :: \; stackdepth \in STACK \to \mathbb{N}$$

$$stkaxm9 :: \; stackdepth(emptystack) = 0$$

$$stkaxm10 :: \; \forall st, d{\cdot}st \in STACK \wedge d \in DATA \Rightarrow stackdepth(push(st \mapsto d)) = stackdepth(st) + 1$$

$$stkaxm11 :: \; \forall stack, data{\cdot}stack \in STACK1 \wedge data \in DATA \Rightarrow stackdepth(pop(stack)) = stackdepth(stack) - 1$$

The informal stack properties expressed by the axioms are as follows.

stkaxm1: *emptystack* the stack that is empty.
stkaxm2: $STACK1$ is the set of stacks that contain at least one component.
stkaxm3: *push* is a total injection from $stack \mapsto data$ pairs to stacks. Different $(stack, data)$ pairs produce different stacks and all non-empty stacks can be reached by *push*.
stkaxm4: *pop* is a function takes any non-empty stack and produces a stack, in which the topmost item on the stack has been removed. All non-empty stacks can be popped. Popping different stacks can produce the same stack, so it is simply a total (non-injective) function from non-empty stacks. The set of all stacks related by push or pop produce a tree, whose root is the empty stack.
stkaxm5: *tos* is a function that when applied to a stack yields the topmost item on the stack.
stkaxm6: If you push an item onto a stack and then apply *pop* to the resultant stack you will get the stack as it was before the *push* operation.
stkaxm7: If you push an item onto a stack and apply *tos* to the resultant stack you will get the item that was just pushed.
stkaxm8: *stackdepth*, yields the *depth* of a stack, that is the net number of items pushed onto the stack.
stkaxm9: the stack depth of the empty stack is 0.
stkaxm10: pushing an item onto a stack increases the depth of the stack 1.
stkaxm11: popping a non-empty stack decreases the depth of the stack by 1.

Some of the properties that flow from the axioms will be presented as theorems. Some of the theorems are present to simplify discharge of proof obligations and others, such as stkthm3 express expectations of the stack construct.

THEOREMS:

$$stkthm1 :: \; emptystack \notin ran(push)$$

$$stkthm2 :: \; dom(pop) = STACK \setminus \{emptystack\}$$

$$stkthm3 :: \; \forall st, d{\cdot}st \in STACK \wedge d \in DATA \Rightarrow push(st \mapsto d) \neq emptystack$$

$$stkthm4 :: \; \forall st{\cdot}st \in STACK \Rightarrow (st \in dom(pop) \Leftrightarrow st \neq emptystack)$$

$stkthm5 :: \ \forall st \cdot st \in STACK \Rightarrow (st \in dom(tos) \Leftrightarrow st \neq emptystack)$

stkthm1: a trivial consequent of stkaxm2;
stkthm2: a trivial consequent of stkaxm3;
stkthm3: the push function will never produce an empty stack;
stkthm4: the pop function does not apply to the empty stack, but is applicable to all non-empty stacks;
stkthm5: the tos function does not apply to the empty stack, but is applicable to all non-empty stacks.

3.2 Stack Machine

The preceding context machine presents the mathematical constructs necessary to specify Stack, a stack machine. The abstract stack machine uses events for the stack operations of Push and Pop. In Classical B these would be modelled using machine operations. A stack machine might normally have a Top operation that yields the topmost element on the stack. Here we have abstracted this with a function on stacks, *tos*, that yields the top of the stack. We also have an abstract function *stackdepth* that when applied to a stack yields its depth. It should be noted that events in Event B do not have the capability of returning results that is provided by Classical B operations.

The following machine could be augmented by events that simply invoke the functions *tos* and *stackdepth* and store the results in variables. This is a trivial extension and has been omitted.

MACHINE : Stack
SEES: StackAxiom

VARIABLES:
: *stack*

INVARIANTS:
inv1 :: $stack \in STACK$

inv1: *stack* is one stack;

EVENTS:
Initialisation:
begin:
act1 :: $stack := emptystack$
end:

The initialisation of the Stack machine consists of setting the stack to the empty stack.

Push: $\widehat{=}$
  any:
    : *data*
  where:
    grd1 :: $data \in DATA$
  then:
    act1 :: $stack := push(stack \mapsto data)$
  end:

Given an item of data as a parameter, the Push event can execute at any time to push that data onto the stack. As a side-effect, *stackdepth* will increment and *tos* becomes the value of the data item just pushed onto the stack.

Pop: $\widehat{=}$
  when:
    grd1 :: $stack \neq emptystack$
  then:
    act1 :: $stack := pop(stack)$
  end:

Whenever the stack is not empty, Pop can remove the topmost element from the stack: *stackdepth* concurrently decrements *tos* resets to the new top-of-stack.

At all times, when the stack is not empty, *tos* yields the topmost element of the stack.

4 The Concrete Stack

4.1 Sequences: Preparing for the Concrete Stack

We intend refining the abstract stack machine to a "concrete stack" using a sequence. Currently, Event B does not have sequences as a special construct in its mathematical toolkit.

Of course it is easy to model sequences as coherent partial functions and we will also add some functions on sequences that we will need. SeqAxiom is specified as an extension of StackAxiom.

CONTEXT : SeqAxiom
EXTENDS : StackAxiom

CONSTANTS:

:	*SEQ*	The set of sequences
:	*SEQ1*	The non-empty sequences
:	*emptyseq*	The empty sequence
:	*append*	Add an element to the end of a sequence
:	*front*	Sequence with last element removed
:	*last*	The last element of a non-empty sequence
:	*delete*	A function to remove any element of a sequence

AXIOMS:

seqaxm1 :: $SEQ \subseteq \mathbb{N}_1 \pfun DATA$

seqaxm2 :: $\forall s \cdot s \in SEQ \Rightarrow finite(s)$

seqaxm3 :: $\varnothing \in SEQ$

seqaxm4 :: $SEQ1 = SEQ \setminus \{\varnothing\}$

seqaxm5 :: $\forall s \cdot s \in SEQ \Rightarrow dom(s) = 1 \mathinner{..} card(s)$

seqaxm6 :: $emptyseq \in SEQ$

seqaxm7 :: $emptyseq = \varnothing$

seqaxm8 :: $append \in (SEQ \times DATA) \to SEQ1$

seqaxm9 :: $\forall s, d \cdot s \in SEQ \land d \in DATA$
$\Rightarrow append(s \mapsto d) = (s \cup \{card(s) + 1 \mapsto d\})$

seqaxm10 :: $front \in SEQ1 \to SEQ$

seqaxm11 :: $\forall s \cdot s \in SEQ1 \Rightarrow front(s) = \{card(s)\} \ndres s$

seqaxm12 :: $last \in SEQ1 \to DATA$

seqaxm13 :: $\forall s \cdot s \in SEQ1 \Rightarrow last(s) = s(card(s))$

seqaxm14 :: $delete \in \mathbb{N}_1 \to (SEQ \to SEQ)$

seqaxm15 :: $\forall p, s \cdot s \in SEQ1 \land p \in dom(s)$
$\Rightarrow card(delete(p)(s)) = card(s) - 1$

seqaxm16 :: $\forall p, s \cdot s \in SEQ1 \land p \in dom(s)$
$\Rightarrow dom(delete(p)(s)) = 1 \mathinner{..} card(s) - 1$

seqaxm17 :: $\forall p, s, n \cdot s \in SEQ1 \land p \in dom(s) \land n \in 1 \mathinner{..} p - 1$
$\Rightarrow delete(p)(s)(n) = s(n)$

seqaxm18 :: $\forall p, s, n \cdot s \in SEQ \land p \in dom(s) \land n \in p \mathinner{..} card(s)$
$\Rightarrow delete(p)(s)(n) = s(n + 1)$

seqaxm1: SEQ is the set of all sequences that we will be using;
seqaxm2: every sequence will be finite;
seqaxm3: the empty set is a (empty) sequence;
seqaxm4: $SEQ1$ is the set of all non-empty sequences;
seqaxm5: every sequence s has a domain that is $1 \mathinner{..} card(s)$, that is the domain is coherent;
seqaxm6: *emptyseq* is a sequence;
seqaxm7: the *emptyseq* is the empty set;
seqaxm8, seqaxm9: $append(s \mapsto x)$ puts x on the end of s;
seqaxm10, seqaxm11: $front(s)$ yields a subsequence of s that excludes the last element;
seqaxm12: the domain of *last* is all sequences except the empty sequence;
seqaxm13: $last(s) = s(size(s))$ for every non-empty sequence s;
seqaxm14: *delete* takes an ordinal and a sequence and produces a sequence with possibly one element deleted;

seqaxm15: the size of a sequence after deletion is one less than before;
seqaxm16: the domain of the sequence after a deletion is one less that before;
seqaxm17: the position of items before the deletion position does not change;
seqaxm18: all items after the deletion position are shifted one position down;

THEOREMS:

$$seqthm1 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow dom(append(s \mapsto d)) = 1 \mathinner{..} card(s) + 1$$

$$seqthm2 :: \forall s \cdot s \in SEQ1 \Rightarrow dom(front(s)) = 1 \mathinner{..} card(s) - 1$$

$$seqthm3 :: \forall s \cdot s \in SEQ \wedge s \neq emptyseq \Rightarrow s \in dom(front)$$

$$seqthm4 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow append(s \mapsto d) \neq emptyseq$$

$$seqthm5 :: \forall s \cdot s \in SEQ1 \Rightarrow card(s) \in dom(s)$$

$$seqthm6 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow front(append(s \mapsto d)) = s$$

$$seqthm7 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow last(append(s \mapsto d)) = d$$

$$seqthm8 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow append(s \mapsto d) \in SEQ1$$

$$seqthm9 :: dom(front) = SEQ1$$

$$seqthm10 :: dom(last) = SEQ1$$

seqthm1: the domain of $append(s \mapsto d)$ is $1 \mathinner{..} card(s) + 1$;
seqthm2: the domain of $front(s$ is $1 \mathinner{..} card(s) - 1$;
seqthm3: any sequence that is not empty is in $dom(front)$;
seqthm4: $append(s \mapsto d)$ is never empty;
seqthm5: if s is not empty then $card(s) \in dom(s)$;
seqthm6: for a sequence s, $front(append(s \mapsto d)) = s$;
seqthm7: for a sequence s, $last(append(s \mapsto d)) = d$;
seqthm8: $append(s \mapsto d)$ is a non-empty sequence;

4.2 The Recursive Structure of Stacks

We have not yet fully captured the recursive nature of an abstract stack, that is a stack whose size is greater than 1 actually contains a number of stacks, given by $pop(stack), pop(pop(stack)), \ldots$ etc. It is clear that, at any point, an abstract stack will have been formed from finite sequences of interleaved *push* and *pop* events. Abstract stacks being opaque means that the inner structure can only be revealed by repeated application of functions.

Significantly, this recursive structure is evident in discharging proof obligations where a property like $stack \neq emptystack$ may become $pop^n(stack) \neq emptystack$ in a different context. Thus we need to be able to present all stack properties in terms of finite recursions of the basic two stack functions, *push* and *pop* combined with the non-recursive functions *tos* and *stackdepth*.

Thus we need to be able to iterate a function f:

$$\begin{array}{l} \mathit{Given}\ \ f \in X \nrightarrow X, x \in X \\ \qquad f^0(x) = x \\ \qquad f^n(x) = f^{n-1}(f(x)) \end{array}$$

More generally iteration is defined similarly on relations.

However currently, Event B does not have functional or relational iteration, so we will give our own functions.

4.3 Sequence-Stack Axioms

We need to prepare the ground for refining the abstract stack to a sequence.

Paralleling the requirement to iterate function application on abstract stacks we will also need to be able to iterate functions on our concrete sequence stack. Unfortunately, because Event B is first-order, we cannot give a generic definition of iteration. We therefore define $STACKIter$ and $SEQIter$ for iterating functions, respectively on values of type $STACK$ and SEQ.

$CONTEXT$: StackSeqAxiom
$EXTENDS$: SeqAxiom

$AXIOMS$:

stsaxm1 :: $STACKIter \in (STACK \nrightarrow STACK) \rightarrow (\mathbb{N} \nrightarrow (STACK \rightarrow STACK))$

stsaxm2 :: $SEQIter \in (SEQ \nrightarrow SEQ) \rightarrow (\mathbb{N} \nrightarrow (SEQ \rightarrow SEQ))$

stsaxm3 :: $\forall f, n, s \cdot f \in STACK \nrightarrow STACK \land n \in \mathbb{N} \land s \in STACK \land n = 0 \Rightarrow STACKIter(f)(n)(s) = s$

stsaxm4 :: $\forall f, n, s \cdot f \in STACK \rightarrow STACK \land n \in \mathbb{N}_1 \land s \in dom(f) \Rightarrow STACKIter(f)(n)(s) = STACKIter(f)(n-1)(f(s))$

stsaxm5 :: $\forall f, n, s \cdot f \in SEQ \rightarrow SEQ \land n \in \mathbb{N} \land s \in SEQ \land n = 0 \Rightarrow (SEQIter(f))(n)(s) = s$

stsaxm6 :: $\forall f, n, s \cdot f \in SEQ \rightarrow SEQ \land n \in \mathbb{N}_1 \land s \in dom(f) \Rightarrow (SEQIter(f))(n)(s) = (SEQIter(f))(n-1)(f(s))$

stsaxm1, stsaxm3: $STACKIter(f)(0)(s) = s$;
stsaxm1, stsaxm4: $STACKIter(f)(n)(s) = STACKIter(f)(n-1)(f(s))$, if $n \neq 0$;
stsaxm2, stsaxm5: $SEQIter(f)(0)(s) = s$;
stsaxm2, stsaxm6: $SEQIter(f)(n)(s) = SEQIter(f)(n-1)(f(s))$, if $n \neq 0$.

The following theorems record equivalences between various compositions of stack and sequence operations.

THEOREMS:

$$ststhm1 :: \forall s, st, d \cdot s \in SEQ \wedge st \in STACK \wedge d \in DATA \Rightarrow last(append(s \mapsto d)) = tos(push(st \mapsto d))$$

$$ststhm2 :: dom(STACKIter) = STACK \nrightarrow STACK$$

$$ststhm3 :: dom(SEQIter) = SEQ \nrightarrow SEQ$$

$$ststhm4 :: \forall s, d \cdot s \in SEQ \wedge d \in DATA \Rightarrow card(append(s \mapsto d)) = card(s) + 1$$

ststhm1: for a sequence *seq* and stack *stack*, s,
$last(append(s \mapsto d)) = tos(push(stack \mapsto d))$.

4.4 Stack Refinement

We will now refine the abstract stack machine, Stack, to a concrete machine in which the stack is modelled as a sequence, SeqStack.

MACHINE : SeqStack
REFINES: Stack
SEES: StackSeqAxiom

VARIABLES:
: *seqstack*

Notice that the state consists of only one variable, *seqstack*. Since this is a sequence the size of the stack is implicit in the length of the sequence.

INVARIANTS:

$$inv1 :: seqstack \in SEQ$$

$$inv2 :: seqstack \neq emptyseq \Rightarrow tos(stack) = last(seqstack)$$

$$inv3 :: stack \neq emptystack \Leftrightarrow seqstack \neq emptyseq$$

$$inv4 :: seqstack \neq emptyseq \Rightarrow (\forall n \cdot n \in \mathbb{N}_1 \wedge n \leq card(seqstack) \Rightarrow last(SEQIter(front)(n-1)(seqstack)) = tos(STACKIter(pop)(n-1)(stack)))$$

$$inv5 :: seqstack \neq emptyseq \Rightarrow (\forall n \cdot n \in \mathbb{N}_1 \wedge n \leq card(seqstack) \Rightarrow STACKIter(pop)(n-1)(stack) \neq emptystack)$$

$$inv6 :: seqstack \neq emptyseq \Rightarrow (\forall n \cdot n \in \mathbb{N}_1 \wedge n \leq card(seqstack) \Rightarrow SEQIter(front)(n-1)(seqstack) \neq emptyseq)$$

$$inv7 :: stackdepth(stack) = card(seqstack)$$

$$inv8 :: stack \neq emptystack \Rightarrow tos(stack) = last(seqstack)$$

The invariant of this refinement, of course contains the refinement relation, which is concerned with ensuring that the concrete stack, *seqstack*, has behaviour that is consistent with that of the abstract stack, *stack*. The following comments explain the significance of each of the invariants.

inv1: *seqstack* is a finite sequence;
inv2: if *seqstack* is not empty, then *tos* is $last(seqstack)$, otherwise it is an arbitrary element of $DATA$;
inv3: $stack \neq emptystack \Leftrightarrow seqstack \neq emptyseq$;
inv4: for any value of n less than the length of *seqstack*, applying $front$ to the sequence n times and taking *last* of the resulting sequences yields the same value as applying *pop* n times to the abstract *stack* and then taking *tos* of the resulting stack;
inv5: applying *pop* to the abstract *stack* any number of times less than the length of *seqstack* does not yield *emptystack*;
inv6: applying $front$ to *seqstack* any number of times less than the length of *seqstack* does not yield the empty sequence;
inv7: the depth of the abstract *stack* is equal to the length of *seqstack*.
inv8: the top of the abstract (non-empty) stack is equal to the last element of (non-empty) sequence.

THEOREMS:

$$\begin{aligned} thm1 :: \ & \forall n, d \cdot n \in \mathbb{N} \wedge n \leq card(seqstack) \wedge d \in DATA \\ & \Rightarrow last(SEQIter(front)(n)(append(seqstack \mapsto d))) \\ & = tos(STACKIter(pop)(n)(push(stack \mapsto d))) \end{aligned}$$

$$\begin{aligned} thm2 :: \ & \forall n, d \cdot n \in \mathbb{N} \wedge n \leq card(seqstack) \wedge d \in DATA \\ & \Rightarrow STACKIter(pop)(n)(push(stack \mapsto d)) \neq emptystack \end{aligned}$$

$$\begin{aligned} thm3 :: \ & \forall n, d \cdot n \in \mathbb{N} \wedge n \leq card(append(seqstack \mapsto d)) \wedge d \in DATA \\ & \Rightarrow last(((SEQIter(front))(n-1))(append(seqstack \mapsto d))) \\ & = tos(((STACKIter(pop))(n-1))(push(stack \mapsto d))) \end{aligned}$$

$$\begin{aligned} thm4 :: \ & seqstack \neq emptyseq \Rightarrow (front(seqstack) \neq emptyseq \\ & \Rightarrow last(front(seqstack)) = tos(pop(stack))) \end{aligned}$$

thm1: clearly this is a special case of *inv5* combined with *axm5* for the *Stack-Axiom*, the axioms for *SeqAxioms* and the axioms for $STACKIter$ and $SEQIter$;
thm2: an expression of the requirement that an abstract *stack* has as many "levels" as its refinement *seqstack*;
thm3, thm4: special cases of more general axioms.

EVENTS:
Initialisation:
begin:
act1 :: $seqstack := emptyseq$
end:

Pop: $\widehat{=}$
Refines: Pop
when:
$grd1 :: seqstack \neq emptyseq$
then:
$act1 :: seqstack := front(seqstack)$
end:

Push: $\widehat{=}$
Refines: Push
any:
$: data$
where:
$grd1 :: data \in DATA$
then:
$act1 :: seqstack := append(seqstack \mapsto data)$
end:

4.5 An Explanation of the Sudden Appearance of Recursion

The appearance of recursion was justified in sec4.2 by the recursive nature of the abstract. This is surprising as that structure is present in the abstract stack and nothing to specifically deal with this was introduced, or indeed required in the abstract stack model. There were 11 proof obligations generated for the StackAxiom machine and 2 for the Stack machine, and they were all discharged automatically.

However, the requirement for the recognition of the recursive structure is forced by the proof obligations generated for the SeqStack machine and derives from the refinement requirement as follows.

The functions $tos/last$ and $pop/front$ are partial and cannot be applied to empty stacks,sequences. The refinement is required to verify the outcomes of $tos(pop(stack))/last(front(seqstack)$, which raise the checking that $pop(stack)/front(seqstack)$ are not empty. So it is clear that there will need to be invariants concerned with things like emptiness *one level* down. But as soon as rules are written for descending to the next lower level of the stacks, the concern recurses, and hence there is a requirement for general (but finite) iterations of the stack functions. Hence, the axioms seen in the SeqStackAxiom machine were originally produced in response to proof obligations.

5 Other Aspects of the Development

We have presented a development of a simple stack that is significantly based on axioms. The development listing that is used as the basis of this paper is dominated by axioms and theorems. The abstractness of this exercise is very

suited to Event B, and my general experience of using Event B is that it seems to be easier to be abstract. In retrospect, it is always the case that a very similar development could have been given in Classical B, but there is a tendency for that not to happen.

5.1 Consistency

Care has been taken with the expression of the axioms and theorems to use consistent expressions for concepts. Provers can be quite sensitive to inconsistency and (interactive) proof can become more difficult as a result of inconsistency in the expression of the same concept or property. While producing the development for this paper the formulation has undergone a number of changes; often not in content, but in expression. A very early version defined *size* as a synonym for *card*, but that has been dropped. It was introduced purely for cosmetic reasons, but Event B/Rodin does not provide an aliasing facility, so the only way of introducing such a synonym seems to be by universal quantification, and that introduces a significant disadvantage into proof discharge.

5.2 Proof Obligations

The statistics on the proof obligations are

Machine	**Total**	**Auto**
hline StackAxiom	11	11
Stack	2	2
SeqAxiom	25	15
StackSeqAxion	10	2
SeqStack	41	8

At this stage no statistics have been given on interactive proof or review. The Rodin provers allow a PO to be "reviewed": that is the PO can be examined carefully and review might be selected. Review is not equivalent to proved; it remains tagged as reviewed until it is proved interactively or automatically. All of the remaining undischarged proof obligations have been at least reviewed; many have been proved interactively, but changes to the development have left the obligations undischarged at this stage. The Rodin provers —there are a number of them— are generally good, but they tend to be uneven, sometimes unable to discharge quite simple lemmas. Particularly annoying are the "well-definedness" (WD) proof obligations, that generally are not discharged automatically. This development uses high order functions and the provers continuously require discharge of WD proof obligations of the form $x \in dom(f)$ and this becomes particularly onerous for higher order functions with such proofs being requested at each level of the function.

It is asserted that all POs are either proved or able to be proved.

5.3 General Experience of the Rodin Toolkit

The Rodin toolkit is generally very pleasant to use and very productive. However for developments which go through a significant number of revisions the editing facilities are much inferior to many standard editors.

It must also be said, that Event B is a very rewarding formal modelling method to use. It encourages abstraction and hence enables top-down modelling.

6 Conclusion

We have reprised the earlier exercise of presenting a B model with a strong axiomatic basis. This has been very successful and has produced a better outcome than the earlier exercise. It can be remarked that the solution presented here was always available in Classical B.

Acknowledgements

I wish to acknowledgement some very useful comments from the reviewers.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: $B^\sharp$: Towards a Synthesis between Z and B. In: ZB 2003: Formal Specification and Development in Z and B, pp. 168–177 (2003)
3. B-Core(UK) Ltd., Oxford Science Park, Oxford UK. B-Toolkit, release 3.0 edition (1996)
4. P. Bowen, J., Dunne, S., Galloway, A., King, S. (eds.): B 2000, ZUM 2000, and ZB 2000. LNCS, vol. 1878. Springer, Heidelberg (2000)
5. Romanovsky, A., et al.: RODIN: Rigorous Open Development Environment for Complex Systems. Technical report, University of Newcastle upon Tyne, UK (2004–2007), `http://www.rodin.cs.ncl.ac.uk`
6. Robinson, K.: Reconciling Axiomatic and Model-Based Specifications Using the B Method. In: Bowen, J.P., et al. (eds.) [4], pp. 95–106 (2000)

A Verifiable Conformance Relationship between Smart Card Applets and B Security Models

Frédéric Dadeau[1], Julien Lamboley[2], Thierry Moutet[2], and Marie-Laure Potet[2]

[1] Laboratoire d'Informatique de Franche-Comté, F-25030 Besançon cedex,
[2] Vérimag, centre équation, 2 avenue de Vignate – F-38610 Gières
dadeau@lifc.univ-fcomte.fr,
{Julien.Lamboley,Thierry.Moutet,Marie-Laure.Potet}@imag.fr

Abstract. We propose a formal framework based on the B method, that supports the development of secured smart card applications. Accordingly to the Common Criteria methodology, we start from a formal definition and modelling of security policies, as access control policies. At the end of the development process, smart card applications are implemented in a standardized way, based on both the life cycle of smart card applets and the APDU protocol. In this paper, we define a conformance relationship that aims at establishing how smart card applications can be related to security requirement models. This embraces both the notions of security conformance as well as traceability allowing to relate basic events appearing at the level of applications with abstract security policies. This approach has been developed in the RNTL POSÉ project[1], involving a smart card issuer, Gemalto.

1 Introduction

Smart cards play an important role in the information systems security. They supply a medium for authentication, confidentiality and integrity. Security in smart cards is based on hardware mechanisms and operating system protections and, at the level of applications, on some security properties that can be established. Nowadays, more and more software applications are embedded on smart cards, as electronic wallets, electronic passports or administrative services.

Because smart cards become a central piece of every day citizen security, it is crucial to produce some assurances in terms of security. The Common Criteria standard is a norm allowing to produce confidence that the process of specification, implementation and evaluation has been conducted in a rigorous and standard manner. In the Common Criteria approach, security is specified as a set of security functional components [6] and the development process must then supply some assurances relatively to these security requirements [7]. The result is a level of confidence (called EAL for Evaluation Assurance Level), based on the measures taken during the development process. Common Criteria evaluations thus rely on the principles of modelling and presentation of evidences, which explain how security functionalities are really guaranteed. EAL are numbered

[1] Work supported by the RNTL POSE project (ANR-05-RNTL-01001).

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 237–250, 2008.

from 1 to 7, depending on the the precision attached to models and evidences. Higher levels are based on semi-formal and formal modellings.

Due to the increasing number of smart card applications, methodologies must be elaborated, in order to dispose of certifying processes which can be reproduced and automated. In the national French POSÉ project, dedicated to a model based testing approach for security policies, a B formal framework has been developed in order to relate smart card applications to security requirements. This paper presents this framework and extends it in order to capture the APDU level implementations, which was encapsulated in the testing tool used in the POSÉ. Section 2 describes the context of smart card development and access control specification. Section 3 describes the proposed B framework dedicated to the conformance relationship. Then, Section 4 illustrates our approach for smart card APDU level implementations. Finally, conclusion and perspectives are presented in Sect. 5.

2 The Context

In smart card applications, access control policies are a central piece of security because they ensure data protection. In the Common Criteria norm, the FDP class (user Data Protection) is relative to access control security exigences. From EAL 6 (Common Criteria version 3.1 [7]) a formal definition of security requirements is expected, called SPM (for Security Policy Modelling). This formalization is exploited in several ways. First, policies are stated in a precise and unambiguous manner, helping in that developers and certifying evaluators. Second, some properties relative to the security policies can be established, as their consistency. Finally, this model is exploited to relate functional specification, functional testing and vulnerabilities analysis to security requirements. The proposed B framework will address this relationship, allowing in that to produce assurance evidences.

2.1 Security Model

Access control which is considered here is the control of *subjects* executing some *operations* on some protected *objects*. Permissions can depend on *security attributes*. In smart card applications, subjects generally correspond to the type of authentication and access control depends on the life cycle of the card or the applet.

In our approach, a security model is composed of two B models: a rule-based model describing which subjects are authorized to execute which operations on which objects, and a dynamic model describing how subjects, objects and security attributes dynamically evolve. From a rule-based model and a dynamic model, a security kernel, enforcing the rules of the first model on the second one, can be automatically generated by the Meca tool [11]. We illustrate our approach with the example of an electronic purse in which some operations may only be executed from specific terminals that represent the subjects. We distinguish three kinds of subjects: administrative terminals dedicated to personalization, bank and pda terminals. The life cycle of the card (personalization, in use or invalid

mode) and the boolean variable `isHoldAuth`, indicating if the card holder has been authenticated, constitute the security attributes. Here are two examples:

$$(\texttt{mode}{=}\texttt{USE} \Rightarrow (\texttt{BANK} \mapsto \texttt{checkPin}) \in \texttt{permission}) \wedge$$
$$(\texttt{mode}{=}\texttt{USE} \wedge \texttt{isHoldAuth}{=}\texttt{TRUE} \Rightarrow (\texttt{BANK} \mapsto \texttt{credit}) \in \texttt{permission})$$

The dynamic model describes how security attributes evolve. Generally, this model is non deterministic due to the fact that some functional values are not modeled, such as the internal pin value. For instance, the `checkPin` operation can result in two different behaviors depending on whether the pin verification succeeds or fails. The `status` result is a witness of the internal behavior. The security model, obtained from Meca, is built from the dynamic model adding security conditions and the witness `sr` stating if the operation execution is authorized or not (`sr:=OK` or `sr:=KO`). Figure 1 describes this model.

2.2 Smart Card Applications

The APDU protocol (Application Protocol Data Unit) [12] governs the communications between cards and terminals. Terminals send APDU commands and cards respond with a response APDU. A command APDU takes the following form:

CLA	INS	P1	P2	Lc	Data Field	Le

The invariant `Lc = size(Data Field)` is verified when APDU commands are received or built. The field CLA is used to identify an application class and INS indicates the instruction code. P1 and P2 are parameter bytes. Lc denotes

```
MACHINE e_purse_security
SETS SUBJECTS = {ADMIN, BANK, PDA}; MODE = {PERSO, USE, INVALID}
VARIABLES subject, mode, isHoldAuth
INVARIANT subject ∈ SUBJECTS ∧ mode ∈ MODE ∧ isHoldAuth ∈ BOOL
INITIALISATION subject:∈SUBJECTS || mode,isHoldAuth:=PERSO,FALSE
OPERATIONS
   sr, status ← checkPin(p) ≙
     PRE p ∈ TAB_BYTE THEN
       IF mode=USE ∧ subject=BANK
       THEN
         CHOICE isHoldAuth:=TRUE || status:=success || sr:=OK
         OR isHoldAuth:=FALSE || status:=failure  || sr:=OK
         END
       ELSE sr:=KO
       END
     END
   ...
   END
```

Fig. 1. Security Model for the Electronic Purse

the number of bytes in the data field of the command APDU. Le denotes the maximum number of bytes expected in the data field of the response APDU. A response APDU takes the form:

Data Field	SW1	SW2

SW1 and SW2 are status words denoting the results of the command. Values of the status words are normalized. For instance, `9000` indicates that the commands terminated in the right way. In the framework of JavaCard, the Java Card Runtime Environment (JCRE) supplies the `APDU` class that manages the APDU buffer [19]. The `ISOException` class also supplies the *throwIt(short)* method that builds an APDU response with a status word corresponding to the parameter. Here is a JavaCard implementation of the `checkPin` operation. Object `hpc` (for holder pin code) is an instance of the `OwnerPin` class, relative to the management of pin codes. A try counter is associated to the `hpc` variable. Contrary to the security model, the number of failing tries is limited.

Error codes are used to set together fields SW1 and SW2 that will be denoted after by SW. `SW_TERMINAL_DENIED` and `SW_MODE_INCORRECT` are two error codes defined at the level of application (respectively as 6810 and 6820). `SW_PIN_BLOCKED` signals when no new try is possible. Other error codes are standardized ones (Interface ISO7816 of [19]). When no error is raised, the APDU

```
private void checkPin(APDU apdu) {
    byte[] apduBuf = apdu.getBuffer();
    // test the conformance of the parameter values
    if (apduBuf[ISO7816.OFFSET_P1] != (byte)0x00
         || apduBuf[ISO7816.OFFSET_P2] != (byte)0x00)
      ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
    // test the conformance of the expected Lc value
    short lc = apdu.setIncomingAndReceive();
    if (lc != 2)
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    // security conditions on terminal and mode
    if ((terminal != BANK))
      ISOException.throwIt(SW_TERMINAL_DENIED);
    if (mode_ != USE)
      ISOException.throwIt(SW_MODE_INCORRECT);
    // verification of the remaining tries number
    if (hpc_.getTriesRemaining() == 0)
      ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
    //verification of the pin code
    if (!hpc_.check(apduBuf, ISO7816.OFFSET_CDATA, (byte) lc)) {
       hpc_.reset();
       if (pin.getTriesRemaining() == 0)
       ISOException.throwIt(SW_PIN_BLOCKED);
       else ISOException.throwIt(ISO7816.SW_WRONG_DATA); } }
```

Fig. 2. APDU implementation for the Electronic Purse

response is implicitly updated with `SW_NO_ERROR`, i.e. 9000. We now define a framework to establish how such an implementation can be related to a security model, as introduced in Sect. 2.1.

3 Conformance Relationship

Our theoretical framework is based on the three following notions: (*i*) a notion of admissible traces associated to a model, (*ii*) a notion of mapping relating the (abstract) level of controlled operations with operations of the implementation level, and, (*iii*) a notion of conformance of an implementation w.r.t. a security model, through the mapping relationship. In the POSÉ project, the implementation level was specified in B in order to produce functional tests with the help of the LTG tool [13]. Then, security models are also expressed using B, and the conformance relationship can be defined using B theoretical tools. We first briefly recall some background for B.

3.1 Some B Notions

Generalized Substitutions, that are used to describe behaviors, can be defined by the Weakest Precondition semantics, introduced by E.W. Dijkstra [9], and denoted here by $[S]R$. The following predicates are attached to generalized substitutions (x being the state variables attached to the substitution S and x' the values of x after the substitution):

$\mathsf{trm}(S)$	$\hat{=}$	$[S]true$	termination
$\mathsf{prd}_x(S)$	$\hat{=}$	$\neg[S]\neg(x' = x)$	before-after predicate
$\mathsf{fis}(S)$	$\hat{=}$	$\exists x\ \mathsf{prd}_x(S)$	feasibility

Operation definitions are of the form $o \leftarrow op(i) \mathrel{\hat{=}}$ PRE P THEN S END. An operation is characterized by its termination and its before-after predicate. A call of the form $r \leftarrow op(v)$ can be defined by [5]:

VAR i, o IN $i := v$; PRE P THEN S END ; r:= o END

The Event B extension [2], dedicated to dynamic aspects, is based on another execution model. Events are of the form SELECT P THEN S END. An event is characterized by its before-after predicate and its feasibility, called here its guard. As soon as the guard holds, the event can be enabled.

Abstract models can be proved and refined. The refinement process consists in building a more concrete model and establishing the refinement relation. The refinement is based on a gluing invariant linking abstract and concrete variables. Refinement proof obligations consists in showing that the concrete initialization refines the abstract one, and that each concrete operation refines its abstract definition. A substitution S is refined by a substitution T, with respect to the gluing invariant L ($S \sqsubseteq_L T$) if and only if:

$$L \wedge \mathsf{trm}(S) \Rightarrow [T]\neg[S]\neg L$$

3.2 Security Traces

Access control being relative to the control of operation execution, security policies can be characterized as the set of admissible traces. A similar approach is adopted by F. Schneider [17] who characterizes access control by security automata. Traces can be syntactically represented as sequences of occurrences of execution calls, stated as triplets (op, v, r) where op is the name of an operation, v a valuation of input parameters and r a valuation of output parameters. Then, a trace associated to a model M is written:

$$< init \ ; \ (op_1, v_1, r_1) \ ; \ \ldots \ ; \ (op_n, v_n, r_n) >$$

with $init$ the initializing substitution of M and op_i an operation of M. Intuitively, an occurrence (op, v, r) has to be interpreted as the observation of the execution of the operation op called with the value v and producing the resulting value r. Furthermore, this execution only takes place in a state where the precondition holds, as it is required when termination is taken into account.

Definition 1 (Call occurrence). *Let $o \leftarrow op(i)$ be an operation defined by the substitution* PRE P THEN S END*. The event $exec(op, v, r)$, corresponding to the execution of a call $op(v)$ returning the value r, can be defined by the substitution:*

SELECT $[i := v]P$ THEN
VAR o IN $[i := v]S$; SELECT $(r = o)$ THEN skip END END
END

Substitution into substitution, as $[i := v]S$, is defined as in [1]. Here, we have $\mathsf{prd}(exec(op, v, r)) \equiv \exists \ o' \ [i := v](P \wedge \ \mathsf{prd}(S) \wedge \ o' = r)$, that exactly describes the observation of an operation call for input v producing the value r as result. We now define admissible traces.

Definition 2 (Admissible trace). *Let t be a trace of the form $< init; (op_1, v_1, r_1) \ ; \ \ldots \ ; \ (op_n, v_n, r_n) >$. This trace t is admissible if and only if the following condition holds:*

$$\mathsf{fis}(init \ ; \ exec(op_1, v_1, r_1) \ ; \ \ldots ; exec(op_n, v_n, r_n))$$

In the following, we denote by T_M the set of admissible traces associated to the model M. T_M can be interpreted as the union of terminating call sequences associated to the set of correct implementations in which preconditions are not widened[2]. We now present a conformance relationship, based on admissible traces, that aims at establishing whether an application conforms to a security model.

3.3 Mapping Security and Functional Levels

In smart card applications, the granularity between operations which are the target of the access control policies and the operations of the implementation

[2] Such precondition can be automatically built, as described in [1] p. 524.

is generally the same one. Nevertheless, for security reasons, implementation operations are defensive and can be invoked in any case whereas the execution of operations of the security model does not make sense if access control conditions do not hold. A mapping is a set of rules stating how application calls can be related to controlled operations of the security kernel. In a more general case, a rule takes one of the two following forms:

$$1.\ (op_{app}, v_{app}, r_{app}) \rightarrow (op_{sec}, v_{sec}, < r_{sec},\ \text{OK} >)$$
$$2.\ (op_{app}, v_{app}, r_{app}) \rightarrow (\mathsf{skip}, < \text{KO} >)$$

The first case maps an implementation behavior with an authorized security behavior. The second case corresponds to non-authorized calls, in which security attributes must not be modified, in any way. OK and KO are values which are introduced in the security kernel (Sect. 2.1). $v_{app}, r_{app}, v_{sec}, r_{sec}$ denotes sequences of expressions with possibly free variables. We denote by $free(t)$ the set of free variables in term t. A mapping can be non-deterministic, meaning that some form of implementation behavior can be matched with different monitored behaviors. Expected properties of mapping are discussed in Sect. 4.2. Let be here an example of a rule relating some behaviours of the `checkpin` implementation (see Fig. 2) with some behaviours of the `checkpin` definition at the security level (see Fig. 1).

$$\begin{array}{l}(checkpin, < apdu >, < SW\_NO\_ERROR >) \\ \quad \mapsto\ (checkpin, < OFFSET\_CDATA..OFFSET\_CDATA + lc - 1 \lhd apdu >, \\ \qquad\quad < success, OK >)\end{array}$$

in which $OFFSET\_CDATA..OFFSET\_CDATA + lc - 1 \lhd apdu$ represents the extraction of the parameter pin value located at index $OFFSET\_DATA$ of the *apdu* buffer. Definition 3 hereafter states how mapping rules can be applied.

Definition 3 (Closure of mapping by instanciation). *Let $l \mapsto r$ be a rule of a mapping R and let σ be any substitution with $dom(\sigma) \subseteq free(l) \cup free(r)$. R is close by σ meaning that $\sigma(l) \mapsto \sigma(r) \in R$.*

Then a term t can be rewritten into s by a mapping R if and only if $t \mapsto s \in R$.

3.4 Conformance Definition

Intuitively, an application conforms to a security model if and only if its traces are accepted by the security model, through the mapping relation. Due to the considered security policies, it means that: (*i*) all sequences of positive calls (associated to an effective execution of operations) can also be played by the security model, and, (*ii*) the application level can refuse more executions than the security level, in particular for functional reasons.

More formally, let $t_{app} = < init_{app}\ ;\ call^1_{app}\ ; \ldots\ ; call^n_{app} >$ be a trace relative to the application and let $t_{sec} = < init_{sec}\ ;\ call^1_{sec}\ ; \ldots\ ; call^n_{sec} >$ be a trace

relative to the security model, with $call^i$ a 3-uplet of the form (op^i, v^i, r^i). A mapping relation R can be extended to traces in the following way:

$$(t_{app} \mapsto t_{sec}) \in R \text{ iff } (call^i_{app} \mapsto call^i_{sec}) \in R \text{ for } i \in 1..n$$

Then, for a given mapping R, the finite set of traces associated to a trace t_{app} of the implementation level can be computed as $\{t_{sec} \mid (t_{app} \mapsto t_{sec}) \in R\}$. Now, operation calls that return KO can be assimilated to stuttering steps [15], because they do not modify security attributes. The operation $Stut$, defined hereafter, erases such calls.

Definition 4 (Elimination of refused executions)

$$\begin{aligned} Stut(< (\mathsf{skip}, < \text{KO} >) \ ; \ s >) &\hat{=} Stut(< s >) \\ Stut(< (op, v, < r, \text{OK} >) \ ; \ s >) &\hat{=} < (op, v, < r, \text{OK} >) \ ; \ Stut(< s >) > \\ Stut(<>) &\hat{=} <> \end{aligned}$$

Finally, the conformance between an application A and a security model S, through a mapping relation R, can be defined.

Definition 5 (Conformance relationship)

$$\forall \ ta \ (ta \in T_A \Rightarrow \exists \ ts \ ((ta \mapsto ts) \in R \wedge Stut(ts) \in T_S))$$

With this definition, it is possible to implement some part of the access control in a wrong way, for instance in making a mistake during the update of a security attribute. The conformance relationship that we propose (only) verifies that a wrong implementation can not be used to obtain rights that are not authorized. Nevertheless, it is the main expected characteristics in security: a security failure which can not be exploited in any way is not really a problem[3]. For instance, in the implementation of Sect 2.2, the result of the authentication (variable `IsHoldAuth` in the security model) is stored in the `OwnerPin` object and can be consulted through the `isValidated()` method [19]. Suppose now this variable is not updated by true, whereas the authentication succeeds. According to the access control rules given Sect. 2.1, the trace:

$$< init \ ; \ (checkpin, apdu, NO\_ERROR) \ ; \ (credit, val, NO\_ERROR) >$$

is not authorized to be played by the application. But it is not a security error. On the contrary, if this variable is not updated by false whereas the authentication fails (call to `hpc._reset()` in the implementation), the trace:

$$< init \ ; \ (checkpin, apdu, TERMINAL\_DENIED) \ ; \ (credit, val, NO\_ERROR) >$$

that intuitively corresponds to an abstract trace where a credit is authorized after an erroneous authentication, will be detected as insecure.

[3] The purpose of this work is not to found malicious errors that can be exploited by attackers. It is rather a mean to ensure that the security aspects, namely the access control mechanisms, have correctly been implemented.

4 Application to APDU Implementations

During the POSÉ project, the security model was designed as a direct abstraction of the functional model, the final goal being to complete functional test campaign with new tests dedicated to security requirements. The closeness between security and functional models was enforced by the strong requirement for our test generation tool (namely LTG [13]) to dispose of deterministic models. The tests generated using the functional model [14] are concretized to be run on the system under test through an *adaptation layer* that encapsulates operation calls and responses into APDU (the Gemalto tool, named EVA for Easy Validation Application). Mapping rules were very simple because they state direct correspondences of the internal behaviors between the functional and security models. In this section, we extend the previous approach by proposing a form of mapping dedicated to APDU level implementations. Furthermore, we propose proof obligations to validate mapping rules and we show how these proof obligations can be used to verify mappings, particularly when they are non deterministic.

4.1 A Form of Mapping Dedicated to APDU Implementations

In the JavaCard framework, an applet extends the `javacard.framework.Applet` class that defines the common methods an applet has to provide. Method `process` "decodes" the APDU command, executes the expected treatment and builds the corresponding APDU response. This method is invoked by the JCRE that manages the exchanges between terminals and cards. Here is the APDU format for invoking the `checkPin` procedure of our example:

CLA	INS	P1	P2	Lc	Data Field	Le
80h	52h	00h	00h	04h	4 bytes (pin value)	00h

Thus, the exchange between a card acceptance device and a smart card can be modelled by the `exchange` operation given hereafter:

```
Data_res, SW ← exchange(CLA, INS, P1, P2, Lc, Data_com, Le) ≙
  PRE Lc=size(Data_com) THEN
    VAR apdu IN
      apdu ← receiveAPDUcommand(CLA, INS, P1, P2, Lc, Data_com, Le) ;
      process(apdu) ;
      Data_res, SW ← sendAPDUresponse
    END
  END
```

Precondition of `exchange` states that the condition `Lc=size(Data_com)` must be verified when APDU commands are built, as pointed out in Sect. 2.2. Method `process` tests the fields `CLA` and `INS` and, depending on these values, invokes the appropriate method. For instance, method `checkPin` will be invoked as soon as `CLA=80h` and `INS=52h`. The mapping associated to the `exchange` operation is given in Tab. 1, with v=<80h, 52h, P1, P2, Lc, Data, Le>. `SW` results are

Table 1. Mapping for the `checkPin` operation

Functional cases	
(`exchange`, v, <NO_ERROR>)	→ (`checkPin`, <Data>, <success, OK>)
(`exchange`, v, <WRONG_DATA>)	→ (`checkPin`, <Data>, <failure, OK>)
(`exchange`, v, <PIN_BLOCKED>)	→ (`checkPin`, <Data>, <failure, OK>)
Errors coming from the security model	
(`exchange`, v, <TERMINAL_DENIED>)	→ (`skip`, <KO>)
(`exchange`, v, <MODE_INCORRECT>)	→ (`skip`, <KO>)
Functional errors	
(`exchange`, v, <COND._NOT_SATISFIED>	→ (`skip`, <KO>)
(`exchange`, v, <INCORRECT_P1P2>	→ (`skip`, <KO>)
(`exchange`, v, <WRONG_LENGTH>	→ (`skip`, <KO>)

those of Sect. 2.2 in which prefixes `ISO7816` and `SW` are omitted. In the case of `checkPin`, no other data response is expected. This value is thus omitted.

4.2 Mapping Properties

The proposed approach is based on the relevance of the mapping. This latter must be a simulation relationship linking behaviors describing similar treatments. We propose here some proof obligations relative to the expected properties for mappings. As considered in the POSÉ project, we suppose disposing of formal models both for the security and the implementation levels. Furthermore, due to the concerned applications, we consider here that there is no internal data refinement between these two models. Subjects, objects and security attributes (Sect. 2.1) are represented in the same way, possibly after renaming. These hypotheses are based on the fact that smart card applications generally do not imply sophisticated algorithms or data representations relatively to access controls. Furthermore, we consider here non-deterministic mappings. As stated before, non-determinism is a way to deal with multiple errors: depending on the order in which verifications are performed in the implementation and in the security model, the result may differ. This non-determinism is very important in the sense that implementations are free to favour one cause over another, avoiding a cause of side channel, making the implementation behavior too predictable. To take non-determinism into account, mappings will now be given as a set of rules of the form $l_i \rightarrow \{r_1^i, \ldots, r_n^i\}$ with l_i describing non-overlapping implementation behaviors: i.e. formulae $\neg(\mathsf{prd}(exec(l_i)) \wedge \mathsf{prd}(exec(l_j))$ for $i \neq j$, must be established under the hypothesis of the invariant of the application and with *exec* defined as in Def. 1.

Definition 6 (Mapping Correctness). *A mapping rule* $l \rightarrow \{r_1, \ldots, r_n\}$ *is correct if and only if:*

$$\text{CHOICE } exec(r_1) \text{ OR } \ldots \text{OR } exec(r_n) \text{ END } \sqsubseteq \quad exec(l)$$

Another interesting property is to establish that there exists a rule that applies for each implementation behavior.

Definition 7 (Mapping Completeness). *Let $o \leftarrow op(i) \mathrel{\hat{=}} S$ be the operation definition at the implementation level and let $(op, v^1_{app}, r^1_{app}), \ldots, (op, v^k_{app}, r^k_{app})$ be the set of left sides relative to op. A mapping R is complete if and only if:*

$$\begin{array}{l}\text{CHOICE SELECT } i = v^1_{app} \text{ THEN } exec(op, v^1_{app}, r^1_{app}) \text{ END} \\ \text{OR } \ldots \text{ OR SELECT } i = v^k_{app} \text{ THEN } exec(op, v^k_{app}, r^k_{app}) \text{ END} \\ \text{END } \sqsubseteq \quad exec(op, i, o)\end{array}$$

In this way, each behavior of the implementation is covered by a mapping rule.

Definitions 6 and 7 can be used to produce proof obligations relatively to a set of mapping rules. These proof obligations are obtained from the refinement definition (see section 3.1), and must be proved under the hypothesis of the invariant of the application. If the mapping is correct and complete, and if the initialization of the application refines the initialization of the security model, then each admissible traces at the application level is in conformance with the security model, according to Def. 5. Conditions of definitions 6 and 7 are sufficient ones but not necessary, as pointed out at the end of Sect. 3.4. That means that there exists some couples (t_{app}, t_{sec}) verifying the conformance relationship that do not correspond to a step by step refinement.

4.3 Example

A B model describing the APDU implementation level has been developed[4]. The JCRE machine contains the buffer APDU and operations `receiveAPDUcommand` and `sendAPDUresponse`. Similar models has been recently developed, for instance in JML [16]. The application model is just the translation of the Java description in B, using the API models. We illustrate here how condition 6 can be used to find error in mappings, particularly when non-determinism is allowed. Suppose now we want to specify, at the level of the security model, that the result of `checkPin` (Fig. 1) depends on the pin format. Then, the security definition of `checkPin` is now:

```
sr, status ← checkPin(p) ≙
  PRE p ∈ TAB_BYTE THEN
      IF  mode=USE ∧ subject=BANK  THEN
          IF size(p)=2 THEN ... /* remainder of the operation */
          ELSE sr := OK ‖ status := data_error END
      ELSE  sr := KO
      END
  END
```

An invocation that contains two security errors (an unsupported terminal and a pin size different from the expected one) prioritizes the `TERMINAL_DENIED` response in the security model (Fig. 1) and the `WRONG_LENGTH` status word in the

[4] `http://www-verimag.imag.fr/~potet/B08`

implementation (Fig. 2). As a consequence, rules relative to `TERMINAL_DENIED` and `WRONG_LENGTH` responses must be:

(exchange, v, <TERMINAL_DENIED>) → (skip, <KO>)
(exchange, v, <WRONG_LENGTH>) → (checkPin, <Data>, <data_error, OK>)
(exchange, v, <WRONG_LENGTH>) → (skip, <KO>)

Suppose now that the last rule has been omitted. Then the mapping correctness (Def. 6) can not be established. The proof obligation associated to the left part (exchange, v, <WRONG_LENGTH>) is:

$$\begin{array}{c} Lc = size(Data) \wedge\ apduBuf[\text{OFFSET\_P1}] = 00 \\ \wedge\ apduBuf[\text{OFFSET\_P2}] = 00\ \wedge\ Lc \neq 2 \\ \Rightarrow mode = use \wedge\ terminal = bank \wedge\ size(Data) \neq 2 \end{array}$$

with $Lc = size(Data)$ the precondition of the operation `exchange` and $mode = use \wedge\ terminal = bank \wedge size(Data) \neq 2$ the prd predicate associated to the right part (`checkPin`, <Data>, <data_error, OK>). This proof obligation can not be proved, showing that some rules are missing. Adding the last rule, the proof obligation becomes:

$$\begin{array}{c} Lc = size(Data) \wedge\ apduBuf[\text{OFFSET\_P1}] = 00 \\ \wedge\ apduBuf[\text{OFFSET\_P2}] = 00\ \wedge\ Lc \neq 2 \\ \Rightarrow (mode = use \wedge\ terminal = bank \wedge\ size(Data) \neq 2 \\ \vee\ \neg(mode = use \wedge\ terminal = bank) \end{array}$$

and allows to establish the correctness of the mapping. In the same way if the first rule of Table 1 is omitted, this error can be detected by the completeness definition: behaviors producing the status word `SW_NO_ERROR` at the application level should not be covered by any rule.

5 Conclusion and Future Work

In this paper, we have defined a formal framework that relates a security model with a Java Card implementation using APDU commands and responses. It relies on the definition of a mapping between these two abstraction levels. Using B as a support for describing this framework makes it possible to perform both correctness and completeness verifications on the mapping in order to ensure the accuracy of the results. This conformance relationship has been employed in the french RNTL POSÉ project, which aimed at validating security mechanism in a JavaCard application by means of dedicated security tests [14,8]. In practice, the conformance relationship acts as an oracle for the test execution, similar to the **ioco** relationship (input-output conformance) [20], extended by a mapping. In this project, correctness and completeness conditions have not been established by proof but were used as a guidance to verify mappings through a review process.

The B method has already been used as a support for access control policies [4,18]. In [4], the authors propose a form of modeling attached to Or-BAC

access control and characterize behaviors which conform to a given access control policy. Our approach can be seen as an extension of [4,18] taken into account the conformance of an application with respect to a security model. To do that questions of observation, refinement and correspondence between models stated at different levels of abstraction, have to be tackled. Although our conformance relationship is based on B refinement, we mainly focused on the computation behaviors of a system, through a mapping relationship, describing what it means to represent abstract behaviors by more concrete ones. This form of refinement is named interface refinement in [15]. Compared to B theory of refinement, the main particularity here is the way of characterizing the set of behaviors associated to a classical B model, based on operations. We do not consider here all sequences of terminating operation calls but traces associated to the observation of operation calls, where calls take place during execution. In this way, we obtain a set of traces that embraces terminating sequences of calls, and corresponding to traces admitted by the set of correct refinements associated to a classical B model and where preconditions could not be widened. The final condition guarantees that abstract invariants are always preserved at the level of implementations, as it is the case when only abstract terminating sequences are considered. From a methodological point of view, our approach can be seen as an instantiation of the Model Driven Security approach proposed in [3]. We revisited here the notions of dialect relating security concepts with a model of an application as done in the Meca tool, and classification of actions in atomic ones (directly mapped to actions of the implementation level) and composite ones. We have here exploited the B theory and particularities of our application domain to formally state the notion of security correctness, based on semantic preservation.

We are currently implementing this approach into a tool that aims to verify smart card applications w.r.t. different forms of security policies. This tool will be based on the API B models under development and our OpenJCard tool[5] allowing to execute APDU implementations. Extensions to other forms of security requirements and systems will be studied in the new ANR project LISE[6], in which traceability between security requirements and components must be formalized in order to point out liabilities when wrong executions are detected. The proposed approach seems to be easily expendable to take into account other types of security properties that can be evaluated on a current execution. On the contrary, security properties as non-interference [10] or side channels, that involve sets of executions, will be based on more sophisticated conformance relations.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Mussat, L.: Introducing Dynamic Constrains in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393. Springer, Heidelberg (1998)

[5] `http://www-lsr.imag.fr/Les.Personnes/Thierry.Moutet/`
[6] ANR-07-SESU-007.

3. Basin, D., Doser, J., Lodderstedt, T.: A temporal logic of actions. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(1) (2006)
4. Benaissa, N., Cansell, D., Mery, D.: Integration of Security Policy into System Modeling. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 232–247. Springer, Heidelberg (2006)
5. 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)
6. Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components. Technical Report CCMB-2006-09-002, v3.1 (September 2006)
7. Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components. Technical Report CCMB-2006-09-003, v3.1 (September 2006)
8. Dadeau, F., Potet, M.-L., Tissot, R.: A B Formal Framework for Security Developments in the Domain of Smart Card Applications. In: SEC 2008: 23th International Information Security Conference, IFIP proceedings. Springer, Heidelberg (to appear, 2008)
9. Dijkstra, E.W.: A discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
10. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
11. Haddad, A.: Meca: a Tool for Access Control Models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 281–284. Springer, Heidelberg (2006)
12. Smart Card Standard: Part 4: Interindustry Commands for Interchange. Technical report, ISO/IEC 7816-4 (1995)
13. Jaffuel, E., Legeard, B.: LEIRIOS Test Generator: Automated Test Generation from B Models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 277–280. Springer, Heidelberg (2006)
14. Julliand, J., Masson, P.-A., Tissot, R.: Generating security tests in addition to functional tests. In: AST 2008, 3rd Int. workshop on Automation of Software Test, Leipzig, Germany, May 2008, pp. 41–44. ACM Press, New York (2008)
15. Lamport, L.: A temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)
16. Poll, E., van den Berg, J., Jacobs, B.: Specification of the JavaCard API in JML. In: Domingo-Ferrer, J., Chan, D., Watson, A. (eds.) CARDIS. IFIP Conference Proceedings, vol. 180, pp. 135–154. Kluwer, Dordrecht (2000)
17. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
18. Stouls, N., Potet, M.-L.: Security Policy Enforcement through Refinement Process. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 216–231. Springer, Heidelberg (2006)
19. Java Card 2.1 Platform API Specification, `http://www.labri.fr/perso/bernet/javacard/spec/api/html/javacard/framework/`
20. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems 29(1), 49–79 (1996)

Modelling Attacker's Knowledge for Cascade Cryptographic Protocols

Nazim Benaïssa*

Université Henri Poincaré Nancy 1
benaissa@loria.fr
LORIA
BP 239
54506 Vandœuvre-lès-Nancy
France

Abstract. We address the proof-based development of cryptographic protocols satisfying security properties. Communication channels are supposed to be unsafe. Analysing cryptographic protocols requires precise modelling of the attacker's knowledge. In this paper we use the event B modelling language to model the knowledge of the attacker for a class of cryptographic protocols called *cascade protocols*. The attacker's behaviour conforms to the Dolev-Yao model. In the Dolev-Yao model, the attacker has full control of the communication channel, and the cryptographic primitives are supposed to be perfect.

Keywords: cryptography, model for attacker, formal methods.

1 Introduction

Proving properties such as secrecy or authentication on cryptographic protocols is a crucial point. A protocol satisfies a secrecy property if it is able to prevent the attacker from learning the content of a secret message intended for other users. By authentication we mean that an attacker can not mislead other honest agents about his identity. To be able to prove such properties on a protocol, we must be able to model the knowledge of the attacker. One popular model of attacker's behaviour is the Dolev-Yao model [6]; this model is an informal description of all possible behaviours of the attacker as described in section 2. In this paper we present an event B [1,2,4] model of the attacker for a class of cryptographic protocols called *cascade protocols* and we prove the secrecy property on it. Our work is based on that of Dolev-Yao [6] where they gave a characterization of secure *cascade protocols*, but proofs in their work were done by hand.

Proving properties on cryptographic protocols such as secrecy is known to be undecidable. However research involving formal methods for the analysis of security protocols has been carried out. Theorem provers or model checkers are

* This work was supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 251–264, 2008.

usually used for proving. For model checking, one famous example is Lowe's approach [7] using the process calculus CSP and the model checker FDR. Lowe discovered the famous bug in Needham-Schroeder's protocol. Model checking is efficient for discovering an attack if there is one, but it can not guarantee that a protocol is reliable. Many other works are based on theorem proving: Paulson [10] used an inductive approach to prove safety properties on protocols. He defined protocols as sets of traces and used the theorem prover Isabelle [9]. Other approaches, like Bolignano [3], combines theorem proving and model checking taking general formal method based techniques as a framework.

We summarize the organisation of the paper: in section 2, we present the Dolev-Yao attacker model. We then present the class of cascade protocols and the characterisation of secure protocols with respect to the secrecy property. The event B model of the attacker is given in section 3 of the paper.

2 The Dolev-Yao Model

In Dolev-Yao's model, cryptographic primitives are assumed to be black boxes satisfying given properties. The most important property is that the only way to obtain the plaintext M from the cipher text $K(M)$, where K is an encryption key, is to know the reverse key of K. In the Dolev Yao model, the attacker has full control of communication channels. He can intercept and remove any message from the channel, split unencrypted messages and decrypt parts of the message if he has the appropriate key. The attacker can also generate an infinite number of messages. All agents can be involved in an unlimited number of protocol instances, and interleaving of protocol instances have to be considered.

2.1 The Dolev-Yao Model for Cascade Protocols

Cascade protocols are a simple class of public protocols in which the agents involved in the protocol can apply several layers of encryption or decryption of messages. The encryption-decryption is made by using only public key operators. Dolev-Yao developed a model specifying the syntax of this class of protocols.

Let S be a set of symbols, we use S^* to denote the set of all finite sequences over S. Let E and D be respectively the set of encryptions and decryption keys of all the agents. If X is an agent, then his encryption key E_x and decryption key D_x are two functions mapping from $\{0,1\}^*$ into $\{0,1\}^*$. These functions satisfy the basic properties of the public key protocols: $E_x D_x = D_x E_x = id$, the identity function. D_x is known only by the agent himself while E_x is public and available in a key server.

Here is a short description of the Dolev-Yao model for cascade protocols (see [6] for detailed information). As shown in figure 1, a two party cascade protocol in the Dolev-Yao model is specified by a series of finite strings:

- $\alpha_i(X,Y) \in \{E_x, D_x, E_y\}^*, \quad 1 \leq i \leq t$
- $\beta_i(X,Y) \in \{E_y, D_y, E_x\}^*, \quad 1 \leq i \leq t' \; with \; t' = t \; or \; t-1$

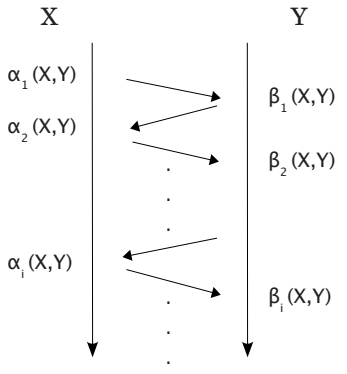


Fig. 1. A cascade protocol between two agents X,Y

When an agent X wishes to transmit a plaintext message M to another agent Y, the exchanged message has the following form: $N_k(X,Y)M$, where $1 \leq k \leq t + t'$ and:

- $N_1(X,Y) = \alpha_1(X,Y)$,
- $N_{2j}(X,Y) = \beta_j(X,Y)N_{2j-1}(X,Y), \quad 1 \leq j \leq t'$,
- $N_{2i+1}(X,Y) = \alpha_{i+1}(X,Y)N_{2i}(X,Y), \quad 1 \leq i \leq t-1$.

An attacker Z is supposed to be able to intercept any exchanged message between two agents X and Y, a cipher message $N_k(X,Y)M$ with $(k = 1, 2....)$, and will try to obtain the plaintext message M by applying different operators from one of three following categories:

1. $E \cup \{D_z\}$, E is known by all agents, and Dz is the attacker decryption key.
2. $\beta_i(X,Y)$ *for all* $X \neq Y$ *and* $i \geq 1$, even if the attacker does not know $\beta_i(X,Y)$'s value, he can start a transmission with any agent Y claiming himself to be agent X. He can then send any message *Msg* to Y in the $(2i-1)$st message and wait for Y's answer. He will then get $\beta_i(X,Y)$ applied to his message *Msg*.
3. $\alpha_i(X,Y)$ *for all* $X \neq Y$ *and* $i \geq 2$, in this case the attacker does not know the value of $\alpha_i(X,Y)$ but he may wait for X sending a message to Y, he can intercept Y's reply and prevent it from reaching X. He can then send any message to X claiming himself to be Y with his own message *Msg* and wait for the reply from X with $\alpha_i(A,B)$ applied to *Msg*.

As a result, the attacker will try to obtain the plaintext Message M from a cipher message $N_k(X,Y)M$ with $(k = 1, 2....)$ by applying operators from these three categories even if the he does not know the value of $\alpha_i(X,Y)$ or $\beta_i(X,Y)$ for two agents X and Y.

2.2 Secure Cascade Protocols in the Dolev-Yao Model

We give here two definitions from the Dolev-Yao model followed by the characterization of secure cascade protocol:

Definition 1. *Let $\pi \in (E \cup D)^*$ be a string and A be a user name. We say that π has the balancing property with respect to A if the following statement holds: if $D_A \in symb(\pi)$ then $E_A \in symb(\pi)$* [1]

Definition 2. *Let X,Y be two distinct user names. A two party cascade protocol is a balanced cascade protocol if*

1. *for every $i \geq 2$, $\alpha_i(X, Y)$ has the balancing property with respect to X, and*
2. *for every $j \geq 1$, $\beta_j(X, Y)$ has the balancing property with respect to Y.*

And the main result of the Dolev-Yao model is the following theorem.

Theorem 1. *Let X,Y be two distinct user names. A two-party cascade protocol is secure if and only if*

1. *$symb(\alpha_1(X, Y)) \cap \{E_x, E_y\} \neq \varnothing$, and*
2. *the protocol is balanced.*

After presenting the Dolev-Yao attacker model and the cascade protocols, we give in the next section an event B model of the attacker and prove on this model that if a cascade protocol is balanced then the secrecy property holds on this protocol.

3 Modelling the Attacker

First we give the static part of the model, the basic carrier sets are the following

- Msg: Set of all possible messages exchanged in the system.
- agent: Set of all agents including attackers.

We also define the set of encryption and decryption keys, respectively E and D. Two total injective functions EA, DA associate keys to their owners. Obviously, two different agents can not have the same encryption or decryption keys.

SETS

Msg
$agent$

AXIOMS

$axm1 : D \subseteq Msg \rightarrow Msg$
$axm2 : DA \in agent \rightarrowtail D$
$axm3 : E \subseteq Msg \rightarrow Msg$
$axm4 : EA \in agent \rightarrowtail E$

The attacker is an agent among others, he has his own encryption and decryption key:

$axm5 : Z \in agent$
$axm6 : Dz \in D$
$axm7 : DA(Z) = Dz$
$axm8 : Ez \in E$
$axm9 : EA(Z) = Ez$

[1] $symb(\pi)$ is the set of symbols of π.

Cryptographic primitive are supposed to be perfect and only the decryption key of an agent can be used to decrypt a message encrypted with his encryption key, this is modeled by the use of sequences and the reduction operation over the sequences.

3.1 Key Sequences

In cascade protocols, agents may apply more than one key on a message they received. A possible modelling of encryptions where several keys are applied is the use of function composition. If X and Y are two agents, $(DA(X); EA(Y))(Msg)$ is an encryption with two keys $DA(X)$ and $EA(Y)$. The problem with using function composition is that it has no memory, and it is therefore not possible to write properties on the structure of an encryption with more than one key. Thus, we use sequences to model an encryption where several layers of keys are used. For example, if X and Y are two agents, $[EA(X), DA(Y), EA(X)]$ is an encryption sequence where $EA(X)$ is first applied, and is followed by $DA(Y)$ and $EA(X)$.

When an encryption key of an agent is immediately followed by a decryption key of the same agent in a sequence, this sequence can be reduced to a shorter sequence where both keys are removed. For example, if X and Y are two agents, $[EA(X), DA(X), EA(Y)]$ can be reduced to $[EA(Y)]$. Formally, we model the reduction relation as the smallest relation that satisfies:

$$
\begin{array}{l}
axm10 : reduction \in (\mathbb{N} \nrightarrow D \cup E) \leftrightarrow (\mathbb{N} \nrightarrow D \cup E) \\
axm11 : \forall seq1, seq2, i, j, k, A \cdot \\
\quad A \in agent \wedge \\
\quad i \mathinner{..} j \subseteq \mathbb{N} \wedge k \in i \mathinner{..} j \wedge k + 1 \in i \mathinner{..} j \wedge \\
\quad seq1 \in i \mathinner{..} j \rightarrow D \cup E \wedge \\
\quad seq1(k) = DA(A) \wedge seq1(k+1) = EA(A) \wedge \\
\quad seq2 \in i \mathinner{..} j - 2 \rightarrow D \cup E \wedge \\
\quad seq2 = i \mathinner{..} j - 2 \lhd (seq1 \Leftarrow \{l \mapsto m | l \in k \mathinner{..} j - 2 \wedge m = seq1(l+2)\}) \\
\quad \Rightarrow \\
\quad seq1 \mapsto seq2 \in reduction
\end{array}
$$

In the previous axiom, we considered the case of a decryption key followed by an encryption key. We added a similar axiom for the case where an encryption key is followed by a decryption key.

To guaranty that reductions are made only between the encryption and decryption key of the same agent, the injectivity of the functions DA and EA is not sufficient and it is necessary to be sure that an encryption key of an agent is not used as a decryption key of another agent.

$$axm12 : ran(EA) \cap ran(DA) = \varnothing$$

We emphasize that since we use the *reduction* relation, it is not necessary to have the following property on agents keys:

$$axm13 : \forall A \cdot A \in agent \Rightarrow (DA(A); EA(A)) = id(Msg)$$

It may be possible to apply several reductions iteratively over a sequence. Thus, the *reduction* relation needs to be applied iteratively. We use a relation *Rep* similar to the one used by Cansell and Méry in [5]. *Rep* behaves like a repeat-until loop, it captures the idea of repeating a relation on a set as long as it is possible to apply the relation. A pair $(seq1, seq2)$ is in *Rep* if either $seq1 \notin dom(reduction)$ and $seq1 = seq2$ or $seq1 \in dom(reduction)$ and there is a path over *reduction* leading to $seq2 \notin dom(reduction)$. Formally, *Rep* is the smallest relation that satisfies:

$$axm14 : NotDOMAIN = id(\mathbb{N} \nrightarrow D \cup E) \setminus id(dom(reduction))$$
$$axm15 : Rep \in (\mathbb{N} \nrightarrow D \cup E) \leftrightarrow (\mathbb{N} \nrightarrow D \cup E)$$
$$axm16 : Rep = NotDOMAIN \cup (reduction; Rep)$$

When no more reductions are possible, we say that the sequence is in the *normal form*. Formally, the normal form is modeled as follows:

$$axm17 : Norm \in ((\mathbb{N} \nrightarrow D \cup E) \rightarrow (\mathbb{N} \nrightarrow D \cup E))$$
$$axm18 : Norm \subseteq Rep$$

If the normal form of a sequence *seq* equals the empty set, it means that the composition of all encryption and decryption keys contained in the sequence equals the identity function and we can obtain the plain text M from $seqM$.

seq_Ai and seq_Bj are two sets containing sequences of encryption or decryption keys. If X and Y are two agents involved in a protocol run, seq_Ai contains all sequences of keys applied in each step of the protocol by agent X, seq_Bj contains those applied by Y. Each sequence contained in one of these sets matches with an $\alpha_i(X, Y)$ or $\beta_j(X, Y)$ defined in the Dolev-Yao model.

$$axm19 : seq\_Ai \subseteq \mathbb{N} \nrightarrow D \cup E$$
$$axm20 : seq\_Bj \subseteq \mathbb{N} \nrightarrow D \cup E$$
$$axm21 : \forall seq \cdot seq \in (seq\_Ai \cup seq\_Bj) \Rightarrow (\exists X, Y \cdot X \in agent \wedge Y \in agent \wedge X \neq Y \wedge ran(seq) \subseteq \{DA(X), EA(X), EA(Y)\})$$

The protocol has to be *balanced* (see definition 2), thus for each sequence from the sets seq_Ai and seq_Bj the following axioms holds:

$$axm22 : \forall X, seq \cdot X \in agent \land seq \in seq\_Ai \land DA(X) \in ran(seq) \Rightarrow EA(X) \in ran(seq)$$
$$axm23 : \forall Y, seq \cdot Y \in agent \land seq \in seq\_Bj \land DA(Y) \in ran(seq) \Rightarrow EA(Y) \in ran(seq)$$

We emphasize the particular case of the first step of the protocol that is not concerned by the previous axiom22. We define a set seq_A1 containing the sequences corresponding to the first step of the protocol. It is not mandatory for sequences from this set to satisfy the *balancing property*, but they should at least contain one encryption key as stated in the Dolev-Yao characterization of secure protocols (see theorem 1):

$$axm24 : seq\_A1 \subseteq \mathbb{N} \nrightarrow D \cup E$$
$$axm25 : \forall seq \cdot seq \in seq\_A1 \Rightarrow (\exists X, Y \cdot X \in agent \land Y \in agent \land X \neq Y \land ran(seq) \subseteq \{DA(X), EA(X), EA(Y)\} \land ran(seq) \cap \{EA(X), EA(Y)\} \neq \varnothing)$$

3.2 Variables

We use a variable seq_Atk to model the structure of the messages that the attacker can obtain through applying his own keys or applying different sequences from the sets seq_Ai and seq_Bj. We also use a variable $size$ containing the size of the sequence seq_Atk and a variable $a1$ that memorizes the size of the sequence from the set seq_A1 used in the first step of the current protocol instance.

VARIABLES

seq_Atk
$size$
$a1$

INVARIANTS

$inv1 : size \in \mathbb{N}_1$
$inv2 : a1 \in \mathbb{N}_1$
$inv3 : seq\_Atk \in 1 \mathinner{..} size \rightarrow D \cup E$

We emphasize that the variable seq_Atk does not contain the plain text message M, but only the sequence of public key operators that may be applied by the attacker. Thus, in order to prove that the protocol satisfies the secrecy property, we must prove that the normal form of the sequence seq_Atk is never equal to the empty set. If the normal form of a sequence equals the empty set, it means that the composition of all encryption and decryption keys contained in the sequence equals the identity function and the attacker can obtain the plaintext M.

$$thm2 : Norm(seq\_Atk) \neq \varnothing$$

3.3 Events

The attacker can intercept any message exchanged between two agents. When a honest agent initiates a transaction with another agent, he first applies to the plain text message M a sequence from the set seq_A1 (first step of the protocol). The two agents apply then alternately sequences from seq_Ai and seq_Bj. Messages exchanged between agents have the form "$(seq\_Ai \cup seq\_Bj)^*$ seq_A1 M", M is the plaintext message. After intercepting the cipher message "$(seq\_Ai \cup seq\_Bj)^*$ seq_A1 M", the attacker applies different sequences from the set $seq\_Ai \cup seq\_Bj \cup E \cup \{D_z\}$. Accordingly, there is no need to model explicit message interception by the attacker, it is enough to initialize the variable seq_Atk with a sequence from the set seq_A1 and add events that model the concatenation of seq_Atk with all possible sequences:

- Initialization of seq_Atk with a sequence from seq_A1.
- Event $Attack\_seq\_Ai$: concatenation of seq_Atk with a sequence from seq_Ai.
- Event $Attack\_seq\_Bj$: concatenation of seq_Atk with a sequence from seq_Bj.
- Event $Attack\_E$: concatenation of seq_Atk with a sequence from E.
- Event $Attack\_D_z$: concatenation of seq_Atk with D_z.

These concatenations are done by some honest agent before the message is intercepted or by the attacker himself after intercepting the cipher message.

In order to write the appropriate events, we need to have tools that let us manipulate sequences such as concatenation or subsequences. In our model we use a modified form of the relation *match* introduced by Jean Raymond Abrial in the Earley algorithm model. We modified this relation to adapt it to our case study:

$$axm26 : match \in (\mathbb{N} \nrightarrow D \cup E) \leftrightarrow (\mathbb{N} \nrightarrow D \cup E)$$
$$axm27 : \varnothing \mapsto \varnothing \in match$$

Unlike the equality, two sequences $seq1 \in i..j \rightarrow D \cup E$ and $seq2 : k..l \rightarrow D \cup E$ may match if the order of the keys in the two sequences is the same even if their respective domains $i..j$ and $k..l$ are different (see example in figure 2).

$$\begin{aligned} axm28 : \forall & i, j, k, l, n1, n2, s1, s2 \cdot \\ & i \in 1 \,..\, j+1 \land \\ & j \in 0 \,..\, n1-1 \land \\ & k \in 1 \,..\, l+1 \land \\ & l \in 0 \,..\, n2-1 \land \\ & s1 \in 1 \,..\, n1 \rightarrow D \cup E \land \\ & s2 \in 1 \,..\, n2 \rightarrow D \cup E \land \\ & i \,..\, j \lhd s1 \mapsto k \,..\, l \lhd s2 \in match \land \\ & s1(j+1) = s2(l+1) \\ & \Rightarrow \\ & i \,..\, j+1 \lhd s1 \mapsto k \,..\, l+1 \lhd s2 \in match \end{aligned}$$

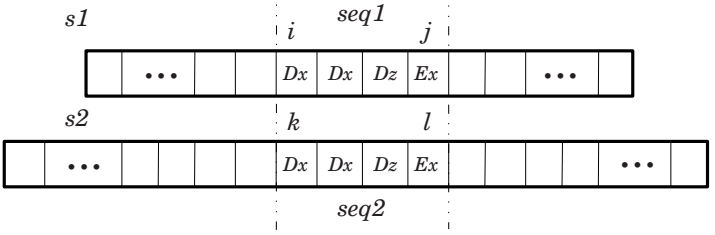


Fig. 2. The match relation

We also add a fixed point axiom saying that *match* is the smallest relation satisfying the axiom 28. Using *match* is convenient to express relations between sequences. For instance, to express the fact that a sequence *seq1* is a subsequence of *seq2*, it suffices to say that there are some i, j such that $seq1 \mapsto i\,..\,j \lhd seq2 \in match$. To express the fact that a sequence $seq \in i..j \rightarrow D \cup E$ is the result of the concatenation of two sequences *seq1* and *seq2*, it suffices to say that there is some k such that $seq1 \mapsto i\,..\,k \lhd seq \in match$ and $seq2 \mapsto k+1\,..\,j \lhd seq \in match$.

Events have been added to the model to express all the attacker's options. The following event shows the case of a sequence randomly chosen from the set *seq_Ai*. This sequence is concatenated with the attacker sequence *seq_Atk*, the variable size is also increased.

EVENT sendAi
ANY
seq_Ax
ax
WHERE
$grd1 : seq\_Ax \in seq\_Ai$
$grd2 : ax \in \mathbb{N}_1$
$grd3 : seq\_Ax \in 1\,..\,ax \rightarrow D \cup E$
THEN
$act1 : size := size + ax$
$act2 : seq\_Atk : | \; seq\_Atk' \in 1\,..\,size + ax \rightarrow D \cup E \wedge$
$seq\_Ax \mapsto 1\,..\,ax \lhd seq\_Atk' \in match \wedge$
$seq\_Atk \mapsto ax + 1\,..\,ax + size \lhd seq\_Atk' \in match$
END

Similar events are added to express all the other possibilities of the Dolev-Yao model. Since the attacker sequence is initialized with a sequence from the set seq_A1, it will have two parts (as shown in figure 3). A part $1..size-a1 \lhd seq\_Atk$ that matches with a sequence from $(seq\_Ai \cup seq\_Bj \cup E \cup \{D_z\})^*$, and a part $(size-a1)+1\,..\,size \lhd seq\_Atk$ that matches with a sequence from seq_A1. It's important to distinguish these two parts since, unlike sequences from the set $seq\_Ai \cup seq\_Bj$, sequences from the set seq_A1 do not satisfy the *balancing property.*

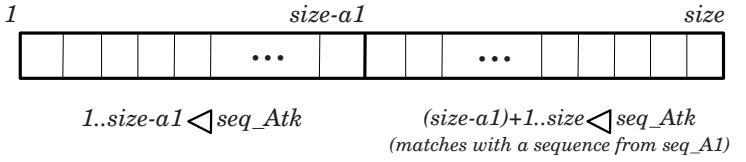


Fig. 3. The two parts of the attacker sequence

3.4 Invariant and Proofs

Proofs of the B model are inspired from the proofs given by Dolev and Yao in their model, but proofs of their models were done by hand and parts of their proofs were stated without being formally proved. Before introducing the main invariant of our model we first give definitions of some important properties over sequences that are necessary to state the invariant. $A\_Norm(A)(seq)$ is the normal form with respect to one agent A of a sequence seq, it is obtained by removing all possible subsequences $[EA(A), DA(A)]$ or $[DA(A), EA(A)]$.

$$axm29 : A\_Norm \in agent \rightarrow ((\mathbb{N} \nrightarrow D \cup E) \rightarrow (\mathbb{N} \nrightarrow D \cup E))$$

For example,

$$A\_Norm(X)([DA(Y), EA(Y), EA(X), DA(X)]) = [DA(Y), EA(Y)]$$

We modeled A_Norm similarly to $Norm$ using a reduction relation where only keys from the appropriate agent are reduced.

In order to prove that the normal form of the attacker sequence never equals the empty set, we need to prove first that the sequence $Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)$ has the *balancing property* with respect to all agents except the attacker himself. We recall that it is not mandatory to have the *balancing property* for sequences from the set seq_A1, this is why this property does not hold for the whole attacker sequence.

$$\begin{array}{l} thm3 : \forall A \cdot A \in agent \wedge A \neq Z \wedge \\ \quad DA(A) \in ran(Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)) \\ \quad \Rightarrow \\ \quad EA(A) \in ran(Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)) \end{array}$$

But unfortunately this property is not an inductive invariant but only a theorem. As a counter example, let us consider the case where $Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)$ equals:

$$[DA(A), DA(Z), EA(A), EA(Z), DA(X), EA(Y), DA(A), DA(Y), EA(X)]$$

This sequence satisfies the balancing property. If the previous event *sendAi* is triggered with the local variable $seq\_Ax = [DA(A), EA(Z), EA(A)]$ (this sequence satisfies the axioms 19 and 21), the new value of $Norm(1 \,..\, size - a1 \lhd seq\_Atk)$ will be: $[EA(Z), DA(X), EA(Y), DA(A), DA(Y), EA(X)]$. The new value does not satisfy the balancing property anymore. Thus we introduce a new property called *A_Balanced property* of a sequence with respect to an agent A:

$$
\begin{array}{l}
axm30 : A\_Balanced \in agent \rightarrow \mathbb{P}(\mathbb{N} \nrightarrow D \cup E) \\
axm31 : \forall A, seq, i, j \cdot seq \in i \,..\, j \rightarrow D \cup E \wedge \\
\qquad i \,..\, j \subseteq \mathbb{N} \wedge j \geq i \wedge \\
\qquad (seq(i) \in D \setminus \{DA(A)\} \wedge \\
\qquad seq(j) \in D \setminus \{DA(A)\} \wedge \\
\qquad ran((A\_Norm(A))(i+1 \,..\, j-1 \lhd seq)) \cap D \subseteq \{DA(A)\} \wedge \\
\qquad DA(A) \in ran((A\_Norm(A))(i+1 \,..\, j-1 \lhd seq)) \Rightarrow \\
\qquad\qquad EA(A) \in ran((A\_Norm(A))(i+1 \,..\, j-1 \lhd seq))) \\
\qquad \Rightarrow \\
\qquad seq \in A\_Balanced(A)
\end{array}
$$

Intuitively, for an agent A, a sequence is $A\_Balanced(A)$ means that if the first and last symbols of this sequence are decryption keys and if the $A\_Norm(A)$ of this sequence contains only A decryption key in its range it should also contain A encryption key.

The main invariant of our model states that each subsequence of the sequence $D_z \ (1 \,..\, size - a1 \lhd seq\_Atk) \ D_z$ has the *A_Balanced property* with respect to all agents except the attacker.

$$
\begin{array}{l}
inv4 : \forall seq, i, j, k, l, A, seq\_Atk\_Dz \cdot \\
\qquad A \in agent \wedge A \neq Z \wedge \\
\qquad seq \in i \,..\, j \rightarrow D \cup E \wedge \\
\qquad seq\_Atk\_Dz \in 1 \,..\, size - a1 + 2 \rightarrow D \cup E \\
\qquad seq\_Atk\_Dz(1) = Dz \\
\qquad seq\_Atk\_Dz(size - a1 + 2) = Dz \\
\qquad 1 \ldots size - a1 \lhd seq\_Atk \mapsto 2 \,..\, size - a1 + 1 \lhd seq\_Atk\_Dz \in match \\
\qquad seq \mapsto k \,..\, l \lhd seq\_Atk\_Dz \in match \\
\qquad \Rightarrow \\
\qquad seq \in A\_Balanced(A)
\end{array}
$$

Let us consider the case of the event *sendAi* shown before. In this event, we concatenate a sequence from the set seq_Ai to the sequence seq_Atk to obtain seq_Atk' . We have then to prove that any subsequence seq of $D_z \ (1 \,..\, size - a1 \lhd seq\_Atk) \ D_z$ (there are some k, l such that $seq \mapsto k \,..\, l \lhd (D_z \ (1 \,..\, size - a1 \lhd seq\_Atk') \ D_z) \in match$) is *A_Balanced* (see figure 4).

To achieve the proof, all possible cases of $k..l$ values have to be considered (especially the values of k and l compared to the value of $ax + 1$), this is made easier by the use of the *match* relation. For each case it is necessary to prove that

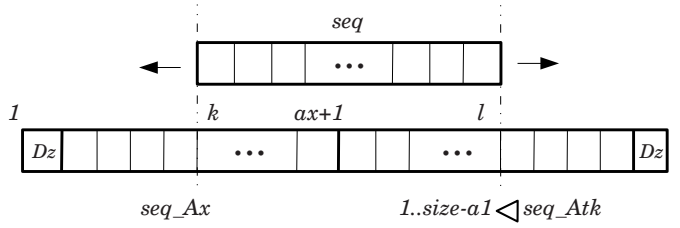


Fig. 4. A_Balanced property has to be proved on seq

the concatenation of a sequence that has the *balancing property* with respect to an agent A with a sequence that has the *A_Balanced property* with respect to A results on a sequence that has the *A_Balanced property* with respect to A, since this has to be done with all events of the model, it was interesting to prove the following theorem:

$$
\begin{array}{l}
thm4 : \forall seq, i, j, k, n, A \cdot \\
\quad A \in agent \wedge seq \in 1 \mathinner{..} n \rightarrow D \cup E \wedge \\
\quad i \mathinner{..} j \subseteq 1 \mathinner{..} n \wedge k \in i \mathinner{..} j \wedge \\
\quad A\_Norm(A)(i \mathinner{..} k \lhd seq) = i \mathinner{..} k \lhd seq \wedge \\
\quad (DA(A) \in ran(i \mathinner{..} k \lhd seq) \Rightarrow EA(A) \in ran(i \mathinner{..} k \lhd seq)) \wedge \\
\quad (DA(A) \in ran(A\_Norm(A)(k+1 \mathinner{..} j \lhd seq)) \Rightarrow \\
\qquad\qquad EA(A) \in ran(A\_Norm(A)(k+1 \mathinner{..} j \lhd seq))) \\
\quad \Rightarrow \\
\quad (DA(A) \in ran(A\_Norm(A)(i \mathinner{..} j \lhd seq)) \Rightarrow \\
\qquad\qquad EA(A) \in ran(A\_Norm(A)(i \mathinner{..} j \lhd seq)))
\end{array}
$$

The last step of our modelling is to prove theorem 2, that states that *seq_Atk* never equals the empty set, from the theorem 3 that states that the sequence $Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)$ has the *balancing property* with respect to all agents other than the attacker. To prove this result, we do a proof by case on the structure of $seq = size - a1 + 1 \mathinner{..} size \lhd seq\_Atk$ (the part of the attacker sequence that matches with the first step of the protocol). According to axiom 23, there are two agents X,Y such that $ran(seq) \subseteq \{DA(X), EA(X), EA(Y)\}$ and $ran(seq) \cap \{EA(X), EA(Y)\} \neq \varnothing$. We give here a sketch of the proof:
By contradiction, suppose that the normal form of *seq_Atk* equals the empty set, two case are possible:

1. $EA(Y) \in ran(seq)$: since $DA(Y) \notin ran(seq)$, the only way to obtain the empty set in the normal form of the whole sequence *seq_Atk* is that the reminder part $Norm(1 \mathinner{..} size - a1 \lhd seq\_Atk)$ contains $DA(Y)$ but not $EA(Y)$. This is impossible because of the *balancing property* of this part of the attacker sequence.
2. The other case is done in a similar way.

Proving these invariants and theorems requires intensive use of operators over sequences. The axiom defining the relation *match* given before is not convenient

in our case, that's why we introduced several theorems over this relation such as identity, reflexivity and transitivity properties to make proofs easier. Here follows an example of one of these theorems:

$$
\begin{array}{l}
thm5 : \forall seq1, seq2, k, i1, i2, j1, j2 \cdot \\
\quad seq1 \in i1 \mathinner{..} j1 \rightarrow D \cup E \wedge \\
\quad seq2 \in i2 \mathinner{..} j2 \rightarrow D \cup E \wedge \\
\quad k \in 0 \mathinner{..} j1 - i1 \wedge \\
\quad seq1 \neq \varnothing \wedge seq2 \neq \varnothing \wedge \\
\quad seq1 \mapsto seq2 \in match \\
\quad \Rightarrow \\
\quad seq1(i1 + k) = seq2(i2 + k)
\end{array}
$$

To prove this theorem we used induction over the size of the sequence. These theorems are not specific to this model, thus they can be reused later in similar protocol models. We used the Rodin platform [8] for modelling and proving our attacker model. 10 theorems were proved interactively on the *match* relation. 25 proofs generated by the prover for the invariants of the model, 13 were done automatically. Interactive proofs were not difficult except proofs of the main invariant 4 of the model that were long because of the high number of the cases that had to be considered.

4 Conclusion

We have written in this paper a B model of the attacker for a class of cryptographic protocols. Events of our model take into account all the options the attacker can perform in the Dolev-Yao model. Unlike the original Dolev-Yao's model for cascade protocols, proofs were mechanized. Accordingly, all constraints on the attacker's model have to be stated explicitly, and some of the constraints were added later during the proving process. Proofs of our model were made easier by the use of the *match* relation and by the theorems we have proved over this relation. These theorems can be reused in future developments. The next step will be modelling attackers for more complex classes of protocols and to study how attacker models can be integrated into the complete protocol model.

Acknowledgements. Thanks are due to Jean Raymond Abrial for his advice on modelling cryptographic protocols. We also thank Dominique Cansell and Dominique Méry for their help and suggestions.

References

1. Abrial, J.-R.: The B book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Bjørner, D., Henson, M.C.: Logics of Specification Languages. EATCS Textbook in Computer Science. Springer, Heidelberg (2007)

3. Bolignano, D.: Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In: CAV, pp. 77–87 (1998)
4. Cansell, D., Méry, D.: The event-B Modelling Method: Concepts and Case Studies, pp. 33–140. Springer, Heidelberg (2007) See [2]
5. Cansell, D., Méry, D.: Incremental parametric development of greedy algorithms. Electr. Notes Theor. Comput. Sci. 185, 47–62 (2007)
6. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
7. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using fdr. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
8. Metayer, C., Abrial, J.-R., Voisin, L.: Event-B language. RODIN Project Deliverable D7 (May 2005)
9. Paulson, L.C. (ed.): Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). LNCS, vol. 828. Springer, Heidelberg (1994)
10. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6, 85–128 (1998)

Using EventB to Create a Virtual Machine Instruction Set Architecture

Stephen Wright

Department of Computer Science, University of Bristol, UK
stephen.wright@bris.ac.uk

Abstract. A Virtual Machine (VM) is a program running on a conventional microprocessor that emulates the binary instruction set, registers, and memory space of an idealized computing machine, a well-known example being the Java Virtual Machine (JVM). Despite there being many binary Instruction Set Architectures (ISA) in existence, all share a set of core properties which have been tailored to their particular applications. An abstract model may capture these generic properties and be subsequently refined to a particular machine, providing a reusable template for development of formally proven ISAs: this is a task to which the EventB [16,18] notation is well suited. This paper describes a project to use the RODIN tool-set [24] to perform such a process, ultimately producing the MIDAS (Microprocessor Instruction and Data Abstraction System) VM, capable of running binary executables compiled from high-level languages such as C [9]. The abstract model is incrementally refined to a model capable of automatic translation to C source code, and compilation for a hardware platform using a standard compiler. A second C compiler, targeted to the VM itself, allows C programs to be executed on it.

1 Introduction

The last 10 years have seen the development of Integrated Modular Avionics (IMA) systems by the aerospace industry. These are generic computing platforms containing built-in system services, allowing application software to be developed as separate modules [1]. This concept may be extended by the introduction of a VM middle-ware layer, offering various technical and industrial advantages. The need for software portability during hardware updates (typically due to hardware obsolescence) is anticipated by providing tools and methods that create platform-portable systems during original development. Control executables may be generated and tested in an accurate environment simulation prior to final target hardware becoming available, and using the same development tool sets. The predictable performance provided by the VM allows system designers to more accurately predict final computational requirements at an earlier stage of development. As with any VM, the significant disadvantages are runtime performance and potential unreliability of the VM itself: the use of Formal Methods is proposed to assist in mitigation of the second issue.

The paper is organized as follows. Section 2 reviews current work in related subjects. Section 3 summarizes the common properties of a practical VM. Section 4 describes the structure of the EventB model. Section 5 describes the most significant

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 265–279, 2008.

EventB constructs used in the model. Section 6 describes how the model may be reused for different architectures and ISAs. Section 7 describes the specific VM ISA that has been implemented via the process as a demonstration. Section 8 describes the extensions to the RODIN platform that were developed for auto-translation of the model to an executable program. Section 9 describes the additional tools that were developed outside the RODIN environment in order to allow execution of a binary image on the developed VM. Section 10 suggests improvements and extensions that could be made to the RODIN tool in order to facilitate future work of this type. Section 11 suggests future work implied by the project.

2 Related Work

This project draws on 2 threads of academic and industrial research: the use of virtual machines in real-time embedded systems and the use of Formal Methods to perform verification of software. A common use of VMs is the implementation of legacy microprocessor instruction sets in order to allow porting of compiled software to new hardware platforms. VMs have been used to implement instruction set emulators for support of legacy software in both non real-time desktop and mainframe microprocessors [26], and real-time avionics systems [14,15].

Formal Methods have previously been applied to model the specification of the Java Virtual Machine (JVM), in order to capture its exact meaning and verify its internal consistency [22]. Complete machine checking of a JVM specification and the greater task of machine-checking an implementation against that specification remains a challenge [17]. However, a machine-checked specification and implementation of parts of the JVM have been achieved using B [5].

3 Common Properties of Instruction Set Architectures

In spite of a vast number of general-purpose microprocessor ISAs in existence, all share a set of core properties [7]. Programs are stored within the machine as a contiguous array of binary values, each locatable via an integer address stored in a special register: the program counter (PC). The selection of the next instruction to be executed ("control flow") is achieved by simple incrementing of the PC, or its overwriting with a calculated value to perform a jump operation. Calculations are performed by taking one or two fixed-size input registers, and outputting a result to an output register. In the case of a destructive operation, this output register may be one of the input registers. Certain ISAs implement specialist instructions with more than two inputs [1], but are not targeted at general purpose applications. Calculations consist of bit-manipulations, typically implementing the basic arithmetic operations (i.e. addition, subtraction, multiplication and division) plus a number of application-oriented operations. Input data may be loaded via "immediate mode" addressing, in which a constant value is incorporated within the executed instruction, or transferred from a specified location in a read-writable data memory. The data memory consists of a contiguous array of binary values: large register values may be stored across two or more contiguous memory locations. Data memory locations are specified by one of several addressing modes,

which may be generally classified as “direct” or “indirect” [7]. During direct mode addressing the location to be accessed is specified as a fixed address value incorporated into the instruction (i.e. the location is itself an immediate mode constant). During indirect mode, the location is specified by a value in data memory accessed by direct mode. Variations on this general paradigm exist, such as PC-relative mode [8], which may be considered a form of direct addressing.

In order to execute binary images compiled from the C high level language using existing tools, particularly the GNU Compiler Collection (GCC), the VM must be further specified to implement multiple 32-bit general registers, 8-bit wide data memory and all instruction and data memory addressing from a single register [21].

4 Model Description

The EventB model addresses only those aspects of a microprocessor’s functionality needed to implement a VM ISA. Therefore, only a functional description of the instruction set and its associated aspects are included, such as those parts of the memory system visible to the instruction stream. No consideration of the mechanisms within a typical microprocessor needed to efficiently implement an ISA is necessary. For example the model does not address instruction pipelining, memory caching or speculative execution mechanisms [7]. The model derives a functioning VM by a total of thirty-one stages of refinement, which may be summarized into seven distinct layers: StateMch, ControlFlowMch, RegMch, MemMch, StkMch, CalcMch and MidasMch. Each layer confers particular properties on the model, described here individually.

4.1 Model Architecture

StateMch defines a machine with only two state variables: an instruction and a machine status. A state machine is derived that defines the four possible states that the machine status may take: LOADING (of the program to be executed), RUNNING (of the loaded program), HALTED (i.e. commanded shutdown of the machine) and FAILED (i.e. detection of an error in the program). State values are specified as dissimilar abstract constants: assigning of numerical values is postponed to the last refinement stages of the model. The model is initialized to LOADING and may continue in that state until loading is complete, when RUNNING is entered. From RUNNING, RUNNING is re-entered on successful execution of a valid instruction, HALTED is entered by the execution of a valid halt instruction, or FAILED entered by execution of an invalid instruction or unsuccessful execution of a valid instruction. Execution results in the modification of the instruction state variable by unspecified means. Once the HALTED or FAILED state is entered, the machine enters explicit deadlock by the definition of appropriate events. Thus a mechanism for the refining of execution exceptions and the guaranteed stopping of the machine in this event is defined. The StateMch state transitions are summarized in Figure 1.

Mutually exclusive sets defining the instruction groups are derived in the corresponding machine context definition.

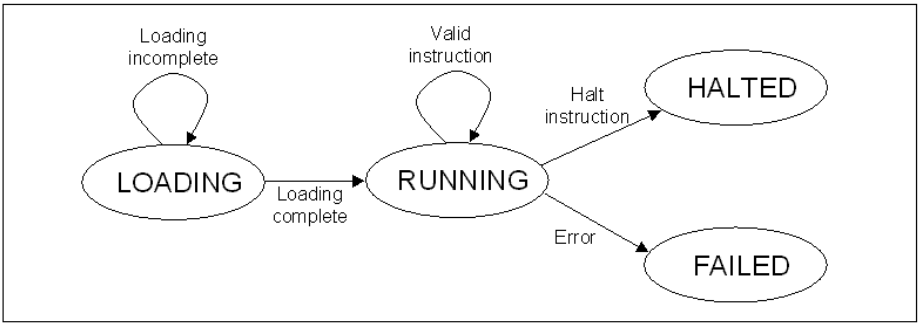


Fig. 1. StateMch State Transitions

ControlFlowMch refines the instruction variable to an array of Inst types and a variable representing the PC. A boolean variable representing an abstract conditional decision is introduced. Sets of instructions are constructed that define different effects on the PC: incrementing by an abstract constant N_2 (representing a normal instruction), incrementing by an abstract constant N_1 (representing a conditional jump on decision false), and overwriting from a stored jump vector (representing a conditional jump on decision true). Instructions are defined to allow setting of the jump vector and conditional flag. These data flows are summarized in Figure 2.

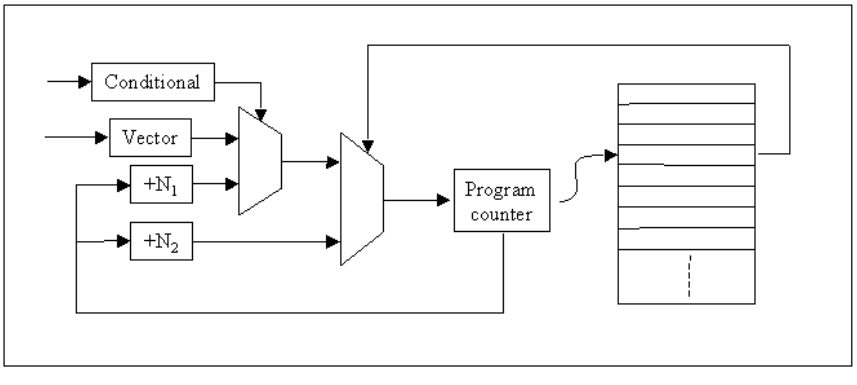


Fig. 2. ControlFlowMch Data Flows

Exception events are constructed identifying attempts to increment or overwrite the program counter outside the domain of the instruction array.

RegMch defines a set of states representing the two operands and one result used during calculations, and two stores of data, one read-only, and the other read-writable. Nine possible interactions are defined between these elements: loading of each of the two operands from both the read-only and read-writable store, a single input operation loading the result, a dual input operation loading the result, loading the program counter from the result (i.e. a jump operation), extraction of the current program

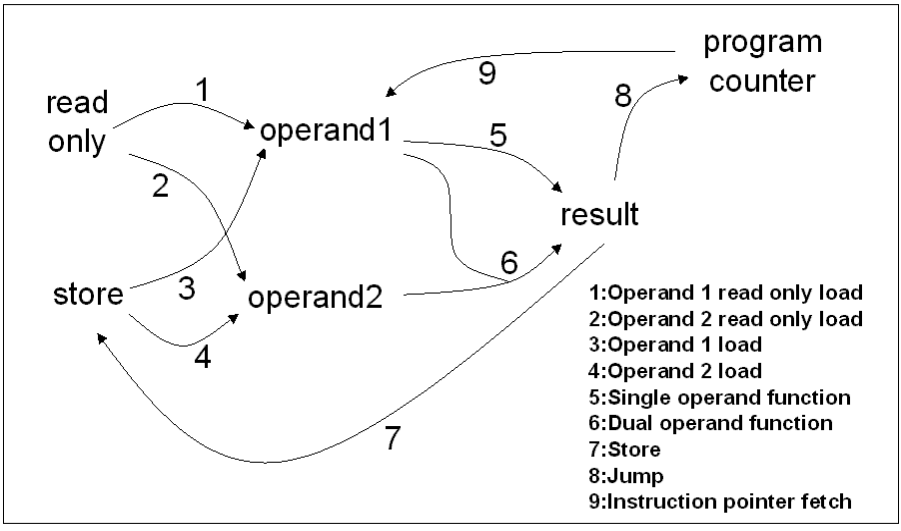


Fig. 3. RegMch Data Flows

counter back to the first operand, and loading of the read-writable store from the result. These interactions are summarized in Figure 3.

The operands and result have associated "write-possible" and "read-possible" flags, allowing defense against uninitialised reading to be modeled. Exception conditions are constructed identifying attempts to read or write a variable when the appropriate availability flag is false. The jump vector defined in the ControlFlowMch layer is refined to the result at this stage. The instruction array and program counter are referenced unmodified.

MemMch refines the read-only and read-writable stores into arrays of an abstract "Data" set, indexed via a pointer, which is then refined to separate direct and indirect pointers, for each addressing mode. The abstract Data set is further refined to Small, Medium and Large sub-sets, and the instruction sets are sub-divided to define distinct sub-sets to handle each addressing mode and size.

In StkMch the general paradigm constructed in the first four layers is specifically refined to a stack-based machine [7]. The operands and result are refined to locations on a data stack, and the write/read-possible flags are refined to checks on the size of that stack. For example, writing to the stack is not-possible if the stack is already full, refining the previously constructed event in which a write-possible flag is false. The EventB event merge capability is used to refine the separate events handling the two operand registers into single events.

CalcMch refines the two operand-to-result operations and conditional flag update constructed in the RegMch layer into a particular set of practical operations necessary for executing applications. The single operand operation is refined to integer-to-floating-point cast and floating-point-to-integer cast. The dual operand operation is refined to addition, subtraction, multiplication and division for both integer and floating point, integer modulo divide, bit-wise OR, AND, XOR, bitmap shift-left and shift-right. The conditional flag update is refined to greater-than, less-than, equal and not-equal comparison operations.

The final layer, MidasMch, merges the separate arrays defining the instruction and data regions into a single addressable array of small data elements. The instructions that have been constructed in the previous layers are mapped to concrete data values. Concrete numerical values are specified for the instruction and data region boundaries, thus defining the memory map of the machine. The event associated with the LOADING state is refined to a state machine capable of sequentially loading the instruction and read-only regions of the new contiguous memory space. Data-write events are refined to define a small region of writable IO within the data memory region.

4.2 Model Metrics

A complete list of the refinement stages within the seven general layers described previously is given in Table 1.

Table 1. The MIDAS refinement layers

Layer	Events	Proof Obligations (Automatically Discharged)	Proof Obligations (Manually Discharged)	Proof Obligations (Total)
StateMch	2	4	0	4
StateMchR1	5	2	0	2
StateMchR2	6	1	0	1
StateMchR3	7	5	0	5
ControlFlowMch	7	13	7	20
ControlFlowMchR1	9	9	0	9
ControlFlowMchR2	11	3	3	6
ControlFlowMchR3	20	13	56	69
RegMch	29	116	59	175
RegMchR1	33	30	9	39
RegMchR2	35	11	1	12
RegMchR3	39	24	1	25
RegMchR4	40	2	1	3
MemMch	40	2	36	38
MemMchR1	40	6	0	6
MemMchR2	51	140	6	146
MemMchR3	75	536	3	539
MemMchR4	75	248	50	298
StkMch	76	846	49	899
StkMchR1	92	111	4	115
StkMchR2	72	270	14	284
StkMchR3	56	67	54	121
CalcMch	61	0	34	34
CalcMchR1	87	0	116	116
CalcMchR2	89	0	16	16
MidasMch	95	0	16	16
MidasMchR1	111	2	772	774
MidasMchR2	111	0	147	147
MidasMchR3	112	0	8	8
MidasMchR4	114	0	10	10
MidasMchR5	122	419	1104	1523

Proof Obligations (PO) for all refinement stages are discharged using all of the RODIN proving tools. The increasing necessity for manual interaction during later refinement stages reflects the large number of hypotheses visible to the RODIN automatic proving tools at this point, thus requiring selection by the developer in order to achieve proof. Also of note is the reduction in the number of events at the StkMch layer. This is a result of the merging of the separate events that define loading of the two operand registers into a single stack push event, a simplification possible for a stack machine architecture.

5 Model Detail

The VM model contains a large number of EventB statements: those most significant to the VM application are described here in more detail.

5.1 Instruction Set Construction

Instructions are constructed by the successive division of a top-level EventB SET *Inst,* which represents the complete instruction space of the VM (i.e. both valid and invalid instructions). Statements are introduced to explicitly state that the derived sub-sets are inclusive of the entire parent set, in order to ensure that the entire instruction space of the VM is decoded. For example, the following EventB fragment summarizes the decomposition of a previously constructed instruction group StoreInst (all instructions that write to the data memory) into direct and indirect addressing mode sub-groupings.

CONSTANTS
StoreDirInst // Direct mode within store instruction
StoreIndirInst // Indirect mode within store instruction
AXIOMS
StoreDirInst $\subseteq$ StoreInst // Subset of StoreInst
StoreDirInst $\neq \varnothing$ // Direct mode exists
StoreIndirInst $\subseteq$ StoreInst // Subset of StoreInst
StoreIndirInst $\neq \varnothing$ // Indirect mode exists
StoreDirInst $\cap$ StoreIndirInst $= \varnothing$ // Mutually exclusive
StoreDirInst $\cup$ StoreIndirInst = StoreInst // Complete coverage (1)

5.2 Data Modeling

Binary data may be assigned a variety of meanings by a C program at run-time: data may represent integer, floating point or bit-field elements and may have different bit lengths, which may be treated as multiple smaller data elements. Therefore, use of the basic EventB types is insufficient and modeling of these features is achieved within the model by the introduction of a new set *Data*. Data is decomposed into 3 sub-sets, representing 3 possible data sizes:

CONSTANTS
DataLarge // Largest data size
DataMed // Medium data size
DataSmall // Small data size
AXIOMS
DataLarge = Data // Largest can contain any Data element
DataMed ⊆ DataLarge // Medium can be contained by DataLarge
DataSmall ⊆ DataMed // Small can be contained by DataMed (2)

In most ISAs, *DataLarge*, *DataMed* and *DataSmall* will ultimately correspond to 8, 16 and 32 bit-fields. However, the level of abstraction shown here is maintained to allow the possibility of other sizes and representations to be postponed to the implementation layer of the model (e.g. the inclusion of error detection mechanisms used by some implementations). Elements of type *DataSmall* are represented within *DataMed* and *DataLarge* by the provision of record accessor functions [6]. For example, the following fragment shows the accessor function for the index-zero *DataSmall* element of *DataLarge*:

$$\mathtt{DataLarge2DataSmall0} \in \mathtt{DataLarge} \rightarrow \mathtt{DataSmall} \tag{3}$$

When the mathematical meaning of a data element is required, conversion of the "raw" binary data to their local numerical meanings is provided via similar uninterpreted functions:

$$\mathtt{DataSmall2Nat} \in \mathtt{DataSmall} \rightarrow \mathtt{0..255} \tag{4}$$

5.3 Modeling of Memory-Mapped Data

The data memory system is initially modeled as a simple array mapping between an address range *MemDom* and the generic *Data* set:

$$\mathtt{dataMem} \in \mathtt{MemDom} \rightarrow \mathtt{Data} \tag{5}$$

Further refinement is required to define storage of the three *Data* sub-sets within the data memory. The data memory is initially refined to an array mapping *dataMemR* to *DataSmall*:

$$\mathtt{dataMemR} \in \mathtt{MemDomR} \rightarrow \mathtt{DataSmall} \tag{6}$$

A gluing invariant establishes equivalence across the entire domain of the abstract array:

$$\forall x \bullet x \in MemDom \Rightarrow dataArray(x) = dataArrayR(x) \tag{7}$$

Mappings to *DataLarge* and *DataMed* are defined as groupings of elements of *DataSmall* retrieved via the accessor functions described in Section 5.2, and distributed across a contiguous domain within the refined array. For example, the following fragments state the locations of the index-zero and index-one *DataSmall* extracted elements of a *DataLarge*, at offsets zero and one respectively from the base location of the element.

$$\begin{aligned} &\forall x,A,B,a,b \bullet x \in MemDom \land A \in DataDom \rightarrow Data \land \\ &\qquad B \in MemDom \rightarrow DataSmall \\ &\land\ a \in DataLarge \ \hat{}\ b = DataLarge2DataSmall0(a) \Rightarrow \\ &\qquad A \lhd \{x \mapsto a\} = B \lhd \{x \mapsto b\} \end{aligned} \tag{8}$$

$$\begin{aligned} &\forall x,A,B,a,b \bullet x \in MemDom \land A \in DataDom \rightarrow Data \land \\ &\qquad B \in MemDom \rightarrow DataSmall \\ &\land\ a \in DataLarge \ \hat{}\ b = DataLarge2DataSmall1(a) \Rightarrow \\ &\qquad A \lhd \{x \mapsto a\} = B \lhd \{x+1 \mapsto b\} \end{aligned} \tag{9}$$

These statements define the precise location of the extracted elements of a data element in memory, thus defining the "endianness" of the ISA [7].

6 Model Structuring Rationale

Consideration is given to the order of model refinement in order to allow the model to be re-used with minimal modifications. Some layers of the model provide essential definitions necessary to facilitate later refinements, and therefore these must be positioned earlier in the refinement process. For example, the StateMch layer is required to define the exception condition concept and its triggering of machine deadlock, before any specific exception conditions, such as program counter out-of-range detection, may be defined in the ControlFlowMch. Other refinements may be performed at any point in the modeling process, and are therefore postponed to as late as possible in order to maximize the flexibility of the model. For example, the refinements specifying the exact calculations supported by the VM in CalcMch are positioned prior to only the final MidasMch layer. The decision to place the CalcMch layer after the StackMch layer is made on the assumption that supported calculation operations will be modified more regularly than the selection of a stack machine architecture. Refinements describing the higher level architectural features of the VM, such as the selection of a stack machine, may be easily identified and placed before implementation-specific features, such as exact instruction code values. Decisions on the relative importance of separate architectural issues are harder to make, and judgment based on

projected applications is required. However, such an approach does not consider the consequences of a large number of events being constructed early in the refinement process, and therefore propagating the associated management and proving burden to all subsequent refinements.

The separate modeling of the executable, read-only and read-write memories of the machine allows its applicability to Harvard or Von Neumann memory architectures (i.e. whether the loaded binary program is visible or write-able in the machine's memory system) [10,11]. In the case of the MIDAS demonstrator, a Modified Harvard Architecture, in which the read-only region is visible but the binary program is not, is selected at the MidasMch layer.

7 The MIDAS VM

In order to demonstrate the completeness and usefulness of the modeling technique, a working ISA was developed. Implementation of an existing instruction set and machine architecture was considered: such an approach would allow use of existing support tools and benefit from existing development. Two example machines were considered: an existing Virtual Machine standard, the Java Virtual Machine (JVM), and a typical hardware microprocessor targeted at embedded systems, the Hitachi SH4. Existing architectures and instruction sets contain complexities related to achieving particular requirements not relevant to this application: for example, the JVM contains various features to allow efficient support for the Java high level language [13], and the SH4 instruction set's use of delay-slotted branch instructions [8], enhances performance in a hardware implementation but increases processor and support tool complexity. Full control over the design allows the instruction set to be reduced to a minimum required for program execution and instruction code formats to be selected that are logically based on the formal model.

The MIDAS (Microprocessor Instruction & Data Abstraction System) specification [25] describes a Modified Harvard Architecture, stack-based 32-bit ISA with a total of 42 instructions in 9 orthogonal groups [7], and a little-endian memory system. The instruction groups implement no-operation (1 instruction), stack-push (9 instructions), stack-pop (6 instructions), single-operand calculations (2 instructions), dual-operand calculations (14 instructions), operand compare (6 instructions), control-flow jump (2 instructions), program counter fetch (1 instruction) and machine halt (1 instruction). A stack machine architecture was selected to increase code density, avoid non-linear performance due to out-of-register spills in real-time systems [7], and emulate the architecture of the JVM in a simplified form [13]. A Harvard Architecture was selected for its guaranteed prevention of executable corruption by bad data accesses, increasing integrity in safety-critical applications. In order to reduce ISA size and complexity the MIDAS ISA is not optimized for performance: for example the single operand compare-to-zero instruction provided by many typical ISAs is not implemented [8].

The basic instruction is an 8-bit field, treated as two 4-bit sub-fields (nibbles). The instruction group is specified by the most significant nibble (MSN) and the precise operation given by a modifier in the least significant nibble (LSN). Instruction groups and modifiers are derived from the groupings constructed in the formal model, allowing for an efficient decoding scheme to be implemented, as events applying to whole instruction groups need only decode the MSN.

8 Implementation Generation

EventB and the RODIN tool are intended to support automatic generation of executable source code from sufficiently refined models [4]. This functionality is not yet part of the current RODIN functionality, and therefore a plug-in extension [19] was developed to support a sufficient subset of EventB to support the VM project, translating to the C language. An example showing the final refinement of the MIDAS NOP instruction, and its translated C implementation is given in Figure 4.

Each event is translated to a separate C function returning a boolean signifying whether the event has been triggered. A function is generated to call all event functions in turn until an event is triggered, or signal if no event has been triggered at the end of the machine iteration (i.e. deadlock has occurred).

The translator requires sufficient refinement of the model such that all non-determinism is resolved: precise values are assigned to all ranges and codes, and set membership is reduced to direct comparison operations. Range checking is performed

```
NopOk
REFINES NopOk
ANY
   op
   opVal
   nextInstPtr
WHERE
   grd6: op :  DataSmall
   grd7: op =  mem(instPtr)
   grd5: opVal :  DataSmallNat
   grd2: opVal= DataSmall2Nat(op)
   grd1: opVal =  16
   grd3: instPtr <=  99994
   grd4: statusCode =  2
   grd8: nextInstPtr :  DataLargeNat
   grd9: nextInstPtr =  instPtr + 1
THEN
   act1: instPtr :=  nextInstPtr
END
```

```
/* Event5 [NopOk] */
BOOL NopOk(void)
{
   /* Local variable declarations */
   DataLargeNat nextInstPtr;
   DataSmall op;
   DataSmallNat opVal;

   /* Guard 1 */
   op = mem[instPtr];
   DataSmall2Nat(op,&opVal);
   if(opVal!=16) return BFALSE;

   /* Guard 2 */
   if(instPtr>99994) return BFALSE;

   /* Guard 3 */
   if(statusCode!=2) return BFALSE;

   /* Local assignments in actions */
   nextInstPtr = (instPtr+1);

   /* Actions */
   instPtr = nextInstPtr;

   /* Report hit */
   ReportEventbEvent("NopOk",5);
   return BTRUE;
}
```

Fig. 4. EventB event and derived C

to ensure that numerical values and ranges may be contained by the implementing C type. State variables are disallowed from the right side of actions, in order to prevent use after modification by preceding action-derived statements.

Guard statements are automatically evaluated for one of three possible interpretations: type definition of local parameters, assignments to local parameters, or conditional evaluations. Conditionals are implemented as negations of the basic comparisons enabling early returns from a function, and local parameter assignments are only calculated immediately prior to use. Thus execution is optimized and assignments are only evaluated in a valid context. Comments and instrumentation is inserted to provide traceability between the model and implementation.

9 MIDAS Demonstration

In order to test the constructed MIDAS VM, a support environment is provided using conventional coding techniques. The environment provides a mechanism for download of binary images into MIDAS executable memory, and text output via a virtual console. These facilities are integrated with the MIDAS implementation described in Section 8 using conventional C development tools.

A C compiler targeted at the MIDAS ISA is provided via an appropriate GCC assembler and compiler back-end [21]. Thus a hand-coded C test-suite could be compiled, loaded and executed, demonstrating the suitability of the MIDAS VM for supporting C programs. The test-suite, which was not developed using Formal Methods, is not a complete test of the C language, but includes the most common constructs and invokes the major executable fragments implemented by the compiler. An assembler-coded bootstrap performs segment initialization and machine shutdown. String initialization demonstrates the correct use of the read-only data region, length calculations test integer-based looping, and output to the virtual console tests character manipulation. Integer-to-string conversions test integer arithmetic and integer digit display via C switch statements test use of dispatch tables. Passing of function arguments and results test variable passing via stack pushes. Explicit tests exercise floating point arithmetic and casting. Integer field extraction functions test bit-wise shift and masking functions. Nested for and if statements test in-function nesting. Nested calls test deeply stacked function calls and returns.

10 EventB and RODIN

Experience gained during the development of the generic VM model and MIDAS demonstrator allows future improvements to both EventB and the RODIN tool to be suggested. Model development using multiple refinement steps requires considerable repetition of event guard and action sections: in larger models refinement can lead to small changes being difficult to identify amongst previously constructed functionality. As the principle of guard strengthening allows additional guards to be legitimately introduced, erroneous guards may be introduced, potentially leading to unintended deadlock conditions. A suggested solution is the extension of the current

EventB "inherited" property, which signifies that an event is to be entirely repeated from a previous step. Expression of an event's guards and actions as incremental additions to those of a previous refinement step would yield more concise, reliable notation, whilst still allowing a complete event to be inspected via appropriate database viewing tools.

Within the RODIN tool, the concept of a stored model database, indirectly editable via specialized tools, allows for efficient storage and traversing of model logic by tool extensions. Improvements in the ergonomics of the editing interface are required to achieve productivity similar to that of conventional text editing interfaces. Search, replace and pasting capabilities within a single view would allow faster, less error-prone development.

In common with other logical proof tools, the RODIN provers are susceptible to incorrect discharge of proof obligations due to false predicates introduced by accidental introduction of contradictory statements within a model. It is suggested that vacuity-checking techniques be introduced to defend against such conditions, as incorporated in other commercially available tools [3].

11 Future Work

The EventB VM model and MIDAS demonstration ISA suggest a number of future areas of investigation. The VM model has been developed using incremental refinement techniques, with all generated POs discharged in order to prove consistency between refinement steps. However, this analysis does not guarantee correctness of the model itself: in the case of the VM, checking against implicit deadlock states is of particular importance. Tools exist for the checking of model correctness [12], and use of these for checking of the VM model is desirable.

The VM model currently allows for non-determinism under certain conditions: for example, two separate events are constructed to raise exceptions if either the source or destination are unavailable on an attempted data transfer, but the event triggered in the case of both being unavailable is not defined. Expansion of the model to deterministically enumerate all such conditions would yield a more precise specification of VM behavior under such error conditions. Expansion of the model to incorporate other common ISA features is also desirable: for example the construction of events transferring data between the registers of the RegMch model layer.

Formally derived models have been recognized as a possible basis for generation of testing criteria [23]. The VM application allows the opportunity for testing to be performed on a deployed machine, test inputs being provided by the loading of appropriate binary executables.

The MIDAS ISA and compiler have been demonstrated to support a hand-coded example application: testing of the VM against a complete C test-suite is required to fully demonstrate the VM [20]. The MIDAS ISA has been specified to demonstrate the EventB modeling technique without regard to other metrics such as performance: expansion of the ISA to include additional performance-enhancing instructions and features, within the EventB modeling paradigm discussed here, is possible.

12 Conclusions

EventB allows the generic properties of binary Instruction Set Architectures to be captured in an abstract model, thus providing a re-usable template for the development of Formally Proved computing machines. The EventB refinement process allows an incremental structure in this abstract model, maximizing its re-usability, and its concretization to a level sufficient for automatic conversion to a usable implementation. Constructed relationships derived within the model may also be used to guide specification of new ISAs.

The RODIN tool enables the management of the multiple refinement stages and Formal Proof analysis required for such a technique, and provides the capability for necessary implementation generation tools to be developed.

The technique has been demonstrated by the construction of such a model, and its refinement to an implementation in the form of a Virtual Machine capable of running compiled binary images.

References

1. AMD Inc. 28-Bit SSE5 Instruction Set (2007)
2. Audsely, N.: Portable Code in Future Avionic Systems, IEE Colloquium on Real-Time Systems (Digest No. 1998/306) (1998)
3. Beer, I., Ben-David, S.: RuleBase: Model checking at IBM. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
4. Butler, M.: RODIN Deliverable D16 Prototype Plug-in Tools (2006), http://rodin.cs.ncl.ac.uk
5. Caset, L.: Formal Development of an Embedded Verifier for Java Card Byte Code. In: International Conference on Dependable Systems and Networks (2002)
6. Evans, N., Butler, M.: A Proposal for Records in Event-B Formal Methods 2006 (2006)
7. Hennessy, J., Patterson, D.: Computer Architecture, A Quantitive Approach. Morgan Kaufmann, San Francisco (2003)
8. Hitachi Ltd. SH7707 Hardware Manual (1998)
9. Kernighan, B., Ritchie, D.: The C Programming Language. Prentice Hall, Englewood Cliffs (1988)
10. Lapsley, P., Bier, J., Shoham, A., Lee, E.: DSP Processor Fundamentals. IEEE Press, Los Alamitos (1997)
11. Lee, E.: Programmable DSP Processors part I and II. IEEE ASSP Mag. (October 1988–January 1989)
12. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: FME 2003, SpringerLink (2003)
13. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn (1999)
14. Lowry, H., Mitchell, B.: Mission computer replacement prototype for Special Operations Forces aircraft an application of commercial technology to avionics. In: Proceedings of The 19th Digital Avionics Systems Conference (2000)
15. Luke, J., Haldeman, D.: Replacement Strategy for Aging Avionics Computers. IEEE Aerospace and Electronic Systems Magazine 14(3) (1999)
16. Metayer, C., Abrial, J.-R., Voisin, L.: RODIN Deliverable 3.2 Event-B Language (2005), http://rodin.cs.ncl.ac.uk
17. Moore, J.: A Grand Challenge Proposal for Formal Methods A Verified Stack. In: Formal Methods at the Crossroads From Panacea to Foundational Support, SpringerLink (2003)

18. Schneider, S.: The B-Method An Introduction. Palgrave (2001)
19. Shavor, S., D'Anjou, J., Fairbrother, S.: The Java Developer's Guide to Eclipse. Addison-Wesley, Reading (2003)
20. Sherridan, F.: Practical Testing of a C99 Compiler Using Output Comparison. Software: Practical and Experience 37(14) (2007)
21. Stallman, R.: Using and Porting the GNU Compiler Collection, Free Software Foundation (2001)
22. Stark, R., Schmid, J., Borger, E.: Java and the Java Virtual Machine. Springer, Heidelberg (2001)
23. Utting, M., Legeard, B.: Practical Model-Based Testing – A Tools Approach. Morgan Kaufmann, San Francisco (2007)
24. Voisin, L.: A Description of the RODIN Prototype (2006), http://rodin.cs.ncl.ac.uk
25. Wright, S.: MIDAS Design Document CSTR-06-014, Bristol University (2008), http://www.cs.bris.ac.uk/Publications/Papers/2000543.pdf
26. http://www.hercules-390.org/

Z2SAL - Building a Model Checker for Z

John Derrick, Siobhán North, and Anthony J.H. Simons

Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK
J.Derrick@dcs.shef.ac.uk

Abstract. In this paper we discuss our progress towards building a model-checker for Z. The approach we take in our Z2SAL project involves implementing a translation from Z into the SAL input language, upon which the SAL toolset can be applied. The toolset includes a number of model-checkers together with a simulator. In this paper we discuss our progress towards implementing as complete as a translation as possible, the limitations we have reached and the optimizations we have made. We illustrate with a small example.

Keywords: Z, model-checking, SAL.

1 Introduction

Z has, for some time, lagged behind other specification languages in its provision of tools. There are a number of reasons for this, although most are connected with the language itself and its semantics: its expressivity has made it more difficult to build tractable tools. However, recently a number of projects have begun to tackle this defficiency. These include the CZT (Community Z Tools) project [8], our own work [6], as well as related work such as ProZ [9], which adapts the ProB [7] tool for the Z notation, and that of Bolton who has used Alloy to verify data refinements in Z [1].

Our concern is that of providing a model-checking [4] tool via translation of Z specifications into the input language of an appropriate toolset. Here we choose the SAL [5] tool-suite, designed to support the analysis and verification of systems specified as state-transition systems. Its aim is to allow different verification tools to be combined, all working on an input language designed as a format into which programming and specification languages can be translated. The input language provides a range of features to support this aim, such as guarded commands, modules, definitions etc., and can, in fact, be used as a specification language in its own right. The tool-suite currently comprises a simulator and four model checkers including those for LTL and CTL.

The original idea of translating Z into SAL specifications was due to Smith and Wildman [10]. In [6] we described the basics of our implementation, which essentially is a Java based compiler of a subset of Z into SAL. Here we discuss the implementation in broader scope, describing how different parts of the Z mathematical toolkit are translated.

The aim of [10] was to preserve the Z-style of specification including predicates where primed and unprimed variables are mixed, and the approach of the Z

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 280–293, 2008.

mathematical toolkit to the modelling of relations, functions etc., as sets of tuples. Given this theoretical basis (the translation in [10] was not optimized for implementation) the actual implementation has preserved this general approach but has increasingly diverged as optimization issues have been tackled.

The general scheme of the translation is that a Z specification is translated to a SAL module, which groups together a number of definitions including types, constants and modules for describing a state transition system. The declarations in a state schema in Z are translated into local variables in a SAL module, and any state predicates become appropriate invariants over the module and its transitions.

A SAL specification defines its behaviour by specifying transitions, thus it is natural to translate each Z operation into one branch of a guarded choice in the transitions of the SAL module. The predicate in the operation schema becomes a guard of the particular choice. The guard is followed by a list of assignments, one for each output and primed declaration in the operation schema.

As would be expected the work to be done in such a translation is in translating the mathematical toolkit; yet a naive translation quickly produces SAL input which is infeasible to simulate or model-check. Thus our work has optimized the translation as much as possible by using appropriate combinations of the inbuilt SAL types. Much of our discussion below concerns these issues.

The structure of the paper is as follows. The basic architecture of our implementation is described in Section 2. Section 3 gives an overview of the approach to translation, and more specifics about translating the mathematical toolkit is given in Section 4. Finally, Section 5 discusses the use of the tool.

2 The Z2SAL Architecture

The Z2SAL tool is currently implemented in Java directly (rather than using the CZT components) in order to rapidly prototype and evaluate a number of ideas. At present it works by scanning a Z LaTeXsource file in a single pass into something which is basically a list of schema class instances with associated classes representing the named constants, types and variables and one expression structure to represent the restrictions derived from the constraints in the axiomatic definitions. The use of a single scan is feasible because we restrict the Z we accept to definition before use of all identifiers (which in practice is not a significant limitation).

Having parsed the Z a certain amount of optimisation is performed with a view to producing efficient SAL. Given we are producing output for use with a model-checker, the first step is to turn any aspect that might be unbounded into a finite size. In order to keep the state space to a minimum the size of any basic types we use is also restricted as far as possible. Thus if $\mathbb{Z}$ is used in Z a type called `INT` will be declared in the SAL output which ranges between one less than the smallest constant used in the Z to one more than the largest. Given types from the Z specification have to be assigned explicit values in the SAL

output, and by default they are set to consist of a type with three elements but this can be varied by supplying the parser with a different value as a parameter.

The combination of small fixed ranges for basic types and giving all constants a value allows us to optimize expressions derived from both the constraints of the schemas and the axiomatic definitions. The latter are optimized first. Z expressions are initially parsed into a conventional tree but this is immediately transformed into a list of subtrees which represents the series of conjoined predicates. This is both a natural way to represent the predicate in a Z schema and a convenient structure to modify when combining predicates derived from different sources, something we have to do in various places starting with the predicates of all the axiomatic definitions.

This combined predicate is scanned both to eliminate redundant predicates and to restrict all the variables constrained by any axiomatic definitions to as small a range as possible. In an earlier release (see [6]) we dealt with these variables by giving them arbitrary values and treating them as constants, however, it turned out that the properties of the resulting SAL were too dependent on the translator's choice. Thus here we restrict their type as far as possible using the conditions imposed by the axiomatic definitions constraints. Sometimes the type restriction process allows a variable to be restricted to a single value, and at that point it is transformed into a constant, removing the need for a SAL predicate, and this in turn can lead to further optimization. If, during this process a Z predicate proves to be unsatisfiable, the translator terminates under the assumption that the Z specification is erroneous. Any predicates remaining, ones that cannot be converted into restrictions of type, are then added to the state schema predicates and the variables are added to those of the state schema.

Next, all the variables are scanned to identify any types that must be constructed in SAL. For example (see Appendix A) to express $rented : PERSON \leftrightarrow TITLE$ in SAL we have to create a symbolic name for a SAL product type `PERSON__X__TITLE : TYPE = [PERSON, TITLE]` for use in the declaration `rented: set{PERSON__X__TITLE}!Set` since the product type cannot be used directly in the instantiation of the set context. So, unless the type is named elsewhere, we have to generate an artificial name for it, which is declared early in the SAL output. Finally the schema predicates are scanned, to optimize them and also to identify any extra declarations we need, including those required by a `count` context (see below), which is used to express set cardinality.

Having cleaned up the Z as far as possible, in the manner just described, the SAL is generated. First the named types and constants are generated. The order of these is insignificant in SAL but, from a human point of view, it is useful to have them in the same order as they appear in the original Z. To this end a list of identifiers in order of first use is kept by the lexical analyser and can be used to order the initial declarations in the SAL correctly. After this the types and counters generated by the translator are generated, the state invariant and finally the operation schemas, transformed into transitions, are exported. Here again we maintain the same order for readability.

3 Overview of the Translation Strategy

A specification in the SAL input language consists of a number of *context* files, in which all the declarations are placed. In our translation, we use a master `CONTEXT` for the main Z specification and refer to other context files, which define the behaviour of the mathematical toolkit. The master context includes declarations of the basic types and constants; and declares the finite state automaton, known as a `MODULE`, which represents the Z state schema as a collection of local state variables and the Z operation schemas as transitions of the automaton, acting on the local, input and output variables.

Types: We adopt the following scheme for the translation of Z types

Z	SAL translation
Built-in types such as $\mathbb{N}$ etc	Finite subranges of SAL equivalent types
Given sets	Enumerated finite type in SAL
Free types	Constructed type in SAL

So, in the example in Appendix A, the given type [*PERSON*] is translated to: `PERSON : TYPE = {PERSON__1, PERSON__2, PERSON__3};` SAL constructed types may be recursive, but some implementations of the SAL tools cannot process recursive definitions because they expand all recursive constructions infinitely as the definitions are compiled to BDDs. This problem may be fixed in future releases of the SAL toolset. We assume for the moment that the input does not contain recursively-defined data types.

Constants and Axiomatic Definitions: The most direct way to translate constants and axiomatic definitions is to declare them as a SAL local variable, within the module clause. However, this multiplies the state space of the system, but with many of the states being over-constrained. We thus attempt to identify suitable exact values for constants in the translation, which can often be done by looking at predicates which involve the constant elsewhere in the specification.

State and Initialisation Schemas: The Z state schema is converted into `LOCAL` variable declarations within the `MODULE` clause, with a corresponding `DEFINITION` clause to represent the schema invariant. This defines a local abbreviation `invariant__` for a boolean equation expressing constraints on the values of local variables; and could also include further constraints resulting from axiomatic definitions. The Z initialization schema is translated in a non-constructive style into a guarded command in the `INITIALIZATION` clause of the SAL module, with the invariant as part of the guard.

Operation Schemas: The translation of Z operation schemas into SAL consists of three stages. First, all input and output variables are extracted and converted into their cognate forms in SAL. In addition, each operation schema is converted into a single transition, such that the Z schema predicate becomes the guard for the guarded command, expressing the relationship between the *primed* and *unprimed* versions of variables. The primed `invariant__'` is added to the guard, to indicate that the invariant must hold after firing each transition.

Finally, a catch-all ELSE branch is added to the guarded commands, to ensure that the transition relation is total (for soundness of the model checking).

In Z. input and output variables are locally scoped to each operation schema, but exist in the same scope in the SAL MODULE clause, therefore we prefix input and outputs by the name of the Z schema from which they originate (additionally, outputs use an underscore decoration rather than a !).

Thus the example in *Appendix A* is translated into the following SAL fragment (we have elided parts of the translation - the complete output is in *Appendix B*). In particular, the assignment of updated values occurs before the --> in the transitions in this non-constructive style of encoding.

```
example : CONTEXT = BEGIN
    PERSON : TYPE = {PERSON__1, PERSON__2, PERSON__3};
    NAT : TYPE = [0..4];
...
State : MODULE =
  BEGIN
    LOCAL members : set {PERSON;} ! Set
    LOCAL rented : set {PERSON__X__TITLE;} ! Set
    LOCAL stockLevel : [ TITLE -> NAT ]
    INPUT RentVideo__p? : PERSON
    INPUT AddTitle__t? : TITLE
...
    OUTPUT CopiesOut__copies_ : NAT
    LOCAL invariant__ : BOOLEAN
    DEFINITION
      invariant__ = ...
    INITIALIZATION [
        members = set {PERSON;} ! empty  AND
      stockLevel = function{TITLE, NAT; TITLE__B, 4}!empty AND invariant__
      -->
    ]
    TRANSITION [
      RentVideo :
...
        -->
          members' IN { x : set {PERSON;} ! Set | TRUE};
          rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
          stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
      []
      AddTitle :
...
     ELSE -->
END
```

4 The Mathematical Toolkit

The heart of the translation deals with the Z mathematical toolkit, which provides a rich vocabulary of mathematical data types, including sets, products, relations,

functions, sequences and bags. The challenge is to represent these types, and the operations that act upon them, efficiently in SAL, whilst still preserving the expressiveness of Z. The basic approach is to define one or more context files for each data type in the toolkit. For example, the *set*-context implements a set as a function from elements to `BOOLEAN`. This is a standard encoding for sets, optimized for symbolic model checkers that use BDDs as the core representation [2,3]. A set is not a single, monolithic entity, but rather a polylithic membership predicate over all of its elements. Set operations are defined in the following style:

```
union(setA : Set, setB : Set) : Set =
  LAMBDA (elem : T) : setA(elem) OR setB(elem);
```

However, the encoding causes problems when calculating the cardinality of sets. In [10] cardinality is defined as the search for a relation between sets and natural numbers. However, we found that this was inefficient when implemented [6]. We also tried two other encodings for counted sets, which relied on brute-force enumeration of elements. Since then, we have removed the `size?` function altogether from the standard, efficient *set*-context and provide this in a separately generated `countN`-context, parameterized over arbitrary N elements. This separation of concerns provides brute-force counting only when the specification actually requires this. For example, three possible elements may be counted by the `size?` function from the *count*3-context:

```
count3{T : TYPE; e1, e2, e3 : T} : CONTEXT =
BEGIN
  Set : TYPE = [T -> BOOLEAN];
    size? (set : Set) : NATURAL =
    IF set(e1) THEN 1 ELSE 0 ENDIF +
    IF set(e2) THEN 1 ELSE 0 ENDIF +
    IF set(e3) THEN 1 ELSE 0 ENDIF;
END
```

A similar strategy was adopted for encoding relations. The initial idea was to define relations as sets of pairs. We found that the SAL parser rejected type-instantiation with anything other than a simple symbolic type name, which initially limited the use of constructed product-types, but we later adopted the work-around of defining symbolic aliases for each product-type.

The standard *relation*-context is parameterized over the domain and range element-types, and internally defines two pair-types, two set-types for the domain and range, and two set-types for the relation and inverse relation, followed by operations on relations:

```
relation{X, Y : TYPE; } : CONTEXT =
BEGIN
  XY : TYPE = [X, Y];
  YX : TYPE = [Y, X];
  Domain : TYPE = [X -> BOOLEAN];
```

```
  Range : TYPE = [Y -> BOOLEAN];
  Relation : TYPE = [XY -> BOOLEAN];
  Inverse : TYPE = [YX -> BOOLEAN];
  ...
  domain (rel : Relation) : Domain =
    LAMBDA (x : X) : EXISTS (y : Y) :
      LET pair : XY = (x, y) IN rel(pair);
END
```

This translation makes maximally-efficient use of the direct encoding of sets as boolean functions, for example, using internal variables (introduced by `LET`) to facilitate the toolset's manipulation of symbolic structures.

There was the option of re-implementing all of the set operations again in the relation-context; however, for efficiency we provide definitions only for the *additional* operations upon relations. In our example, specific relation operations (such as `domain`) are selected from this *relation*-context, whereas standard set-operations (such as `contains?`) are selected from a *set*-context, treating the same relation as a set of pairs:

```
  ... relation {PERSON, TITLE;} ! domain(rented) ...
  ... set {PERSON__X__TITLE;} ! contains?(rented,
                      (RentVideo__p?, RentVideo__t?)) ...
```

The success of this partitioning approach motivated our splitting the complete definition of relations over three contexts, according to the number of types related. The standard context provides all operations on relations between two distinct base types. A separate context provides all operations on relations closed over a single type (such as identity, transitive closure); while a third context defines relational composition, relating three types. The translator exports these contexts only if they are needed.

In translating Z functions, we could either model them as sets of pairs, thereby easing the integration into the models for sets and relations, or use the SAL built-in function type. Timing experiments confirmed that using native SAL functions was far more efficient. However, SAL functions are *total*. To support the more commonly-occurring partial functions in Z we adopt a *totalizing* strategy, in which every type appearing in a function signature is extended with a *bottom* value. This is typically an extra symbolic value (such as `TITLE__B`) for basic types and an out-of-range value for numeric types. Partial Z functions are converted into total SAL functions, in which some domain or range values are *bottom*. At the same time, extra invariants are added to the translation of Z operation schemas to assert that input and output variables never take the *bottom* value. This approach was more scalable than defining two versions of each type, with and without *bottom*.

The *function*-context is parameterized over the domain and range element-types, but also includes value-parameters for the *bottom* element of each of these types. This allows operations on functions to recognize undefined cases. The context defines the function-type, but also pair-types for the corresponding relation and inverse relation, and supplies a `convert` operation to convert a SAL-function back into our preferred encoding for a relation as a set of pairs:

```
function {X, Y : TYPE; xb : X, yb : Y} : CONTEXT =
BEGIN
  XY : TYPE = [X, Y];
  YX : TYPE = [Y, X];
  Function : TYPE = [X -> Y];
  Relation : TYPE = [XY -> BOOLEAN];
  Inverse : TYPE = [YX -> BOOLEAN];
  Domain : TYPE = [X -> BOOLEAN];
...
  convert (fun : Function) : Relation =
    LAMBDA (pair : XY) : fun(pair.1) = pair.2
      AND pair.1 /= xb AND pair.2 /= yb;
END
```

Since this encoding is quite different from the relation encoding, we re-implemented some of the standard operations on relations, to optimize these for functions, eg:

```
  image (fun : Function, set : Domain) : Range =
    LAMBDA (y : Y) : EXISTS (x : X) :
      set(x) AND fun(x) = y AND y /= yb;
```

Here, maximal use is made of native function application, which is extremely efficient in SAL, while including extra side-constraints to rule out mappings that would include *bottom*. To simplify the definition of these and other operations on functions, a global constraint `fun(xb) = yb` is asserted in the main context.

Z distinguishes many function types for plain, injective, surjective and bijective functions (in total and partial combinations). The strategy in SAL is not to create additional function types, which would either require duplication of all function operations, or would prevent treating e.g. an injective function just as a plain function. Instead, the Z definitions of each function type are converted into predicates, that are added to the system invariant. For example, a (partial) surjective function is constrained by the predicate:

```
  surjective? (f : Function) : BOOLEAN =
    FORALL (y : Y) : EXISTS (x : X) : f(x) = y;
```

and the extra conjunction is added to constrain *myFun* in the invariant:

```
  DEFINITION invariant__ = ...
    AND function{Dom, Ran, dom__b, ran__b}! surjective?(myFun)...
```

The predicates must be coded in such a way that they are not violated if the functions are in fact empty, as is typical at the start of a simulation.

Finally, the SAL contexts encoding Z sets, functions and relations also include a number of optimized operations for dealing with common Z cases. For example, the insertion of single elements into sets is expressed in Z as the union of a set with a constructed singleton set. In SAL, this can be achieved much more simply with an extra `insert` operation in the *set*-context:

```
insert (set : Set, new : T) : Set =
  LAMBDA (elem : T) : elem = new OR set(elem);
```

Likewise, the Z style of replacing maplets in functions using the override operator is handled much more efficiently by providing an extra `insert` operation in the *function*-context:

```
insert (fun : Function, pair : XY) : Function =
  LAMBDA (x : X) : IF x = pair.1
    THEN pair.2 ELSE fun(x) ENDIF;
```

These special cases are identified in our parser. Similar special operations are supplied for empty and universal sets, singleton sets and empty functions.

5 Use of the Tool

The SAL toolset uses a command line interface, as does our translator. The translator accepts the LaTeX markup as defined in the Z standard, and the translator output is a plain SAL file. We have work in progress to port the translator to accept the ZML markup for Z, using the ASTs constructed by the parser produced by the CZT project [8]. We also have work in progress to build a GUI interface to the command-line tools and interpret the results, which are rather dense at the moment.

Simulation: The SAL translation of the example can be simulated by running the `sal-sim` tool and loading the SAL file. The compilation process takes about 6-7 seconds for this example on a standard desktop. Our example creates 11664 initial states, most of which are due to assigning all possible values to the `INPUT` and `OUTPUT` variables (since we initialise the `LOCAL` variables to fixed values). While it is necessary to represent all possible input conditions for the first simulation step, we could reduce the number of initial states by constraining the unused values of output variables. The simulation is triggered by repeated calls to the `(step!)` function and the number of resulting states may be viewed:

Step	0	1	2	3	4	5
States	11664	221040	1752048	7918848	24593328	61568640

As well as displaying n of the states found at each step, it is possible to see an arbitrary trace through the system, by a command which selects a random trace. For example, after five steps, a trace is returned that shows how the system performed the following:

Step	Transition	Updates
0	Init	$members, rented, stockLevel = \varnothing$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_2 \in members$
3	AddMember	$PERSON\_3 \in members$
4	RentVideo	(PERSON_3, TITLE_2) $\in rented$
5	Else	no change

From this, it can be seen that the system acquired some videos and members and rented a title to one of the members. The final step selected the default `ELSE`-transition, a nullop that is always possible, in case a simulation deadlocks.

Model Checking: The SAL toolkit has several simple and bounded model-checkers that support both LTL and CTL temporal logics. At the moment, we add theorems by hand to the end of the translated SAL file. Eventually, we expect to add an extension to Z to express theorems in temporal logic.

Suppose that we want to show that videos eventually get rented to members of the video club. In SAL, we propose the negation of this as a theorem:

```
th1 : THEOREM State |- G(set {PERSON__X__TITLE;}!empty?(rented));
```

This says that "the `State` module allows us to derive that the relation `rented` is always empty," using the LTL operator `G` for always. We run this through the model checker and this generates the smallest counterexample that proves the desired property:

Step	Transition	Updates
0	Init	$members, rented, stockLevel = \varnothing$
1	AddTitle	$stockLevel(TITLE\_2) = 3$
2	AddMember	$PERSON\_1 \in members$
3	RentVideo	$(PERSON\_1, TITLE\_2) \in rented$

For our example, the time taken is again about 6-7 seconds, however, the majority of time is taken up compiling the example, and the execution time to find the counterexample was 0.17 seconds. Proper evaluation and scalability is left for future work.

6 Conclusion

In conclusion, we have achieved a fairly efficient translation of Z into SAL, demonstrating the benefits of encodings that are close to SAL's internal BDD structures and giving heuristics for reducing the initial state-space. New results reported in this paper include the translation of schema invariants and the optimized datatypes for the Z mathematical toolkit. We have also identified some problems in handling constructed and recursive types in SAL. Future work will include translating the rest of the mathematical toolkit.

Acknowledgements. This work was done as part of collaborative work with the University of Queensland, and in particular, Graeme Smith and Luke Wildman. Tim Miller also gave valuable advice on the current CZT tools.

References

1. Bolton, C.: Using the Alloy Analyzer to Verify Data Refinement in Z. Electronic Notes in Theoretical Computer Science 137(2), 23–44 (2005)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)

3. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), SRI International (2003)
6. Derrick, J., North, S., Simons, T.: Issues in implementing a model checker for z. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 678–696. Springer, Heidelberg (2006)
7. 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)
8. Miller, T., Freitas, L., Malik, P., Utting, M.: CZT Support for Z Extensions. In: Romijn, J., Smith, G., Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 227–245. Springer, Heidelberg (2005)
9. 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)
10. Smith, G., Wildman, L.: Model checking Z specifications using SAL. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 85–103. Springer, Heidelberg (2005)

Appendix A

The following defines a video shop and the process of renting videos etc.

$[PERSON, TITLE]$

State

$$members : \mathbb{P}\, PERSON$$
$$rented : PERSON \leftrightarrow TITLE$$
$$stockLevel : TITLE \nrightarrow \mathbb{N}$$
$$\text{dom}\, rented \subseteq members$$
$$\text{ran}\, rented \subseteq \text{dom}\, stockLevel$$

Init

$$State'$$
$$members' = \varnothing$$
$$stockLevel' = \varnothing$$

RentVideo

$$\Delta State$$
$$p? : PERSON$$
$$t? : TITLE$$
$$p? \in members$$
$$t? \in \text{dom}\, stockLevel$$
$$stockLevel(t?) > \#(rented \rhd \{t?\})$$
$$(p?, t?) \notin rented$$
$$rented' = rented \cup \{(p?, t?)\}$$
$$stockLevel' = stockLevel$$
$$members' = members$$

AddTitle

$$\Delta State$$
$$t? : TITLE$$
$$level? : \mathbb{N}$$
$$stockLevel' = stockLevel \oplus \{(t?, level?)\}$$
$$rented' = rented$$
$$members' = members$$

DeleteTitle

$\Delta State$
$t? : TITLE$

$t? \notin \operatorname{ran} rented$
$t? \in \operatorname{dom} stockLevel$
$stockLevel' = \{t?\} \lhd stockLevel$
$rented' = rented$
$members' = members$

AddMember

$\Delta State$
$p? : PERSON$

$p? \notin members$
$stockLevel' = stockLevel$
$rented' = rented$
$members' = members \cup \{p?\}$

CopiesOut

$\Xi State$
$t? : TITLE$
$copies! : \mathbb{N}$

$t? \in \operatorname{dom} stockLevel$
$copies! = \#(rented \rhd \{t?\})$

Appendix B

This following is the SAL output from the translation of the above.

```
example : CONTEXT = BEGIN

PERSON : TYPE = {PERSON__1, PERSON__2, PERSON__3}; TITLE : TYPE =
{TITLE__1, TITLE__2, TITLE__3, TITLE__B}; PERSON__X__TITLE : TYPE
= [PERSON, TITLE]; NAT : TYPE = [0..4];

PERSON__X__TITLE__counter : CONTEXT = count12 {PERSON__X__TITLE;
  (PERSON__1, TITLE__1), (PERSON__1, TITLE__2), (PERSON__1, TITLE__3),
  (PERSON__1, TITLE__B), (PERSON__2, TITLE__1), (PERSON__2, TITLE__2),
  (PERSON__2, TITLE__3), (PERSON__2, TITLE__B), (PERSON__3, TITLE__1),
  (PERSON__3, TITLE__2), (PERSON__3, TITLE__3), (PERSON__3, TITLE__B)};

State : MODULE =
  BEGIN
    LOCAL members : set {PERSON;} ! Set
    LOCAL rented : set {PERSON__X__TITLE;} ! Set
    LOCAL stockLevel : [ TITLE -> NAT ]
    INPUT RentVideo__p? : PERSON
    INPUT RentVideo__t? : TITLE
    INPUT AddTitle__t? : TITLE
    INPUT AddTitle__level? : NAT
    INPUT DeleteTitle__t? : TITLE
    INPUT AddMember__p? : PERSON
    INPUT CopiesOut__t? : TITLE
    OUTPUT CopiesOut__copies_ : NAT
    LOCAL invariant__ : BOOLEAN
```

```
DEFINITION
  invariant__ = (set {PERSON;} ! subset?(relation {PERSON, TITLE;} !
        domain(rented), members)  AND
    set {TITLE;} ! subset?(relation {PERSON, TITLE;} ! range(rented),
        function {TITLE, NAT; TITLE__B, 4} ! domain(stockLevel))  AND
    stockLevel (TITLE__B) = 4  AND
    RentVideo__t? /= TITLE__B  AND
    AddTitle__t? /= TITLE__B  AND
    AddTitle__level? /= 4  AND
    DeleteTitle__t? /= TITLE__B  AND
    CopiesOut__t? /= TITLE__B  AND
    CopiesOut__copies_ /= 4)
INITIALIZATION [
    members = set {PERSON;} ! empty  AND
    stockLevel = function {TITLE, NAT; TITLE__B, 4} ! empty AND invariant__
  -->
]
TRANSITION [
  RentVideo :
      set {PERSON;} ! contains?(members, RentVideo__p?)  AND
      set {TITLE;} ! contains?(function {TITLE, NAT; TITLE__B, 4} !
          domain(stockLevel), RentVideo__t?)  AND
      stockLevel (RentVideo__t?) > PERSON__X__TITLE__counter !
          size?(relation {PERSON, TITLE;} ! rangeRestrict(rented, set
          {TITLE;} ! singleton(RentVideo__t?)))  AND
      NOT set {PERSON__X__TITLE;} ! contains?(rented, (RentVideo__p?,
          RentVideo__t?))  AND
      rented' = set {PERSON__X__TITLE;} ! insert(rented, (RentVideo__p?,
          RentVideo__t?))  AND
      stockLevel' = stockLevel  AND
      members' = members AND
      invariant__'
    -->
      members' IN { x : set {PERSON;} ! Set | TRUE};
      rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
      stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
  []
  AddTitle :
      stockLevel' = function {TITLE, NAT; TITLE__B, 4} ! insert(stockLevel,
          (AddTitle__t?, AddTitle__level?))  AND
      rented' = rented  AND
      members' = members AND
      invariant__'
    -->
      members' IN { x : set {PERSON;} ! Set | TRUE};
      rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
      stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
  []
  DeleteTitle :
      NOT set {TITLE;} ! contains?(relation {PERSON, TITLE;} !
```

```
            range(rented), DeleteTitle__t?)  AND
        set {TITLE;} ! contains?(function {TITLE, NAT; TITLE__B, 4} !
            domain(stockLevel), DeleteTitle__t?)  AND
        stockLevel' = function {TITLE, NAT; TITLE__B, 4} ! domainSubtract(set
            {TITLE;} ! singleton(DeleteTitle__t?), stockLevel)  AND
        rented' = rented  AND
        members' = members AND
        invariant__'
      -->
        members' IN { x : set {PERSON;} ! Set | TRUE};
        rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
        stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
    []
    AddMember :
        NOT set {PERSON;} ! contains?(members, AddMember__p?)  AND
        stockLevel' = stockLevel  AND
        rented' = rented  AND
        members' = set {PERSON;} ! insert(members, AddMember__p?) AND
        invariant__'
      -->
        members' IN { x : set {PERSON;} ! Set | TRUE};
        rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
        stockLevel' IN { x : [ TITLE -> NAT ] | TRUE}
    []
    CopiesOut :
        members = members' AND
        rented = rented' AND
        stockLevel = stockLevel' AND
        set {TITLE;} ! contains?(function {TITLE, NAT; TITLE__B, 4} !
            domain(stockLevel), CopiesOut__t?)  AND
        CopiesOut__copies_' = PERSON__X__TITLE__counter ! size?(relation
            {PERSON, TITLE;} ! rangeRestrict(rented, set {TITLE;} !
            singleton(CopiesOut__t?))) AND
        invariant__'
      -->
        members' IN { x : set {PERSON;} ! Set | TRUE};
        rented' IN { x : set {PERSON__X__TITLE;} ! Set | TRUE};
        stockLevel' IN { x : [ TITLE -> NAT ] | TRUE};
        CopiesOut__copies_' IN { x : NAT | TRUE}
    []
    ELSE -->
  ]
 END;

END
```

Formal Modeling and Analysis of a Flash Filesystem in Alloy

Eunsuk Kang and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA. U.S.A
{eskang,dnj}@mit.edu

Abstract. This paper describes the formal modeling and analysis of a design for a flash-based filesystem in Alloy. We model the basic operations of a filesystem as well as features that are crucial to NAND flash hardware, such as wear-leveling and erase-unit reclamation. In addition, we address the issue of fault tolerance by modeling a mechanism for recovery from interrupted filesystem operations due to unexpected power loss. We analyze the correctness of our flash filesystem model by checking trace inclusion against a POSIX-compliant abstract filesystem, in which a file is modeled simply as an array of data elements. The analysis is fully automatic and complete within a finite scope.

1 Introduction

Flash memory is becoming an increasingly popular choice of medium for non-volatile data storage. Among a wide range of applications, flash memory has been used by NASA for on-board storage in planetary rovers. In one well known incident, *Spirit*, a Mars Exploration Rover, suffered a major system failure that resulted in 10 days of lost scientific activity [19]. The cause of the failure was later determined to be a flaw in the flash filesystem software. On investigation, it turned out that the failure scenario was neglected during testing because the development team considered it to be an "unanticipated behaviour."

Testing is an essential part of any software development process, but cannot alone ensure the reliability of software. Formal methods can mitigate the weakness of testing by allowing an exhaustive analysis. However, applying formal methods to a poorly designed piece of code in an after-the-fact, ad hoc fashion is impractical, and rarely yields high confidence for reliability. Instead, by formalizing important aspects of a design and analyzing them early in a development process, software engineers can identify key reliability issues and address them in the simpler context of the design, where they can be resolved before the complexities of implementation are introduced.

This paper describes the formal modeling and analysis of a design for a flash filesystem in Alloy [16]. Our model addresses three primary aspects of a flash filesystem: (1) The underlying flash hardware, (2) filesystem software with basic file operations such as read and write, and (3) a fault-tolerance mechanism

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 294–308, 2008.

for handling unexpected hardware failures. Unlike disk-based storage devices, flash memory suffers from a major limitation in that blocks can be written only a limited number of times. In order to address this issue, our model includes techniques for efficiently managing block erasures, such as wear-leveling and erase-unit reclamation [10].

Alloy provides completely automatic (but bounded) analysis of a model, given a property and a scope for the size of each domain in the model. In order to verify the correctness of the filesystem design, we used a simplified version of the POSIX standard as a reference model. Then, we checked trace inclusion of the concrete flash filesystem against the reference model using backwards simulation [24]. To our knowledge, this is the first fully automatic analysis of a design for a flash filesystem.

The paper presents the POSIX reference model (Section 2), a model of the underlying flash hardware (3), and then a design of a flash filesystem that uses the flash hardware and is intended to conform to the reference model (4). Unlike the first two models, this design does not formalize existing descriptions, but it incorporates a variety of mechanisms that have appeared in the literature [10,15]. Subsequent sections describe the analysis that was performed (5), relate our approach to others (6), and discuss the challenges faced during this project (7) and plans for future work (8).

2 Abstract POSIX Filesystem

POSIX (Portable Operating System Interface) [21] is an international standard that specifies the function signatures and expected behaviours of a set of filesystem operations. The widespread adoption of POSIX by many popular operating systems, such as UNIX and Mac OS X, makes it an attractive choice as a reference model for a flash filesystem. A specification for a UNIX filesystem was formalized in the Z notation [20] by Morgan et al. [18] and served as a starting point for our work.

Alloy [16] is a declarative modeling language based on first-order logic with transitive closure. The origin of Alloy is rooted in Z, drawing on the latter's simple and intuitive semantics that is well suited for object and data modeling. For example, a file with an array of data elements can be declared as follows[1]:

```
sig File { contents : seq Data }
sig Data {}
```

Then, an entire filesystem may be viewed as a container for a relation that maps each file identifier to zero or one file[2]:

```
sig AbsFsys { fileMap : FID -> lone File }
sig FID {}   // File identifier
```

[1] In Alloy, the keyword `sig S` declares a set of atoms of type `S`, and `seq T` constructs a sequence whose elements are of type `T`.

[2] The keyword `lone` imposes a cardinality constraint of "less than or equal to one".

The basic operations of the filesystem—reading from and writing to a file—are modeled using functions and predicates. The read operation, as defined by POSIX, takes three arguments—a file identifier, an offset, and a size—and returns a data sequence of the specified size, starting at the offset within the file:

```
fun readAbs[fsys: AbsFsys, fid: FID, offset, size: Int]: seq Data {
   let file = fsys.fileMap[fid] |
       // subseq[m,n] returns a subsequence between m and n, inclusively
       (file.contents).subseq[offset, offset + size - 1]
}
```

Like Z, Alloy does not support the notion of an implicit global state. Thus, `readAbs` explicitly takes an instance of `AbsFsys` as an argument.

The write operation takes a file identifier, an offset, a size, and a buffer containing the input data, and writes a data sequence of the specified size from the buffer into the file, starting at the offset[3]:

```
pred writeAbs[fsys, fsys': AbsFsys, fid: FID, buffer: seq Data,
              offset, size: Int] {
   let buffer' = buffer.subseq[0, size - 1],
       file = fsys.fileMap[fid], file' = fsys'.fileMap[fid] {
          (#buffer' = 0) => file' = file
          (#buffer' != 0) =>
             file'.contents =
                (zeros[offset] ++ file.contents) ++ shift[buffer', offset]
          promote[fsys, fsys', file, file', fid]
       }
}
```

Sequences are represented, as in Z, by functions from integers to values. The expression `zeros[n]` gives a sequence of zeros of length `n`, and is used for padding; `shift[s,i]` gives a function like the sequence `s` but with the index of each element shifted by `i`. Unlike `readAbs`, `writeAbs` is expressed as a predicate (rather than a function) because it may modify the state of the filesystem.

There are three distinct cases to consider in this operation. First, if the input buffer is empty, `writeAbs` does not modify the contents of the file. If the offset is located within the file, `writeAbs` overrides the existing data in the overlapping positions with the input data. Lastly, if the offset is greater than the file size, `writeAbs` fills the gap between the end of the file and the offset with zeros.

Changes in the state of the filesystem are modeled using an explicit constraint between a pair of pre- and post- states (`fsys` and `fsys'`). It is also necessary to ensure that the operation does not affect any other files. This style of modeling changes in system state is called *promotion* [24]. The following predicate "promotes" a change in the contents of a file to a change in the entire filesystem:

```
pred promote[fsys, fsys': AbsFsys, file, file': File, fid: FID] {
   file = fsys.fileMap[fid]
   fsys'.fileMap = fsys.fileMap ++ (fid -> file')
}
```

[3] The cardinality operator `#` returns the length of a sequence, and `++` is the operator for relational override.

3 NAND Flash Memory

Two types of flash memory are currently in widespread use: NOR and NAND. Although NOR allows random access and is easier to program, the higher density and performance of NAND makes the latter more suitable as a storage device. Our model of the flash hardware is based on Open NAND Flash Interface (ONFi), an industry-wide standard for the specification of NAND flash memory [14]. However, it is important to note that the focus of our work is on the design of a filesystem, not a flash device. Therefore, our hardware model includes only the minimum amount of detail necessary for modeling basic flash operations, erase-unit reclamation, and fault tolerance.

3.1 Memory Hierarchy

The smallest unit for reading or programming flash memory is called a *page*; it consists of a fixed number of data elements[4]:

```
sig Page { data : seq Data } { #data = PAGE_SIZE }
```

In addition, each page is associated with one of four status constants:

```
abstract sig PageStatus {}
one sig Free,                          // Erased and ready to be programmed
        Allocated,                     // Allocated for a file write operation
        Valid,                         // Contains valid data in a file
        Invalid extends PageStatus {}  // Contains obsolete data
```

A *block*, also called an *erase-unit*, contains an array of pages and is the smallest unit for erase operations. A *logical unit* (LUN) is the minimum independent entity that receives and executes a flash command[5]:

```
sig Block { pages : seq Page } { #pages = BLOCK_SIZE }
sig LUN { blocks : seq Block } { #blocks = LUN_SIZE }
```

Two forms of addresses are used during flash operations: the *row address* and the *column address*. A row address is used to access a particular page:

```
sig RowAddr { lunIndex, blockIndex, pageIndex : Int }
```

A column address, simply of type `Int`, identifies the position of a data element within a page.

Finally, a flash *device* is the top-level component that directly communicates with the host filesystem:

[4] A formula `F` in `sig A {...}{F}` is a constraint that applies to every atom of type `A`.

[5] ONFi defines another level of hierarchy—called *targets*—above LUNs. To simplify our analysis, we abstract away this detail from our model.

```
sig Device {
   luns : seq LUN,
   pageStatusMap : RowAddr -> one PageStatus,
   eraseCountMap : RowAddr -> one Int,
   reserveBlock : RowAddr
}{ #luns = DEVICE_SIZE }
```

The three fields `pageStatusMap`, `eraseCountMap`, and `reserveBlock` are auxiliary data structures used for erase-unit reclamation and fault tolerance[6]. The `pageStatusMap` associates each page in the device with its current status. The `eraseCountMap` associates each block with the number of times it has been erased; erase counts play a crucial role in wear-leveling. Lastly, `reserveBlock` holds the address of a block that temporarily stores valid pages during erase-unit reclamation. The usage of these data structures is further discussed in Section 4.

3.2 Flash API Functions

During a file operation, the host filesystem may make one or more calls to three flash API functions: read, program, and erase. Due to limited space, we present only the interface declarations of these operations:

```
// Reads data from the page at "rowAddr", starting at the index "colAddr"
fun fRead[d: Device, colAddr: Int, rowAddr: RowAddr] : seq Data { ... }

// Programs (i.e. writes) "newData" into the page at "rowAddr", starting at
// the index "colAddr", and sets the status of the page to "Allocated"
pred fProgram[d,d': Device, colAddr: Int, rowAddr: RowAddr,
              newData: seq Data] { ... }

// Erases the entire block that contains the page at "rowAddr", increments
// its erase count, and sets the status of every page within the block to "Free"
pred fErase[d,d': Device, rowAddr: RowAddr] { ... }
```

Note that `fProgram` and `fErase` are expressed as constraints between two device states (`d` and `d'`) since these operations may modify the state of the device.

4 Flash Filesystem

Given the model for the underlying hardware, we now describe a concrete filesystem that communicates with the flash device to perform file operations. This concrete model is not based on one particular flash filesystem; rather, our design incorporates a variety of mechanisms that have appeared in literature. Namely, we adopted the techniques for wear-leveling and erase-unit reclamation from Gal and Toledo's survey paper on flash memory algorithms [10]. The division of the write operation into separate phases and the mechanism for power-loss recovery

[6] ONFi does not explicitly mention these data structures. On an actual device, they would be scattered across the flash memory using sophisticated techniques, but this detail is not suitable for the level of modeling abstraction in this work.

were modeled after the Intel Flash File System Specification [15]. It is important to note that in our modeling task, we were primarily concerned with the correctness of the design, not its performance. Thus, when multiple techniques were available, we adopted the alternative that we considered to be the simplest.

A file, represented by an `Inode`, consists of a list of virtual blocks (`VBlock`), each of which points to a particular page on the flash device:

```
sig Inode { blockList : seq VBlock }
sig VBlock {}
```

Like its abstract counterpart, the concrete filesystem contains a relation that maps each file identifier to at most one inode. In addition, the filesystem contains a bijective map from a virtual block to a row address:

```
sig ConcFsys {
   inodeMap : FID -> lone Inode,
   blockMap : VBlock one -> one RowAddr
}
```

Rather than being a fixed map, `blockMap` is dynamically updated during write operations, and plays a crucial role in wear-leveling, as discussed below.

4.1 Concrete Operations

The two basic file operations that we describe here—read and write—are substantially more complex than their counterparts in the abstract filesystem. A concrete operation involves multiple calls to the flash API functions, since an inode consists of a number of fixed-size pages. Due to limited space, we focus on the most distinctive aspects of the operations.

Concrete Read. Like `readAbs`, the `readConc` operation (Fig. 1) accepts three arguments—`fid`, `offset`, and `size`. In addition, `readConc` requires a `ConcFsys` and a `Device`, which together represent a particular state of the filesystem. Note that unlike `readAbs`, `readConc` is a predicate (not a function) that constrains `buffer` to be the result of the operation[7].

The core part of `readConc` (shown in Fig. 1) involves reading each of the virtual blocks in the inode and storing the output into a single, contiguous buffer. Prior to line 4, `blocksToRead` is constrained to be a sequence of virtual blocks to be read, based on `offset` and `size`. For each index `i` in this sequence, `readConc` retrieves the address of the page to which the virtual block at `i` is mapped (line 6), invokes `fRead` (line 9), and stores the page data into a buffer slot between two indices, `from` and `to`. After line 9, `buffer` contains all of the data from `blocksToRead` in the order that they appear within the inode.

[7] Sometimes, it is more natural to describe a result implicitly rather than to construct it explicitly using a function with the formula as the body of a set comprehension.

```
 1: pred readConc[fsys: ConcFsys, d: Device, fid: FID, offset, size: Int,
 2:               buffer: seq Data] {
 3:   ...
 4:   all idx : blocksToRead.inds |   // "inds" returns the set of all indices
 5:     let vblock = blocksToRead[idx],
 6:         rowAddr = fsys.blockMap[vblock],
 7:         from = PAGE_SIZE * idx, to = from + PAGE_SIZE - 1 |
 8:           // Read a flash page and store data into correct buffer slot
 9:           buffer.subseq[from,to] = fRead[d, 0, rowAddr]
10:   ...
11: } // 90 LOC in total, including comments
```

```
 1: pred writeConc[fsys, fsys': ConcFsys, d, d': Device, fid: FID,
 2:                buffer: seq Data, offset, size: Int] {
 3:   ...
 4:   some stateSeq : seq TranscState, interDev : Device {
 5:     // Phase 1: Program pages
 6:     stateSeqConds[d, interDev, stateSeq, numPagesToProgram]
 7:     all idx : stateSeq.butlast.inds {
 8:       let inode = fsys.inodeMap[fid],
 9:           from = PAGE_SIZE * idx, to = from + PAGE_SIZE - 1,
10:           dataFragment = buffer.subseq[from, to],
11:           vblock = inode.blockList[startBlkIndex + idx],
12:           rowAddr = fsys.blockMap[vblock]
13:           preState = stateSeq[idx], postState = stateSeq[idx + 1] |
14:             // Program one page worth of data into the flash
15:             programVBlock[preState, postState, rowAddr, dataFragment]
16:     }
17:     // Phase 2: Invalidate/validate old/new pages
18:     updatePageStatuses[interDev, d']
19:     // Update virtual-block-to-page mapping and the list of inode blocks
20:     updateFsysInfo[fsys, fsys', fid, stateSeq.last]
21:   }
22:   ...
23: } // 549 LOC in total, including comments
```

Fig. 1. Concrete Operations

Wear-Leveling and Erase-Unit Reclamation. Unlike sectors in a traditional disk-based filesystem, flash pages must be completely erased before they can be rewritten. One major limitation of flash memory is that each block can be erased only a finite number of times. Thus, a *wear-leveling* technique that distributes erasures evenly across flash memory is essential in any flash filesystem.

Using an example, let us illustrate the standard wear-leveling technique [10] that is adopted by most flash filesystem, including our design. Suppose an inode `n` consists of a list of virtual blocks, one of which (`vblk`) is mapped to a physical flash page `p1`. A client sends a request to the filesystem to overwrite the existing data, including `vblk`, in the inode. A simple approach would be to erase the physical flash block `fblk` that contains `p1` and then program new data into `p1`. However, if operations involving `n` were frequent, then `fblk` would wear out much more quickly than others. Thus, rather than erasing `p1`, we instead program the new data into a free, available page `p2`. In addition, we mark the data in `p1` as

obsolete by modifying the page status to `Invalid`. Lastly, we update `blockMap` in `ConcFsys` to indicate that `vblk` is now mapped to `p2`.

Over time, the flash device accumulates obsolete data and eventually runs out of free pages. In order to free up space for new program operations, the filesystem carries out a procedure called *erase-unit reclamation*. A reclamation procedure involves the following steps:

1. Search for all blocks that contain obsolete data. Among these blocks, select the one with the lowest erase count by checking `eraseCountMap` in `Device`.
2. Relocate all valid pages in the selected block. This block (call it `dirtyBlock`) may still contain one or more pages that hold valid data. For the purpose of relocation, the filesystem keeps one completely erased block as a spare (`reserveBlock` in `Device`). Each valid page is relocated from `dirtyBlock` to the corresponding position in `reserveBlock`.
3. Erase `dirtyBlock` using the `fErase` command. This block becomes the new reserve block for the filesystem.

After Step 3, all pages within the old `reserveBlock` that do not hold the relocated data from `dirtyBlock` are free and available for programming.

Concrete Write. The `writeConc` operation (Fig. 1) is expressed as a constraint between two pairs of `ConcFsys` and `Device` atoms. The pair (`fsys, d`) represents the state of the filesystem at the beginning of the operation, and (`fsys', d'`) represents the state at the end. At the filesystem client level, `writeConc` is a single-step transition between `fsys` and `fsys'`, modifying the filesystem in the following ways: 1) If `writeConc` involves overwriting existing data in the inode, then it updates `fsys.blockMap` with a new virtual-block-to-page mapping, and 2) if `writeConc` involves writing data beyond the current end of the inode, then it appends one or more virtual blocks to `inode.blockList`.

At the flash device level, `writeConc` makes a sequence of calls to the flash command `fProgram`, depending on the size of the input data and `PAGE_SIZE`. In order to model the flash operations closely to ONFi, we explicitly introduce a sequence of intermediate device states between `d` and `d'`; each pair of adjacent states in this sequence corresponds to a pair of pre- and post- states that are passed as arguments to `fProgram`. After each step along the sequence, `writeConc` maintains information about allocated-to-obsolete-page pairs and a list of new pages to be added to the inode; this auxiliary information is used to update `fsys.blockMap` and `inode.blockList` at the end of the operation. The signature `TranscState` encapsulates all of the stateful information:

```
sig TranscState {
   dev : Device,                // Current device state
   allocToObsoletePagePairs : RowAddr -> lone RowAddr, // New-old page pairs
   newPageList : seq RowAddr    // List of new pages to be added to inode
}
```

Then, we can model a sequence of flash-level transitions using a sequence of `TranscState`'s, with additional constraints as follows:

```
pred stateSeqConds[init, final: Device, stateSeq: seq TranscState, length: Int]{
   stateSeq.first.dev = init    // Beginning of trace is initial device state
   stateSeq.last.dev = final    // End of trace is final device state
   #stateSeq = length + 1       // Constrain the length of sequence
   no stateSeq.first.allocToObsoletePagePairs  // Initially empty pairs
   no stateSeq.first.newPageList               // Initially empty list
}
```

Fig. 1 shows a core snippet from a simplified version of the full `writeConc` model. Based on the Intel specification (Section 2.5) [15], we divide the operation into two distinct phases. Phase 1 (lines 6-16) involves partitioning the input buffer into fixed-size fragments and programming them into the flash memory. For each `i`, which corresponds to the $\texttt{i}^{th}$ transition in `stateSeq`, `writeConc` extracts a data fragment of length `PAGE_SIZE` from the buffer (lines 9-10). Next, `writeConc` retrieves the row address of the virtual block that will be overwritten with this fragment (lines 11-12). Finally, in line 15, the predicate `programVBlock` programs the data fragment into the virtual block (which will be mapped to a new flash page) by executing `fProgram` and adds a (new, old) row address pair to `preState.allocToObsoletePagePairs`.

If the expression `(startBlkIndex + i)` evaluates to an index beyond the end of `inode.blockList`, then both `vblock` and `rowAddr` will be empty expressions (lines 11 and 12). The predicate `programVBlock` handles this case by appending a page to `preState.newPageList`. In addition, if the device is out of free pages, `programVBlock` performs erase-reclamation before programming a page.

In Phase 2, we invalidate the pages that contain obsolete data and then validate all of the pages that were allocated during Phase 1; for simplicity, we present this phase as being carried out inside the predicate `updatePageStatuses` (line 18). The quantified variable `interDev`, introduced in line 4, acts as an intermediate device state that joins the two phases together.

Lastly, after all of the flash-level transitions have been completed, `writeConc` updates `fsys.blockMap` and `inode.blockList` using the information accumulated up to the last `TranscState` in the state sequence (line 20).

4.2 Fault Tolerance Mechanism

Over the course of its lifetime, a flash device is susceptible to a variety of unexpected hardware failures. Therefore, one crucial aspect of designing a flash filesystem is its robustness in recovering from such failures. After recovery, the filesystem must be either in a state as if an operation has never begun, or in a state where the operation has been successfully completed. In this work, we modeled one particular type of fault-tolerance mechanism—recovery from power loss in the middle of a write operation. Our model is based on the mechanism that is described in the Intel specification (Section 2.5) [15].

A power failure can occur during either Phase 1 or Phase 2 of the write operation. We give a high-level description of the fault-tolerance mechanism in these two distinct cases:

```
pred alpha[asys: AbsFsys, csys: ConcFsys, d: Device] {
  all fid : FID |
  let file = asys.fileMap[fid], inode = csys.inodeMap[fid],
      vblocks = inode.blockList {
         #file.contents = #vblocks * PAGE_SIZE
         (all i : vblocks.inds |
          let vblock = vblocks[i],
              from = i * PAGE_SIZE, to = from + PAGE_SIZE - 1,
              absDataFrag = file.contents.subseq[from,to],
              concDataFrag = findPageData[vblock,csys,d] |
              absDataFrag = concDataFrag)
      }
}

assert WriteRefinement {
  all csys, csys': ConcFsys, asys, asys': AbsFsys, d, d': Device,
      fid: FID, buffer: seq Data, offset,size : Int |
        concInvariant[csys, d] and
        writeConc[csys, csys', d, d', fid, buffer, offset, size] and
        alpha[asys, csys, d] and
        alpha[asys', csys', d']
        => writeAbs[asys, asys', fid, buffer, offset, size]
}
```

Fig. 2. Abstract Relation and Refinement Property for Write

Phase 1: At the point of the failure, one or more pages have been programmed and their statuses have been modified to `Allocated`. To recover from this failure, we set the status of every allocated page to `Invalid`. After recovery, the device contains extra invalid pages, but to a filesystem client, the inode appears to have the same data as it did at the beginning of the operation.

Phase 2: To recover from power loss during this phase, we first invalidate every page `p1` that is paired with an allocated page `p2` (i.e. `p2` is the replacement for `p1`). Then, we validate every such `p2` by setting its status to `Valid`. In essence, the recovery process is here equivalent to completing the rest of Phase 2 that was interrupted by the power failure. At the end of the recovery, the inode contains the input data as expected by the caller of `writeConc`.

5 Analysis

Given the models for the abstract and concrete filesystems, we used the Alloy Analyzer to check refinement properties for read and write operations. First, we defined an abstraction relation `alpha` that maps a concrete state (represented by a pair of `ConcFsys` and `Device` atoms) to an abstract state (represented by an `AbsFsys` atom). The relation is expressed as a predicate (Fig. 2) that states that for every file in the abstract filesystem, the concrete state includes an inode with

a correctly ordered sequence of virtual blocks containing the same data elements as in the abstract file[8].

The assertion `WriteRefinement` posits a backwards simulation for the write operation[9]. We performed backwards (rather than forwards) simulation since `alpha` maps a concrete state "upwards" to an abstract state. The predicate `concInvariant` defines a valid state in the concrete filesystem—for example, that every free page in the flash device must be completely erased—and its preservation is checked independently. When the Alloy Analyzer finds a scenario that violates the assertion within a specified scope, it graphically displays the counterexample using its built-in visualizer. In the final version of the model, the analyzer returned no counterexamples for the assertion. We used a scope of 5 for every signature in the model, with 6 flash pages, each of which was constrained to contain 4 data elements. The total size of the filesystem was therefore 24 data elements. The property was checked on a 3.6 GHz Pentium 4 machine with 3GB RAM in approximately 8 hours.

Even though the size of the filesystem that we checked is too small to represent a realistic system, we were able to find over 20 non-trivial bugs over the entire course of our design process. These bugs were removed from the model throughout the various iterations of our modeling task. In a typical filesystem, many types of errors occur in "boundary cases", which involve only a small number of components (i.e. pages, blocks, etc.). For example, consider the model for the `readConc` operation in Fig. 1. As currently shown, this operation is buggy in the following two ways: (1) If `offset` is not a multiple of `PAGE_SIZE`, then the length of the first slot in the output buffer must be less than `PAGE_SIZE`, and similarly, (2) if the expression `(offset + size)` is not a multiple of `PAGE_SIZE`, then the length of the last slot in the buffer must also be adjusted accordingly. An instance of a filesystem state with two pages is sufficient to generate a counterexample that demonstrates both of these bugs; increasing the scope to a higher value would not reveal any useful information about bugs of a similar nature.

6 Related Work

Our work is a contribution to the second pilot project in the Verified Software Repository (VSR) [3]. The idea of verifying a flash filesystem as a mini-challenge was suggested by Joshi et al. [17], and several groups are now actively working on this project [5,7,9,13].

Filesystems were an early target for case studies in formal methods. As a historically significant example, Morgan and Sufrin first formalized a specification for a UNIX filesystem in Z [18]. Freitas and his colleagues refined an abstract POSIX filesystem to a concrete implementation and proved the refinement relation using Z/Eves [8]. Similarly, Arkoudas et al. proved a refinement relation between an abstract filesystem and a disk-based implementation in the Athena

[8] For simplicity, we restrict the size of every abstract file as shown in Fig. 2 to be a multiple of `PAGE_SIZE`. The complete version on the web is free of this restriction.

[9] We can obtain `ReadRefinement` by replacing the `write` predicates with `read`.

theorem prover [2]. In comparison to previous two works, which employ theorem proving, the analysis in Alloy is fully automatic, but it guarantees the correctness of the refinement relation only up to a finite bound.

Butterfield et al. formalized NAND flash memory in Z [6], following the ONFi specification, which formed a basis for our hardware model as well. Ferreira and their colleagues also formalized the ONFi specification and a POSIX filesystem in VDM++ [7]. They performed the analysis of the filesystem using the HOL theorem prover [12] and Alloy. They used the Alloy Analyzer primarily for finding a counterexample to proof obligations that could not be automatically discharged by HOL, whereas we used the analyzer to perform the analysis in its entirety.

Groce et al. performed randomized testing on a POSIX filesystem implementation that is based on NAND flash memory [13]. Yang et al. used model checking to find errors in existing filesystem implementations [25]. Although their work is not flash-specific, the nature of their analysis is similar to ours; they deliberately scaled down the size of the filesystem for increased tractability of analysis but were still able to find numerous bugs, many of which were due to complex interactions among a small number of components.

7 Discussion

In this project, we have shown that we were able use Alloy to successfully model and analyze the types of complexity that arise in a flash filesystem design. We believe that the scope we used in the final analysis was sufficient to ensure that the refinement relation is sound, but we currently cannot justify our intuition rigorously. It is also possible, as it would be even if theorem proving were used, that the concrete model harbours unintentional overconstraints that slipped through our analysis unnoticed.

Our experience has raised a number of interesting questions about the Alloy language and the analyzer. First of all, due to the declarative nature of the language, modeling multi-step operations in Alloy is not always straightforward. Although the language is expressive enough for describing such operations, writing certain types of control constructs (such as loops) in Alloy can be cumbersome, since it does not support a built-in notion of an implicit global state. In particular, in order to model changes to the device after each call to `fProgram` in `writeConc`, we explicitly introduced a sequence of state atoms and imposed a constraint between each pair of adjacent states. A language such as ASM [4]—with the notion of an implicit global state—may be more suitable for this particular aspect of the filesystem model. We are currently investigating an extension to Alloy that will provide the user with control constructs, while maintaining the declarative power of the language.

Our filesystem model, as one of the largest case studies that we have done to date, has pushed the boundary of Alloy's scalability. The analyzer uses as its backend a relational model finder called Kodkod [23] that translates an Alloy model to a CNF formula, which can then be handled by powerful SAT solvers. Although checking the refinement relation in our latest model took several hours

to complete, the analyzer is fully automatic, and so we were able to leave the analyzer running unattended overnight. On the other hand, due to the exponential nature of SAT, the duration of the analysis can grow rapidly as the scope is incremented or as additional layers of complexity are added to the model. Kodkod already employs a variety of techniques to reduce the size of a SAT problem, such as symmetry breaking and sharing detection [23]; we are looking into further opportunities for an improved scalability by leveraging available techniques (e.g. additional decision procedures [11]).

Another useful feature of the Alloy Analyzer is the extraction of an unsatisfiable core [22], which highlights top-level constraints in a model that are used to establish the correctness of an assertion. In some cases, an overconstraint may cause an assertion to be vacuously true; the user can usually tell when this has happened by noticing that formulas that were expected to be highlighted as part of the core were not. The unsatisfiable core facility was very useful in this project, and did indeed expose overconstraints on several occasions. However, the granularity of the core can sometimes be too coarse to be useful to the user. In particular, a top-level formula that is existentially quantified over a conjunction of sub-constraints is treated as a single constraint in the core; this grouping might suppress potentially useful information. We are currently implementing a mechanism that will overcome this problem and extract a finer-grained unsatisfiable core.

8 Future Work

While formalizing and analyzing a design model are useful exercises on their own, one interesting question is whether the usage of the model can be extended beyond the design into the implementation and testing phases. We are looking into possible uses of the flash filesystem model. For example, by mapping our model to an existing flash filesystem implementation (such as YAFFS [1]), we can leverage the power of the Alloy Analyzer as a model finder to automatically generate a large set of test cases. Other research questions that we are planning to explore include simulation, model-based diagnosis, and code generation.

The current functionality of our filesystem is rather limited. For future work, we plan to include a larger set of POSIX file operations, such as `creat`, `open`, and `close`, and the support for directories. We also plan to model recovery mechanisms for other types of failures (besides power loss), such as bad blocks and bit corruption.

Complete versions of all Alloy models that appear in this paper are available at `http://sdg.csail.mit.edu/projects/flash`.

Acknowledgements. We are grateful to Felix Chang, Greg Dennis, Vijay Ganesh, Derek Rayside, Sivan Toledo, and Emina Torlak for helpful discussions and feedback. This research was supported by the National Science Foundation under Grant Nos. 0541183 and 0438897, and by the Nokia Corporation as part of a collaboration between Nokia Research and MIT's Computer Science and Artificial Intelligence Lab.

References

1. Aleph One. YAFFS: A flash file system for embedded use, http://www.yaffs.net
2. Arkoudas, K., Zee, K., Kuncak, V., Rinard, M.: On verifying a file system implementation. In: 6th ICFEM, pp. 373–390 (2004)
3. Bicarregui, J., Hoare, C.A.R., Woodcock, J.: The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing 18, 143–151 (2006)
4. Borger, E., Start, R.F.: Abstract State Machines: A method for high-level system design and analysis. Springer, New York (2003)
5. Butler, M., Damchoom, K., Abrial, J.-R.: Some filestore developments with Event-B and Rodin. In: Verifiable File Store Mini-Challenge Workshop, co-located with the 9th ICFEM (2007)
6. Butterfield, A., Woodcock, J.: Formalizing flash memory: First steps. In: 12th ICECCS, pp. 251–260 (2007)
7. Ferreira, M.A., Silva, S.S.: J. N. Oliveira Verifying Intel flash file system core specification. In: 4th VDM-Overture Workshop, FM 2008 (2008)
8. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: an experiment in the verified software repository. In: 12th ICECCS, pp. 3–14 (2007)
9. Freitas, L., Woodcock, J., Butterfield, A.: POSIX and the Verification Grand Challenge: a roadmap. In: 13th ICECCS, pp. 153–162 (2008)
10. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Computing Surveys 37, 138–163 (2005)
11. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
12. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York (1993)
13. Groce, A., Holzmann, G.J., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: 29th ICSE, pp. 621–631 (2007)
14. Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0. ONFi Workgroup (2006), http://www.onfi.org
15. Intel. Flash File System Core Reference Guide. Technical Report 304436001. Intel Corporation (2004)
16. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
17. Joshi, R., Holzmann, G.J.: A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005)
18. Morgan, C., Sufrin, B.: Specification of the UNIX filing system. IEEE Transactions on Software Engineering 10, 128–142 (1984)
19. Reeves, G., Neilson, T.: The Mars Rover Spirit FLASH Anomaly. In: IEEE Aerospace Conference (2005)
20. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Englewood Cliffs (1998)
21. The Open Group. The POSIX 1003.1, 2003 Edition Specification, http://www.opengroup.org/certification/idx/posix.html
22. Torlak, E., Chang, F.S.-H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008)

23. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: 13th TACAS, pp. 632–647 (2007)
24. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, NJ (1996)
25. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. In: 6th OSDI, pp. 273–288 (2004)

Unit Testing of Z Specifications

Mark Utting[1] and Petra Malik[2]

[1] Department of Computer Science, The University of Waikato, NZ
marku@cs.waikato.ac.nz,
[2] Faculty of Engineering, Victoria University of Wellington, NZ
petra.malik@mcs.vuw.ac.nz

Abstract. We propose a simple framework for validation unit testing of Z specifications, and illustrate this framework by testing the first few levels of a POSIX specification. The tests are written in standard Z, and are executable by the CZT animator, ZLive.

1 Introduction

In [Hoa03], Hoare proposes a grand challenge for computer science—a verifying compiler—and inspired researchers from all over the world to work jointly towards this ambitious goal. A key part of the grand challenge is a series of case studies [BHW06], which provide examples of specified and verified code and can be used as benchmarks to exercise and test current and future verification tools.

The Mondex case study was tackled as a first pilot case study in 2006. Several teams using a variety of techniques and tools worked on a fully automated proof of the Mondex smart-card banking application. A verifiable filesystem has been proposed [JH07] as another mini challenge. An initial small subset of POSIX has been chosen [FFW07] and participants of the ABZ 2008 conference were challenged to contribute to this project.

Lots of work and effort is typically put into either deriving or verifying a correct lower level specification or implementation from an abstract specification. We argue that before those activities, the abstract specification should be carefully validated against the requirements. While there is hope that verification can be mostly automated, validation remains a task for human designers.

This paper proposes a simple approach to validate Z specifications by writing positive and negative unit tests for each schema within the specification. These tests give the designer more confidence that their specification reflects their intentions, allows regression testing after each modification of the specification, and can help the unfamiliar reader of the specification to understand the specification more quickly and easily. The framework has been used to design tests for the first few levels of a refactored version of Morgan and Suffrin's Z specification of the POSIX file system [MS84]. The tests were executed and checked by the CZT animator ZLive.

The structure of this paper is as follows. In Section 2, we present our testing framework. Section 3 gives an overview of the ZLive animator, which is used to evaluate the tests. Section 4 describes the refactored POSIX specification.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 309–322, 2008.

In Section 5, the tests for the data system are explained and Section 6 shows how they can be promoted to test the storage system. Finally, Section 7 gives conclusions.

The main contributions of the paper are: the unit testing framework for validating Z specifications, a simple style for promoting tests from one level to another, the refactored and standard-compliant POSIX specification, the example unit tests we have developed for it, and the use of the ZLive animator to execute those tests.

2 How to Test a Z Specification

This section introduces a simple framework for expressing unit tests of a Z specification. We assume that the specification is written in the common Z *sequential* style, with each section containing a state schema, an initialisation schema, several operation schemas, as well as various auxiliary definitions and schemas.

The first question that must be asked is why we do not use an existing testing framework for Z, such as the Test Template Framework (TTF) [Sto93, CS94] or Heirons' Z-to-FSM approach [Hie97]. The main difference is that they all aim at generating tests *from* a Z specification, in order to test some implementation of that specification. But in this paper, we want to design some tests that actually test the Z specification itself. So the goal of those approaches is *verification* (of some implementation), whereas our goal is *validation* (of the Z specification). This results in quite a different style of testing.

An important philosophical difference is that when our aim is the *validation* of a Z specification, the correctness of our tests must be ensured by some external means (not just the specification itself), to avoid circular reasoning. That is, we cannot generate tests from the specification, then test them against that same specification, and expect to find any errors. There needs to be some independence between the tests and the system being tested. In this paper we avoid such circular reasoning by designing our tests manually, based on the informal English description of the POSIX operations. So the oracle for the correctness of the tests is our human understanding of the POSIX requirements.

A practical difference is that when testing an implementation (for verification testing), we can control its inputs but not its outputs, whereas when testing a Z specification (for validation purposes) we can ask arbitrary questions about inputs or outputs. So when validating a Z specification, we can use a richer variety of tests. In this paper, by a *test* of a Z schema *Op* we mean a boolean property of some finite/small subset of S that can (in principle) be enumerated in a reasonable time. For example, we may be able to test *Op* by instantiating it with a specific output value and asking which input values could generate that output value – such 'tests' are not possible on black-box implementations.

Another difference is that when validating a specification, we want to design both positive tests, which test that a given behaviour is allowed by the specification, and negative tests, which test that a given behaviour is not allowed

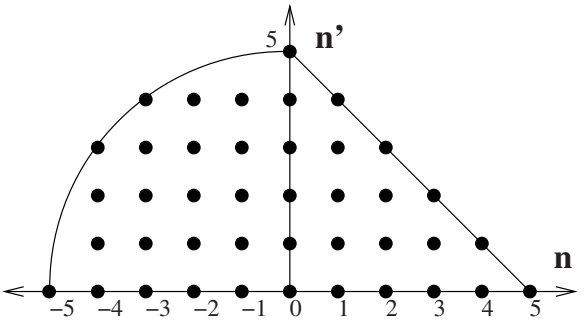


Fig. 1. A graph of the solutions to the Wedge schema

by the specification. In contrast, when testing whether an implementation is a refinement of a specification, the implementation is usually free to add behaviour outside the precondition of the specification and we can therefore use only positive tests.[1]

To illustrate our testing framework, we shall use a simple example specification given by the following specification, which defines the shape shown in Figure 1. Our tests will be written using standard Z conjecture paragraphs, so that they can be checked by theorem provers or by the ZLive animator. For each section S, we write the unit tests for that section within a separate section (typically called *STests*) that has S as a parent. This clearly identifies the tests and separates them from the rest of the specification, so that tools that operate on the main specification can ignore the unit tests.

section *wedge* **parents** *standard_toolkit*

$$State == [n : -10 \,..\, 10]$$
$$Wedge == [\Delta State \mid n * n + n' * n' \leq 25;\ n + n' \leq 5;\ 0 \leq n']$$

2.1 Positive Tests

The first kind of testing we want to do is positive tests to check that an operation has a desired input-output behaviour. When specifying tests, it is usual to specify the expected output values as well as the input values, so we write all the inputs and output values as a Z binding and use a membership test. Here are two examples that test the non-deterministic output behaviour of Wedge when the input n is zero.

section *wedgeTest* **parents** *wedge*

[1] Note that a test of exceptional behaviour that is formalised in the specification is a positive test.

$$\text{Wedge0Gives0} \quad \vdash? \ \langle\!|\ n == 0, n' == 0\ |\!\rangle \in \mathit{Wedge}$$

$$\text{Wedge0Gives5} \quad \vdash? \ \langle\!|\ n == 0, n' == 5\ |\!\rangle \in \mathit{Wedge}$$

Note that since 2007, the Z standard requires conjectures to be written within a LaTeX *theorem* environment, and allows each conjecture to be given a name. Our naming convention for tests is that the name of a test should start with the name of the operation being tested, followed by some phrase that expresses the essential property of the test.

An alternative style of writing a suite of positive tests is to name each test tuple, group the tuples into a set, and then test them using a single subset conjecture.

$$\mathit{Wedge0Gives0} == \langle\!|\ n == 0, n' == 0\ |\!\rangle$$
$$\mathit{Wedge0Gives5} == \langle\!|\ n == 0, n' == 5\ |\!\rangle$$

$$\text{WedgePos} \quad \vdash? \ \{\mathit{Wedge0Gives0}, \mathit{Wedge0Gives5}\} \subseteq \mathit{Wedge}$$

2.2 Negative Tests

It is also useful to perform *negative* tests to validate an operation. For example, we may want to check that an input value is outside the precondition of the operation, or check that a certain output can never be produced by the operation. The idea is to validate the specification by showing that its behaviour is squeezed in between the set of positive tests and the set of negative tests. The more positive and negative tests that we design, the more sure we can be that we have specified the desired behaviour, rather than allowing too many or too few behaviours.

We can write negative tests using a negated membership test ($\mathit{test} \notin \mathit{Op}$) or, equivalently, we can test that the tuple is a member of the negated schema ($\mathit{test} \in \neg\ \mathit{Op}$). Our conjecture naming convention is the same as for positive tests, but the phrase after the operation name usually contains a negative word such as *not* or *cannot* to emphasize that it is a negative test.

$$\text{Wedge0NotNeg} \quad \vdash? \ \langle\!|\ n == 0, n' == -1\ |\!\rangle \notin \mathit{Wedge}$$

$$\text{Wedge0Not6} \quad \vdash? \ \langle\!|\ n == 0, n' == 6\ |\!\rangle \in \neg\ \mathit{Wedge}$$

We can also test just the precondition of the operation.

$$\text{WedgePreNot6} \quad \vdash? \ \langle\!|\ n == 6\ |\!\rangle \notin \text{pre}\ \mathit{Wedge}$$

As we did for positive tests, we may also combine several negative tests into a group. In this case, our conjecture may be written as $\mathit{tests} \subseteq \neg\ \mathit{Op}$, or equivalently we may check that the intersection of the negative test suite and the operation is empty, $\mathit{tests} \land \mathit{Op} = \varnothing$. The latter style is often more convenient, since it allows us to write *tests* using schema notation, and to omit variables

that we are not interested in (because the schema conjunction will expand the type of *tests* to match the type of *Op*). For example, the following two negative test suites check that 6 can never be an input for *Wedge*, and that 6 can never be an output of *Wedge*.

$$\text{Wedge6Not} \vdash? ([n == 6] \land Wedge) = \varnothing$$

$$\text{WedgeNot6} \vdash? ([n' == 6] \land Wedge) = \varnothing$$

It is sometimes convenient to write these kinds of negative test values *within* the operation schema (e.g., $[Wedge \mid n = 6] = \varnothing$), which can make the tests even more concise.

2.3 Promoting Tests

A heavily-used pattern in the POSIX Z specification is the use of *promotion* to lift the operations of one data type up to work on a more complex data type. It is useful to be able to promote the tests of those operations as well. To illustrate an elegant way of doing this, we shall promote the *Wedge* tests up to the following function space:

section *manyStates* **parents** *wedge*

$$ManyStates == [states : \mathbb{N} \pfun \mathbb{Z}]$$

$$\begin{array}{|l} \Phi ManyStates \\ \hline \Delta State \\ \Delta ManyStates \\ curr? : \mathbb{N} \\ \hline (curr? \mapsto n) \in states \\ states' = states \oplus \{curr? \mapsto n'\} \\ \hline \end{array}$$

We promote the *Wedge* operation up to the *ManyStates* level simply by conjoining *Wedge* with the *framing schema* $\Phi ManyStates$. The effect is to apply the *Wedge* operation to just the *curr*? element of the *states* function.

$$ManyWedge == \Phi ManyStates \land Wedge$$

We use the same framing schema to promote the tests. But to ensure that all inputs of the promoted operation are given, we find it useful to further instantiate the framing schema $\Phi ManyStates$, to obtain a testing-oriented framing schema ($\Phi ManyStatesTest$) that also specifies which *curr*? input will be tested and an initial value for the *states* mapping.

section *manyStatesTest* **parents** *manyStates*, *wedgeTest*

$$\Phi ManyStatesTest == [\Phi ManyStates;\ curr? == 1 \mid states = \{0 \mapsto 3, 1 \mapsto n\}]$$

$$\text{ManyWedgePos} \vdash? (\{Wedge0Gives0, Wedge0Gives5\} \land \Phi ManyStatesTest) \subseteq ManyWedge$$

In fact, this style of promoted test theorem will always be true, because for any test suite $OpTests$ of an operation Op, and any promoted operation defined as $Op_P == Op \land \Phi P$, it is true that:

$$(OpTests \subseteq Op) \land (\Phi PTests \subseteq \Phi P) \Rightarrow (OpTests \land \Phi PTests \subseteq Op_P)$$

Proof: follows from the monotonicity of $\subseteq$ and schema conjunction. □

However, there is a possibility that some of the promoted tests might be inconsistent with the framing schema (either ΦP or $\Phi PTests$), which would mean that the set $OpTests \land \Phi PTests$ would be empty or smaller than our original set of tests $Optests$. If we want to check that all of the original tests can be promoted without being lost, we can check the conjecture $(\Phi PTests \upharpoonright OpTests) = Optests$.

If this mass-promotion test fails, we may want to check each promoted test vector $v \in Optests$ separately. To do this, we can define the promoted vector as $pv == (\mu \Phi PTests \land \{v\})$, and then we can test $pv \in Op_P$.

3 The ZLive Animator

One of the tools available in the CZT system is the ZLive animator. It is the successor to the Jaza animator for Z [Utt00], and its command line interface is largely backwards compatible with that of Jaza. However, the animation algorithm of ZLive is quite different and more general. It is based on an extension of the *Z mode* system developed at Melbourne University by Winikoff and others [KWD98, Win98].

When an expression or predicate is evaluated in ZLive, it is parsed and typechecked, and then unfolded and simplified using the *transformation rules* system of CZT [UM07]. This unfolds schema operators, expands definitions and performs a variety of simplifications. For example, the test $\langlebar\ n == 0, n' == 0\ \rangle\!\rangle \in Wedge$ is unfolded to:

$$\begin{array}{l} \langlebar\ n == 0, n' == 0\ \rangle\!\rangle \in [n: \ -10\,..\,10;\ n': \ -10\,..\,10 \mid \\ \qquad\qquad n * n + n' * n' \leq 25;\ n + n' \leq 5;\ 0 \leq n'] \end{array}$$

The resulting term is then translated into a sequence of primitive relations, each of which corresponds to a single Z operator. For example, the term $n*n+n'*n' \leq 25$ is translated into a sequence of four relations:

$$\begin{array}{ll} FlatMult(n, n, tmp1), & // \ n * n = tmp1 \\ FlatMult(n', n', tmp2), & // \ n' * n' = tmp2 \\ FlatPlus(tmp1, tmp2, tmp3), & // \ tmp1 + tmp2 = tmp3 \\ FlatLessThanEq(tmp3, 25) & // \ tmp3 \leq 25 \end{array}$$

ZLive next performs a bounds analysis phase, which does a fixpoint calculation to infer conservative lower and upper bounds for each integer variable, bounds

on the size and contents of sets, and aliasing between variables. This infers that $n \in -10 \,..\, 5$ and $n' \in 0 \,..\, 10$.

The last analysis phase attempts to reorder the sequence of primitive *Flat* ... relations into an efficient computation order. Each of the relations can be executed in one or more *modes*. A mode determines whether each parameter is an input or an output. In addition, ZLive estimates the expected number of results for each mode. For example, the mode IIO:1 for $FlatPlus(x, y, z)$ means that x and y are inputs and z is an output (with one result expected), while mode III:0.6 means they are all inputs and there is a 60% probability of getting a result. The reordering algorithm gives preference to modes that have a small number of expected results, which means that filter predicates are moved as near to the front as possible, which reduces the search space.

Finally, ZLive enumerates all possible results via a depth-first backtracking search of all the solutions generated by the sorted sequence. However, for membership tests such as $A \in S$, where A is a known value and S is a set comprehension, it substitutes the value of A into the set comprehension for S and then checks whether the set is empty or not – this avoids generating all of S.

ZLive can usually evaluate expressions that range over finite sets only, and can sometimes handle expressions that contain a mixture of finite and infinite sets. So it is a useful tool for checking the correctness of tests, and can sometimes be used to generate one or all solutions of a partially specified test or schema.

4 POSIX Standardized

In this section, we briefly describe the refactored POSIX specification. The main change was to break up the original specification into sections. Figure 2 shows the structure of the resulting Z sections, using a notation similar to a UML class diagram. Each box represents a Z section, and the three parts within each box show the name of the section, the main variables within the state schema of that definition, and the names of its operation schemas (we omit Init schemas and auxiliary schemas). We also added a state schema and initialization schema to some of the sections (e.g., the *ds* section) and made several naming changes so that the specification follows the usual Z sequential style more closely.

section *ds* **parents** *standard_toolkit*

This section specifies the data system (ds) of the filing system.

$$BYTE == 0 \,..\, 255$$
$$ZERO == 0$$
$$FILE == \mathrm{seq}\, BYTE$$
$$DS == [file : FILE]$$
$$InitDS == [DS' \mid file' = \langle\rangle]$$

The *after* operator returns the subfile that starts after a given offset. We write this as an explicit definition (==) rather than axiomatically, because it is clearer, avoids possible inconsistency, and is easier to evaluate.

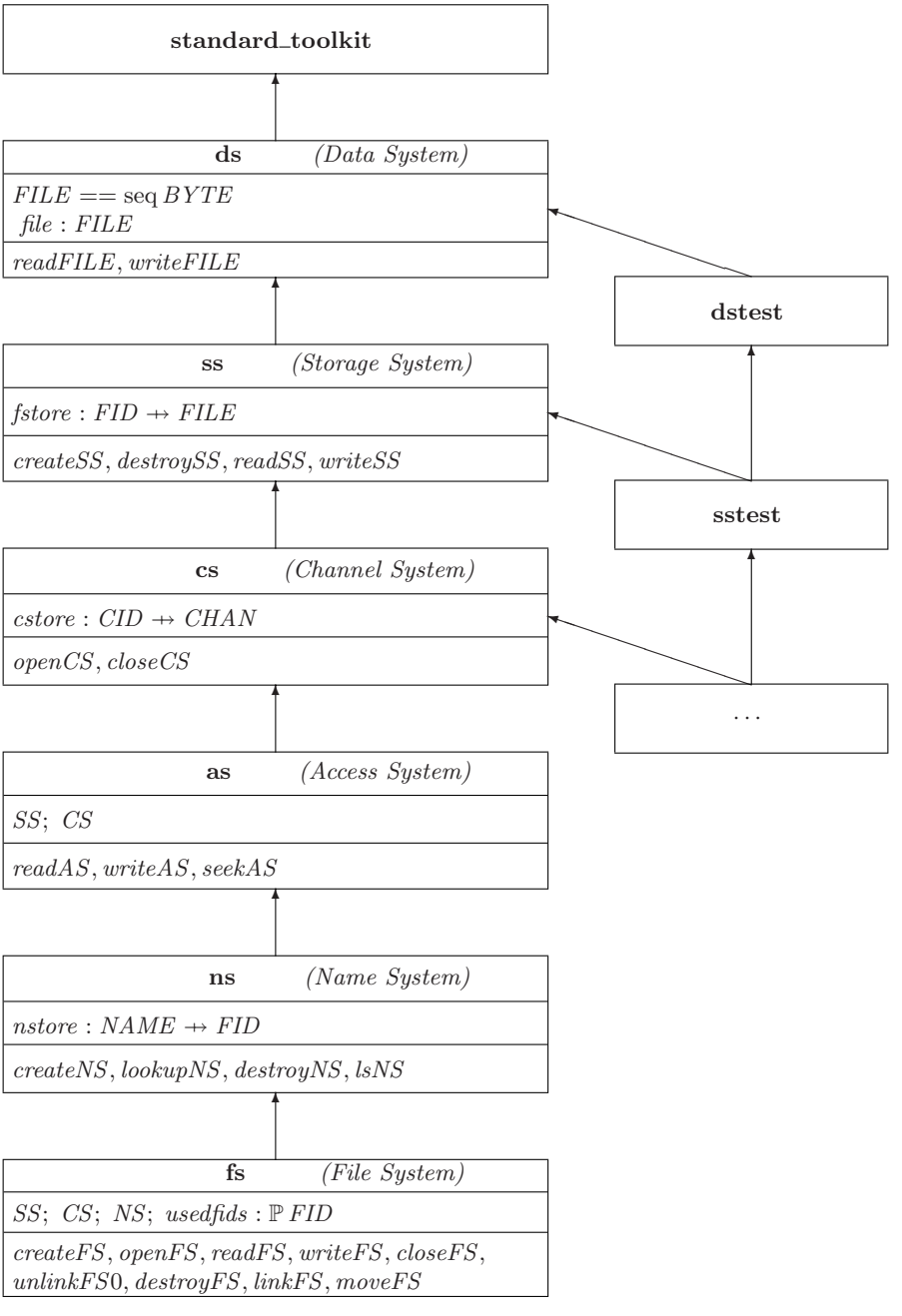


Fig. 2. Overview of Z Sections in the Refactored POSIX Specification

function 42 **leftassoc**(_*after*_)

$$\_after\_ == (\lambda f : FILE;\ offset : \mathbb{N} \bullet (\lambda i : 1..\#f - offset \bullet f(i + offset)))$$

The *readFile* operation is defined similar to the one in Morgan and Sufrin's specification but we use the usual Ξ notation for convenience here.

readFILE

$$\Xi DS$$
$$offset?, length? : \mathbb{N}$$
$$data! : FILE$$

$$data! = (1 \,..\, length?) \lhd (file\ \ after\ \ offset?)$$

The auxiliary function *zero* returns a *FILE* containing a given number of *ZERO* bytes. The infix operator *shift* takes a *FILE* and an offset and shifts the content of the file by the offset. Once again, we give an explicit rather than an axiomatic definition.

$$zero == (\lambda n : \mathbb{N} \bullet (\lambda k : 1 \,..\, n \bullet ZERO))$$

function 42 **leftassoc**(_*shift*_)

$$\_shift\_ == (\lambda f : FILE;\ offset : \mathbb{N} \bullet (1 \,..\, offset) \lhd (zero\ \ offset \frown f))$$

While the *readFILE* operation does not change the file, the *writeFile* operation given next changes the file. It is defined similar to the *writeFile* operation given in Morgan and Sufrin's specification but we use the usual Δ notation for convenience here.

writeFILE

$$\Delta DS$$
$$offset? : \mathbb{N}$$
$$data? : FILE$$

$$file' = (zero\ \ offset? \oplus file) \oplus (data?\ \ shift\ \ offset?)$$

5 Testing the DS Specification

section *dstest* **parents** *ds*

We start by testing the *InitDS* schema.

InitDSEmpty $\vdash? \ll file' == \langle\rangle \gg \in InitDS$

InitDSNot3 $\vdash? \ll file' == \langle 3 \rangle \gg \notin InitDS$

For *readFILE*, we design a set of positive tests that all work on the same input file contents, *eg*1. So when writing the set of tests, it is convenient to use the schema calculus to factor out the unchanging *file*, *file*′ from the other test values.

$$
\begin{aligned}
&eg1 == \langle 1, 255 \rangle \\
&dsPos == [file == eg1;\ file' == eg1] \land \\
&\qquad \{ \llparenthesis\ offset? == 0, length? == 0, data! == \langle\rangle\ \rrparenthesis, \\
&\qquad\ \llparenthesis\ offset? == 0, length? == 3, data! == \langle 1, 255 \rangle\ \rrparenthesis, \\
&\qquad\ \llparenthesis\ offset? == 1, length? == 1, data! == \langle 1 \rangle\ \rrparenthesis, \\
&\qquad\ \llparenthesis\ offset? == 3, length? == 2, data! == \langle 0, 0 \rangle\ \rrparenthesis \\
&\qquad \}
\end{aligned}
$$

$$
\text{readFILEPos} \ \vdash?\ dsPos \subseteq readFILE
$$

Here is the output from ZLive when we use its `conjectures` command to evaluate all the conjectures in this *dstest* section.[2]

```
dstest> conjectures
Conjecture on line 7 (InitDSEmpty)
true
Conjecture on line 11 (InitDSNot3)
true
Conjecture on line 29 (readFILEPos)
false
```

To investigate the failing conjecture *readFILEPos* on line 29, we ask ZLive to evaluate $dsPos \setminus readFILE$. This displays just the test vectors that are not members of *readFILE*, which tells us that the third and fourth tests failed. For each of these, we investigate why it failed by using the ZLive '`do`' command to search for any solution to the *readFILE* schema with the given input values. For the third test, we get this output:

```
dstest> do [readFILE | file=eg1; offset?=1; length?=1]
```

$$
\begin{aligned}
&1 : \llparenthesis\ file == \{(1,1),(2,255)\}, file' == \{(1,1),(2,255)\}, \\
&\qquad offset? == 1, length? == 1, data! == \{(1,255)\}\ \rrparenthesis
\end{aligned}
$$

Oops, this test should have had 255 in *data*! rather than 1, because in POSIX, $offset? = 1$ refers to the *second* byte of the file.

```
dstest> do [readFILE | file=eg1; offset?=3; length?=2]
```

[2] We have added the conjecture names into the ZLive output by hand in this example, but hope to automate this in the future. The problem is that the Z standard currently does not pass the conjecture names from the LaTeX markup to the Unicode markup, so getting access to the names within the parser and ZLive will require some extensions to the Z standard, which we have not yet made.

$$1 : \langle\!| \mathit{file} == \{(1,1),(2,255)\}, \mathit{file}' == \{(1,1),(2,255)\}, \\ \mathit{offset}? == 3, \mathit{length}? == 2, \mathit{data}! == \{\} |\!\rangle$$

Ah, of course! Reading past the end of the file should return empty *data*!, rather than zeroes. (When designing this fourth test, Mark was incorrectly thinking of the behaviour of the Write command past the end of the file, which inserts zeroes automatically.) Once these two errors in the expected output values are corrected, all tests give true.

5.1 Negative Tests for *readFILE*

Our first two negative tests check that -1 is not a valid input for *offset*? or *length*?. In the latter test (*ReadNotLenNeg*), we show how we can write the test values inside the *readFILE* schema, which can sometimes be more convenient.

$$\text{ReadNotOffNeg} \quad \vdash? \ ([\mathit{offset}? == -1] \land \mathit{readFILE}) = \{\}$$

$$\text{ReadNotLenNeg} \quad \vdash? \ [\mathit{readFILE} \mid \mathit{length}? = -1] = \{\}$$

The remaining negative tests are partially specified, so each conjecture actually checks a set of negative test tuples. The *ReadNoChange* test checks that the read operation does not change the contents of our example file *eg*1. The *ReadNotLonger* test checks that the output *data*! is never longer than the contents of file *eg*1. This is actually proving that the property $\#\mathit{data}! > 2$ is false for all lengths $0 \,..\, 3$, which is a form of finite proof by exhaustive enumeration.

$$\text{ReadNoChange} \quad \vdash? \ [\mathit{readFILE} \mid \mathit{file} = \mathit{eg}1; \ \mathit{file}' \neq \mathit{eg}1] = \{\}$$

$$\text{ReadNotLonger} \quad \vdash? \ [\mathit{readFILE} \mid \mathit{file} = \mathit{eg}1; \ \mathit{offset}? = 0; \ \mathit{length}? < 4; \ \#\mathit{data}! > 2] = \{\}$$

This illustrates that there is a continuum between testing and proof. We usually test just one input-output tuple at a time, but the idea of testing can be extended (as in this paper) to allow a given property to be evaluated for all members of a finite/small set. This is similar to model-checking, where properties of finite systems are proved by exhaustive enumeration. Animators like ZLive use a mixture of symbolic manipulation techniques and exhaustive enumeration. The more they use symbolic manipulation, the closer they become to general theorem provers. So the testing-proof continuum ranges from testing of single input-output tuples, through enumeration (or model-checking) of finite systems, to full symbolic proof.

We can write positive and negative unit tests for the *writeFile* operation in a similar way to *readFile*, but space does not permit us to show the details of this.

6 Testing the SS Specification

The storage system is reponsible for mapping *file identifiers FID* to file contents. While testing, we instantiate FID to naturals, so test values are easier to write.

section *ss* **parents** *ds*

$FID == \mathbb{N}$
$SS == [fstore : FID \pfun FILE]$

The *ss* section then defines *createSS* and *destroySS* operations, plus the following framing schema, which is used to promote *readFILE* and *writeFILE*.

$$
\begin{array}{|l}
\hline
\Phi SS \\
\Delta SS;\ \ \Delta DS;\ \ fid? : FID \\
\hline
fid? \in \mathrm{dom}\, fstore \\
file = fstore(fid?) \\
fstore' = fstore \oplus \{fid? \mapsto file'\} \\
\hline
\end{array}
$$

$readSS == (\Phi SS \land readFILE) \setminus (file, file')$

section *sstest* **parents** *dstest*, *ss*

To promote the ds tests, we define a special case of the ΦSS framing schema.

$\Phi SSTest == [\Phi SS \mid fid? = 101;\ fstore = \{100 \mapsto \langle 3, 5\rangle, 101 \mapsto file\}]$

Then we can promote the *dsPos* tests and check if they satisfy *readSS*.

$\mathrm{SSTestPos} \vdash? (dsPos \land \Phi SSTest) \setminus (file, file') \subseteq readSS$

Unfortunately, due to an inefficiency in its sorting/optimization algorithms, ZLive currently says the left side of the $\subseteq$ is too large to evaluate. However, it should be capable of evaluating it, and we expect that it will be able to in the next few months.

7 Conclusions

In this paper, we have proposed a framework for unit testing Z specifications. The framework uses the sections and conjectures of the Z standard to allow various kinds of validation tests to be expressed in an elegant and concise style. The ability to promote a large set of tests in a single expression makes it practical to develop multiple layers of tests, matching the layers of the specification. Testing is a useful validation technique for specifications, especially when the execution of the tests can be automated, as we have done with ZLive. We believe that most Z specifications should include validation unit tests in this style.

We plan to add a `unittest` command to ZLive that executes all the unit tests in all the sections whose names end with 'Test'. This will make it easy to rerun all unit tests after each modification of a specification, so will support regression testing during development of Z specifications. It would also be useful if ZLive measured structural coverage metrics of the operation schemas during testing,

so that we can see what level of, say, branch coverage (each predicate evaluating to true and to false) our test suite obtains.

Our style of unit testing is quite complementary to the specification validation facilities of ProB/ProZ [LB08], because we focus on unit testing of each individual operation schema using a manually designed test suite, while ProB focusses on automatic testing of sequences of operations, and tests only a few input values of each operation. The goal of our unit testing is to test the input-output functionality of each operation, while the main goal of ProB is to try to validate several standard system properties such as absence of deadlock and preservation of invariants.

The refactored Z specification of POSIX, and our simple test suite, may be a useful starting point for other researchers who want to work on refinement or proofs about the POSIX case study. It is available from the CZT sourceforge website [CZT]. Interestingly, the original POSIX specification did include several examples that used test values to illustrate some of the operations, but those examples were written within the English commentary, so were not even type-checked, let alone proved correct. Our unit tests are more systematic and are formalized so that they can be checked automatically by an animator like ZLive.

References

[BHW06] Bicarregui, J.C., 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)

[CS94] Carrington, D.A., Stocks, P.: A tale of two paradigms: formal methods and software testing. In: Proceedings of the 8^{th} Z User Meeting (ZUM 1994), June 1994, pp. 51–68. Springer, Heidelberg (1994)

[CZT] Community Z tools, `http://czt.sourceforge.net`

[FFW07] Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/EVES: an experiment in the verified software repository. In: ICECCS 2007: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 3–14. IEEE Computer Society, Los Alamitos (2007)

[Hie97] Hierons, R.: Testing from a Z specification. Software Testing, Verification & Reliability 7, 19–33 (1997)

[Hoa03] Hoare, T.: The verifying compiler: A grand challenge for computing research. Journal of the ACM 50(1), 63–69 (2003)

[JH07] Joshi, R., Holzmann, G.J.: A mini challenge: build a verifiable file system. Formal Aspects of Computing 19(2), 269–272 (2007)

[KWD98] Kazmierczak, E., Winikoff, M., Dart, P.: Verifying model oriented specifications through animation. In: Proceedings of the 5th Asia-Pacific Software Engineering Conference, pp. 254–261. IEEE Computer Society Press, Los Alamitos (December 1998)

[LB08] Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer (page accepted for publication, 2008)

[MS84] Morgan, C., Sufrin, B.: Specification of the UNIX filing system. IEEE Transactions on Software Engineering (1984)

[Sto93] Stocks, P.: Applying Formal Methods to Software Testing. PhD thesis, The University of Queensland (1993), http://www.bond.edu.au/it/staff/publications/PhilS-pubs.htm

[UM07] Utting, M., Malik, P.: Transforming Z with rules. In: ZUM 2007 at ICECCS 2007 in conjunction with FMinNZ 2007, Auckland (July 2007)

[Utt00] Utting, M.: Data structures for Z testing tools. In: Schellhorn, G., Reif, W. (eds.) FM-TOOLS 2000, 4th Workshop on Tools for System Design and Verification, vol. 2000-07, pp. 2000–2007. Ulmer Informatik Berichte (May 2000)

[Win98] Winikoff, M.: Analysing modes and subtypes in Z specifications. Technical Report 98/2, Melbourne University (1998)

Autonomous Objects and Bottom-Up Composition in ZOO Applied to a Case Study of Biological Reactivity

Nuno Amálio[1], Fiona Polack[2], and Jing Zhang[3]

[1] Dept of Computing, City University, Northampton Sq, London, EC1V 0HB, UK
nuno.amalio@gmail.com
[2] Dept of Computer Science, University of York, York, YO10 5DD, UK
fiona@cs.york.ac.uk
[3] China Exim Bank, DongCheng District, Beijing 100009, P.R. China
jackiezhang219@hotmail.com

Abstract. As part of our work on the formal analysis of object-oriented models, we turn to systems where many autonomous individuals interact to give rise to complex collective behaviour. We adapt our ZOO [1,2] structuring and apply it to a case study based on a published model of part of the immune system [3]. The formalisation calls for a bottom-up solution with no central control over individual units, and includes an approach to represent *feedback channels* enabling broadcast communication between individuals and across levels.

Keywords: object-orientation, Z, complex systems, statecharts.

1 Introduction

Natural systems often comprise many autonomous individuals interacting locally to give rise to complex collective behaviour. The analogy of individual and object suggests that the object-oriented (OO) paradigm is suited to modelling such systems. However, more classical OO design builds collective behaviour as orchestrations (or collaborations) of object computations with a clear centre of control. Designers use modelling notations, such as UML collaboration and sequence diagrams [4], to specify such central orchestrations. They use design patterns, such as *Mediator* [5] and *Controller* [6], to build units that encapsulate the logic of object orchestration. In systems characterised by numerous and complex interactions among individuals, explicit control is no longer possible or desirable.

We illustrate an approach using our ZOO structuring to model such systems. The approach is based on a representation of autonomous objects that are capable of sensing events, and of reacting to them when appropriate, and on bottom-up composition of local structures to bring about global behaviour with no central control. We use a case study of biological reactivity [3], a typical example of such systems. We focus on modelling issues, rather than on biological

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 323–336, 2008.

aspects of the case study. The approach differs from our previous representation of statecharts [1,2], where objects are passive units, reacting to events only when stimulated by central control.

2 The Case Study Model

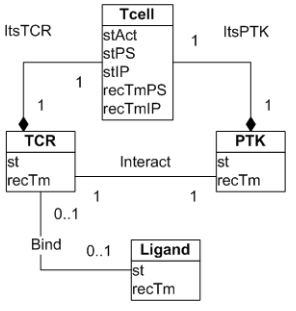


Fig. 1. Main classes involved in T-cell activation

The original case study comes from the interdisciplinary work of Irun Cohen, David Harel and others, and concerns the events that take place in a cell following the engagement of a T-cell antigen receptor (TCR) by its cognate ligand. These events can lead either to cellular activation (lymphokine production and cell proliferation) or to anergy (a state of unresponsiveness). Kam, Harel and Cohen [3,7] rework a state-transition model in a simple Boolean formalism [8] to a diagrammatic model comprising a class diagram and a suite of statecharts.

2.1 A T-cell class Diagram

The original model represents the static structure of T-cell activation in a class diagram and text commentary [3,7]. In figure 1, we rework this to a UML-style class diagram. In the diagram,

- TCR and PTK are components of the Tcell class. This is represented as the composition associations ItsPTk and ItsTCR.
- Ligand and TCR are linked by the Bind association. Note that we change the multiplicity of the association (originally it was $1-1$); this is required by our semantics, for consistency with the statecharts (see discussion).
- TCR and PTK are associated by the Interact association.
- In all classes, the *st* attributes record the current state of an object as defined by its class's statechart. The *recTm* attributes are used by objects to keep track of the passage of time.

2.2 Statecharts

In the statecharts of [3], each class has its own events, objects are capable of generating events on other objects, and global behaviour is based on chain reactions: from an event coming from the environment or some time delay, an object reacts and may cause further reactions on other objects. In our model, events come from the environment and they are shared, each being propagated to all objects, which then decide autonomously whether they should react or not. These autonomous object reactions, however, must not take place in an unconstrained way: they must be synchronised and subject to feedback from other objects. What was in [3] a chain reaction, where some object would generate

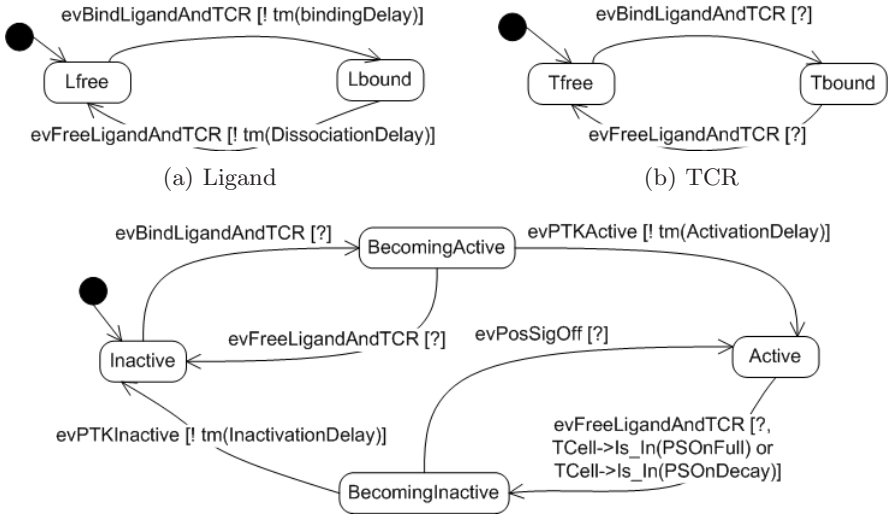


Fig. 2. Statecharts of classes Ligand (model with rebinding), TCR and PTK

events on other objects, is in our model a single synchronised event subject to feedback. To allow this we introduce the notion of a *feedback channel* to enable objects to broadcast information and receive feedback. We show how we represent feedback channels in ZOO and introduce a special notation to represent them in statecharts.

Figures 2 and 3 present statecharts of classes Ligand, TCR, PTK and Tcell. These are the result of a meaning-preserving refactoring of the original statecharts (see [3] for a detailed explanation). We label every state transition with an event name; those corresponding to timing delays have a timing pre-condition on the appropriate state transition. As discussed above, the evaluation of a timing pre-condition must affect all objects that share that event; we use the following notation to represent the *feedback channel* that allows this: (a) a state transition pre-condition with the symbol ! means that the object broadcasts the result of evaluating the pre-condition to other objects; (b) a state transition pre-condition with the symbol ? means that the object receives feedback regarding the pre-condition. Because events are shared, events evBindLigand and evTCRBound of [3] are merged into evBindLigandAndTCR, and event evTCRFree becomes evFreeLigandAndTCR. We also collapse superstates in the original diagrams. Superstates are a notational convention to reduce the overall number of transitions [9]; in [3] superstates do not reduce the number of transitions so the structure can be simplified.

3 The ZOO Model

ZOO [2,1] is an OO style for Z that uses the logical concept of views, each describing one aspect of an OO system. A ZOO model is built incrementally and in a layered fashion, view-by-view.

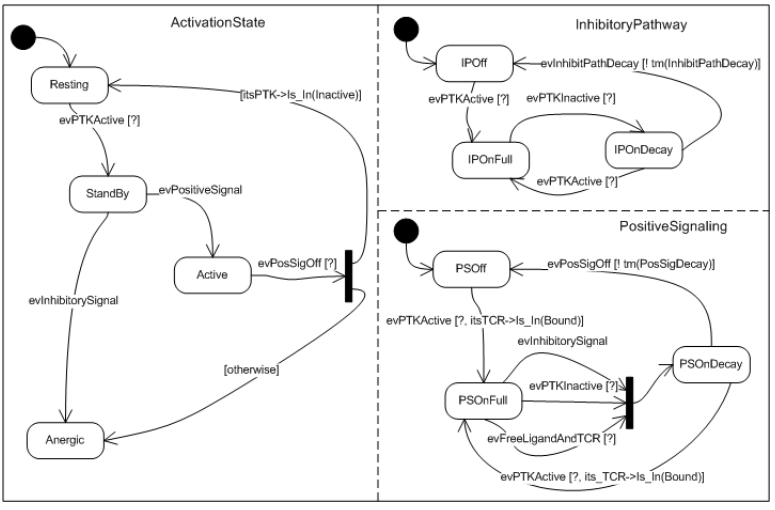


Fig. 3. The statechart of class Tcell

A ZOO model is built from structures representing the main OO concepts: objects, classes, associations and system. An object is represented as an atom, a member of the set of all possible objects, and of the set of possible objects of its class. In ZOO, a class is a *promoted* Z abstract data type [10] with a dual representation. The class intension (the inner type) defines a class in terms of properties shared by its objects (for example, a class *Person* with properties *name* and *address*). Class extension (the outer type) defines a class in terms of its existing object instances (for example, *Person* is $\{MrSmith, MrAnderson, MsFitzgerald\}$). An association relates objects of classes and denotes a set of object tuples describing linked objects. The ZOO representation is as a Z relation between class objects. Systems are ensembles of classes and associations, with properties expressed in terms of the ensemble. The structure of a ZOO system of two classes and an association is shown in figure 4.

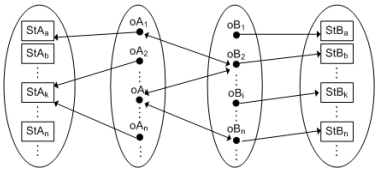


Fig. 4. Objects (oA, oB), classes ($\{oA \mapsto StA\}$, $\{oB \mapsto StB\}$), associations ($\{oA, oB\}$) and the system in ZOO

The following highlights the most interesting issues regarding the ZOO specification of the biological system. The complete ZOO specification is available online [11]. The reader is referred to [2,1] for a detailed explanation on the structure of a ZOO specification.

3.1 Time and Events

To use ZOO with the statecharts, we need to consider how to handle time and events. Time, because the behaviour of objects is subject to characteristic time delays. Events, because they differ from our previous our previous work [2] on state-

charts, where events were class operations that affect one class only; here events are global and they can affect various objects.

In the ZOO specification [11], time and events are defined in the structural view. The type *TIME* represents time as a set of time-points isomorphic to the natural numbers. This gives a simple discrete view of time that is enough for our purposes and admits use of integer operators. The constants corresponding to the characteristic time delays of [3] are defined axiomatically to be of type *TIME*. Events (figures 2 and 3) are defined as a Z free type.

3.2 States, State Transitions and Feedback Channels

We need to handle states and transitions between them and the special case of feedback that is associated with events. The following shows how we handle this in ZOO's structural and intensional views.

Structural View. We need to introduce a *feedback channel* to broadcast the evaluation of pre-conditions. This is defined in the structural view (see [11]). We introduce the set *TCEvent* of all events that are time constrained and whose evaluation of timing pre-conditions is to be broadcast. We also define the type *EvPreEval* of sets of *TCEvent*, an instance of which indicates the event time pre-conditions that are true. Finally, the operator *IsTrueEvPre*, which describes a property of sets of *TCEvent* and *EvPreEval*, indicates whether the evaluation of the pre-condition is *true* in a set of pre-condition evaluations; this is just an abbreviation for set membership, but provides a useful abstraction making expressions involving timing pre-conditions easier to read.

$$TCEvent == \{evBindLigandAndTCR, evFreeLigandAndTCR, evPTKActive, evPTKInactive, evPosSigOff\}$$

$$EvPreEval == \mathbb{P}\, TCEvent$$

$$IsTrueEvPre\_ : \mathbb{P}(TCEvent \times EvPreEval)$$

$$\forall\, tev : TCEvent;\ evpreEv : EvPreEval \bullet IsTrueEvPre(tev, evpreEv) \Leftrightarrow tev \in evpreEv$$

Intensional View. As in [2], the intensional view includes a type to represent statechart states. In [11], an attribute *st* in each class intension records the current state; in the class's initialisation *st* is equated to the statechart's initial state. A class whose objects need to track time has a *recTm* attribute. This is initialised with the current system time, and updated on each event that the class's objects need to respond to; the current time is provided by the environment as the input *now*?. This is illustrated here for the *Ligand* class.

$$LigandST ::= Lfree \mid Lbound$$

Ligand

$$st : LigandST$$
$$recTm : TIME$$

LigandInit

$$LigandInitI$$
$$Ligand'$$
$$st' = stLfree$$
$$recTm' = now?$$

LigandInitI

$$now? : TIME$$

A statechart transition is represented by a Z operation schema, with an event ($ev?$), current time ($now?$) and a set of pre-condition evaluations to communicate or receive feedback ($evPreEval?$) as inputs. We define the conditions under which the transition's pre-condition is true to communicate feedback and we require the pre-condition to be true. The pre-condition requires (a) the event to be that specified for the transition, (b) the object's current state to be the before state of the transition, and (c) for a transition with a timing condition, the time elapsed since the last recorded time must be the same as the relevant timing delay. The postcondition says (a) that the current state becomes the transition's after state and (b) that the recorded time becomes the current time. The definitions below specify how a Ligand broadcasts feedback and how a TCR makes use of feedback received from a Ligand object.

$$
\begin{array}{|l}
\hline
Ligand\Delta FrFreeToBound \\
\Delta Ligand \\
now? : TIME \\
ev? : Event \\
evpreEval? : EvPreEval \\
\hline
IsTrueEvPre(ev?, evpreEval?) \\
\quad \Leftrightarrow ev? = evBindLigandAndTCR \\
\quad \land now? - recTm = BindingDelay \\
\quad \land st = Lfree \\
IsTrueEvPre(ev?, evpreEval?) \\
st' = Lbound \land recTm' = now? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
TCR\Delta FrFreeToBound \\
\Delta TCR \\
ev? : Event \\
evpreEval? : EvPreEval \\
\hline
ev? = evBindLigandAndTCR \\
IsTrueEvPre(ev?, evpreEval?) \\
st = Tfree \land st' = Tbound \\
\hline
\end{array}
$$

3.3 Autonomous Objects That Can Sense

We need to define the transition occurrence and autonomous behaviour of objects. To do this, objects sense events – they are constantly listening for events that happen in the system. When an event is detected, objects react appropriately; otherwise they do nothing.

Intensional View. We now show how we make the behaviour of Ligand objects autonomous. The act of doing something is modelled as the disjunction of state transition operation schemas ($Ligand\Delta DoSomething$). An object does nothing if the pre-condition of doing something is not true ($Ligand\Xi DoNothing$). An object reacts by either doing something or doing nothing ($LigandReact$):

$$Ligand\Delta DoSomething == Ligand\Delta FrFreeToBound \lor Ligand\Delta FrBoundToFree$$

$$Ligand\Xi DoNothing == \neg (\mathrm{pre}\, Ligand\Delta DoSomething) \land \Xi Ligand$$

$$LigandReact == Ligand\Delta DoSomething \lor Ligand\Xi DoNothing$$

3.4 Orthogonal Components of Statecharts

The statechart of Tcell comprises three orthogonal components, each with its own statechart. In ZOO, we represent each statechart separately in the intensional view (each orthogonal component has its own intensional view) and then

compose them using schema conjunction. See the definition of Tcell intension in [11] for further details.

3.5 Bottom-Up Composition

The collective behaviour of the system is a composition of behaviours of the autonomous objects. It is expressed as the conjunction of the react operations from every structure, class extension or association.

$$SysObjectsReact == \mathbb{S}LigandReact \wedge \mathbb{S}TcellReact \wedge \mathbb{S}PTKReact \wedge \mathbb{S}TCRReact \\ \wedge \mathbb{A}BindReact \wedge \mathbb{A}InteractReact \wedge \mathbb{A}ItsPTKReact \wedge \mathbb{A}ItsTCRReact \setminus (evpreEval?)$$

Note that there is no central control to determine which specific computation some structure needs to carry out. We simply say that it is up to each component to react and do what is appropriate.

3.6 Handling Global Pre-Conditions

In [1,2], we showed that certain pre-conditions are only expressible in terms of the ensemble, because they only make sense in terms of the composition. In [1,2] we gave examples of such pre-conditions and how to represent them in the global view associated with system operations.

This case study includes conditions of state transitions that involve the state of other objects, which cannot be expressed locally in a direct way. One such condition occurs in the statechart of PTK (figure 2(c)), on the transition from Active to BecomingActive. The condition involves the PTK's Tcell: the positive signaling component must either be in state PSOnFull or PSOnDecay.

The problem in this particular context is exacerbated, as these pre-conditions cannot be formalised globally either, because they are associated with a transition whose processing is made locally – at the level of objects. Our solution to this problem is to again use the idea of *feedback channels* and propagate global pre-conditions to the local object level.

Structural View. The structural view introduces the global pre-condition feedback channel which is similar to the definition of the feedback channel given above. In [11], the definition of this feedback channel introduces a constant for each global pre-condition defined as part of the free type *GPre*, and the type *GPreEval* of sets of *GPre*, an instance of which says the global pre-conditions that are true. It also introduces *IsTrueGPre*, a property of sets of *GPre* and *GPreEval*, indicating whether global evaluation of some global precondition (*GPre*) is true in a set of global evaluations.

Intensional View. The intensional view describes global pre-conditions as local pre-conditions by using the *IsTrueGPre* operator. This involves the input *gpreEval*? which receives a set of global pre-conditions; the global pre-condition of this transition is evaluated with respect to *gpreEval*? using *IsTrueGpre*. In [11], this can be observed in the definition of the transition from Active to BecomingActive in the class PTK (schema $PTK\Delta FrActiveToBecomingInactive$), which is then the used to define the react operation of PTK as shown above for Ligand.

Global View. In the global view, we define a schema for each global pre-condition and specify the conditions under which the global pre-condition is true. The following describes the global pre-condition for the transition from Active to BecomingActive in the statechart of PTK.

$$
\begin{array}{|l}
\hline
GPrePTKFrActiveToBecomingActive \\
\mathbb{S}Tcell;\ \mathbb{S}PTK \\
oTcell? : OTcellCl \\
gpreEval? : GPreEval \\
\hline
IsTrueGPre(PTKFrActiveToBecomingActive, gpreEval?) \\
\quad \Leftrightarrow (stTcell\ oTcell?).stPS = PSOnFull \\
\qquad \lor (stTcell\ oTcell?).stPS = PSOnDecay \\
\hline
\end{array}
$$

Finally, we define the global reaction by conjoining the reaction of local structures, the input connector, the global pre-conditions. We also define the global reaction to be total: the system either does something or nothing.

$$
\begin{array}{l}
SysReactDoSomething == \Delta System \land SysObjectsReact \land ConnSysReact \\
\quad \land\ GPrePTKFrActiveToBecomingActive \\
\quad \land\ GPreTcellActFrActiveToRestingOrAnergic \\
\quad \land\ GPreTcellPSFrOffToOnFull \\
\quad \land\ GPreTcellPSFrOnDecayToOnFull \setminus (oTCR?, oPTK?, oLigand?, oTcell?, gpreEval?) \\
SysReactDoNothing == \neg\ (\mathrm{pre}\ SysReactDoSomething) \land \Xi System \\
SysReact == SysReactDoSomething \lor SysReactDoNothing
\end{array}
$$

The feedback between local (or intensional) and global levels takes place through the internal input *gpreEval?*. The input is received by the local view through the operation *SysObjectsReact* (in our example, this includes *PTKReact*, which in turn includes *PTK*Δ*FrActiveToBecomingInactive*). In the global view, the input is transmitted through the *GPre* schemas. Both *SysObjectsReact* and the *GPre* schemas are part of the *SysReact* operation.

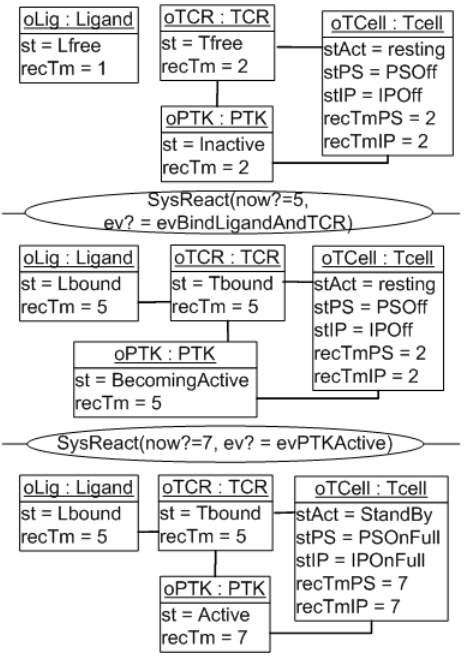
4 Snapshot Analysis

config : Constants
BindingDelay = 4
DissociationDelay = 4
ActivationDelay = 2
InactivationDelay = 4
InhibitPathDecay = 5
PosSigDecay = 5

Fig. 5. Values of characteristic timing delays used in snapshot analysis

The ZOO model constructed above is now analysed using our snapshot analysis technique [2,12]. First, we assign values to the characteristic timing delays for the purpose of the analysis and represent this configuration information as a snapshot (figure 5). The snapshot analysis technique is illustrated here with the snapshot sequence and the snapshot pair of figure 6.

The snapshot sequence of figure 6(a) simulates the engagement of a ligand with a TCR followed by the activation of the PTK. In the first snapshot, the ligand and TCR objects are free — not linked by association Bind and in states Lfree and Tfree. The second snapshot



(a) Snapshot Sequence: Ligand-TCR binding followed by PTK activation

(b) Ligand enters state bound without engaging with a TCR

Fig. 6. Snapshots used in the analysis of ZOO model given above

presents the effect of event evBindLigandAndTCR, where ligand and TCR become bound — linked by association Bind and in states Lbound and Tbound — and the PTK starts its activation process — state BecomingActive. The last snapshot in the sequence shows the effect of a PTK activation. The PTK changes to the Active state, and the states of the Tcell components also change as defined by the statechart. All snapshot proofs for this sequence (see [2]) are provable in Z/Eves [13].

As observed in [8], this sequence is possible because in this setting *ActivationDelay* < *DissociationDelay* (see figure 5). If it were not the case, the event evFreeLigandAndTCR would occur before the PTK activation event, and so ligand and TCR would dissociate becoming both free and the PTK would go back to the Inactive state (see statecharts).

The snapshot-pair (figure 6(b)) highlights an oddity of the statecharts model given above. It shows a system with two ligand and one Tcell composite objects, where one of the ligands is bound to the Tcell and the other is free. The free ligand is stimulated to react, and it does so changing to the state Lbound, but without actually being bound to a TCR. This is an inconsistency that is accepted by the model of the system — snapshot proof is provable in Z/Eves. This behaviour was introduced when we relaxed the multiplicity of the association Binds to

Table 1. Experimental data regarding proof of snapshots with the Z/Eves theorem prover. Table indicates snapshot and number of proofs required to prove it; for each proof it says how many Z/Eves commands were required and how much time it took to carry out the proof in Z/Eves.

Snapshot	No of proofs	Proof	No of Z/eves commands	Time to prove
Fig. 6(a)	2	Correctness of 1st snapshot pair	5	32s
		Correctness of 2nd snapshot pair	5	38s
Fig. 6(b)	1	Correctness of snapshot pair in model without fix	6	32s
Fig. 6(b)	1	Negation of correctness snapshot pair in model with fix	11	1m32s

$0 \mathinner{..} 1, 0 \mathinner{..} 1$. In [3], a ligand is linked to a TCR throughout its lifetime, binding and re-binding always involves the same (ligand, TCR) pair, and so the problem highlighted by the snapshot-pair does not occur. In our model, this problem can be fixed with a global pre-condition on the transition free to bound in the ligand statechart by requiring that the TCR is in the free state[1] The ZOO model of [11] does precisely this and here the snapshot is no longer provable (the negation is provable, which means that the model no longer accepts the snapshot).

Table 1 presents experimental data regarding the proof of the snapshots above with the Z/Eves theorem prover. The correctness of the snapshot sequence (figure 6(a)) involved two proofs (one for each pair). We proved the snapshot pair (figure 6(b)) before and after fixing the problem discussed above. After the fix, we proved the negation of correctness to conclude that the snapshot pair was no longer accepted (see [2] for further details).

5 Discussion

We illustrate a ZOO model, the result of systematic formalisation of a statechart model of part of the immune system. In [3], the authors analysed the model through simulation in the Rhapsody tool; we have analysed the ZOO model using snapshot analysis, a technique based on formal proof; all proofs were discharged in the Z/Eves interactive theorem prover [13]. Our formalisation revealed a problem with the original object model's $1 \;:\; 1$ association multiplicity of Binds. In our semantics, this multiplicity requires a link to exist in all states of the system; in [3] all relevant statecharts require a period where a

[1] The precondition of transition Lfree to Lbound becomes $!tm(bindingDelay)$, $TCell? -> ItsTCR -> Is\_In(Tfree)$.

Ligand is free, and has no link in the Binds association. Our anaysis also revealed an erronoeus behaviour introduced by our relaxation to the $1 : 1$ association multiplicity constraint (see above).

This case study started as a masters project [14], applying the ZOO templates and snapshot analysis to the Tcell statecharts of [3,7]. We soon found that the OO templates from [2] were not sufficient for this small, but non-trivial system of autonomous objects. The Z representation presented here involved several stages of work, constituting an interesting modelling challenge.

Elsewhere, we show that diagrammatic notations have multiple interpretations [15]. The appropriate semantics must be chosen based on the problem at hand; our generative approach [2] is guided by this principle. Here, we capture the semantics of autonomous objects and feedback, demonstrating again the importance of interpretation. We wish to use ZOO to further explore this sort of system, and then to devise new FTL templates to capture the underlying semantics and modelling patterns, so that our generative approach [2] to build ZOO models can be applied to other models of this sort.

The phenomena evidenced in our case study are not exclusive to biological systems. They also appear in graphical user interfaces (GUIs) where an event triggers a reaction in many components, and the GUI components have subtle interactions and constraints; we have observed this in development of a graphical user interface for a large-scale critical system. The Harel statecharts seem very appropriate to this sort of model, helping in understanding and providing an intuitive visual documentation. As said above, statecharts have multiple-interpretations, and so they are at their best when accompanied with a clear description of the semantics being adopted, or, like we did here, accompanied with a description in a more precise formal language that effectively resolves the semantic ambiguities of the diagrammatic description.

The ZOO representation presented here highlights several patterns for Z modelling of this sort of system. The *autonomous objects* pattern enables a bottom-up composition of lower-level autonomous structures in the global view: instead of orchestration by system events, we simply ask lower-level structures to react. The *feedback channels* pattern is also key; it allows propagation of global constraints to the local level enabling full bottom-up composition: constraints that needed handling globally can be propagated to the lower-level so that they can be expressed naturally. It also enables exchange of information between objects, enabling objects to act autonomously but in a synchronised way subject to feedback regarding the evaluation of event pre-conditions.

The autonomous objects and bottom-up composition provide an interesting and practical approach to the dreaded *frame problem* [16]. No longer is a global frame needed to describe the structures that do not change as a result of a global operation; that decision is taken locally by the individual *reacts*: an object either changes or does nothing.

Our *feedback channels* pattern enables *broadcast* communication between objects. This was observed to be a limitation of Rhapsody in [7], where the authors mention that communication between objects is “point-to-point” and that

"Rhapsody does not allow *broadcasting*". Here, feedback channels enable broadcasting, avoiding the need to send messages to each object that is interested in some event — one object (the main entity responsible for dealing with the event) emits information to be processed by other objects. The pattern relies on an internal input that acts as an internal information channel. To ease specification, the pattern introduces a predicate (defined as an operator) that gives some abstraction, helping in specifying and reading Z expressions involving feedback-channels. The pattern may look odd at first, but once it is understood it is repeatable and applicable in slightly different contexts (as the paper has shown). The key in understanding what the pattern does is that there is one transmitting and several receiving ends that must be connected in the global view so that communication through the channel can effectively take place.

The *feedback channels* pattern and other aspects of ZOO may look like *advanced Z*, but this advanced Z is repeatable and its underlying structure can be represented as templates expressed in the Formal Template Language (FTL) [17,2]. FTL representations of Z enable generation of Z by a process of template instantiation; [2] provides a catalogue of FTL templates to support generation of ZOO models that comprises most of structures used here; the new patterns introduced here would also be expressible as FTL templates. More advanced Z users can grasp the general pattern from the instances described here or from FTL, and apply it to various contexts where feedback-channels may be applied, and even find new contexts where the underlying structure is applicable. More naive Z users and newcomers can either use the template form, and have the appropriate ZOO generated from templates by providing an instantiation, or use the diagrams and have the ZOO generated from their diagrammatic specifications via a process of template instantiation. Provided that there is a template representation for the statecharts interpretation presented here, then generation of a ZOO specification from diagrams is possible by following the approach defined in [2].

Perhaps the biggest limitation of our approach lies in the fact that the system needs to be stimulated from the environment. Our model does not describe a self-reactive system; the environment needs to stimulate a reaction by providing the event to be observed and the current time. This is because the Z schema calculus does not allow recursive operations. Despite this limitation, we are able to perform the analysis of the original study [8], and observe the way the relationships between different time delays influence the behaviour of the system and how the system reacts as a whole to events. This is done within a mathematical model that expresses clearly what are the system's constituents parts and how its parts collaborate to make the whole.

6 Related Work

In our ZOO formalisation, we can observe several characteristics of emergent systems. One such characteristic is that of levels, where our constituent parts are at the lower, local level, and the emergent property, the collective behaviour of the objects, at the higher global level [18]. Another characteristic is the very

interesting interplay between levels – it is not only the parts that determine the ensemble, but also how the ensemble affects its parts [18]. This can be observed in our need to propagate constraints from the global to the local level.

Liu and Tsui [19] describe a nature-inspired computational setting as a system operated by populations of autonomous entities. Each autonomous entity consists of a detector, an effector and a repository of local behavioural rules. A detector receives information related to its neighbours and the environment, an effector carries out actions and facilitates information sharing between autonomous entities, and the local rules indicate how the entity should react to the information collected by the detector. Our ZOO-based model of autonomous objects matches this description. The state transitions as described by the statecharts and formalised in ZOO's intensional view are the local rules of our autonomous entities; the receiving end of the feedback channel acts as the detector, and the transmission of information through the feedback channel as the effector.

7 Conclusions

This paper presented an approach to model, in Z, systems with emergent phenomena characterised by global interactions at the individual level that give rise to a complex collective behaviour of individual units. This approach was developed in the context of ZOO, our OO structuring for Z. Our approach comprises several smaller contributions: (a) an approach to model statecharts of such systems in ZOO, (b) an approach to make objects behave autonomously, which enables bottom-up composition of object behaviours, and (c) an approach to represent *feedback channels*, which allows propagation of global level constraints to the local/individual level and broadcast communication between objects.

Our approach is illustrated with a case study of biological reactivity. We started from a statecharts model to obtain a ZOO model, which we then analysed formally using our snapshot analysis technique. Our formalisation highlighted several interesting characteristics of *emergent* systems.

References

1. Amálio, N., Polack, F., Stepney, S.: An object-oriented structuring for Z based on views. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 262–278. Springer, Heidelberg (2005)
2. Amálio, N.: Generative frameworks for rigorous model-driven development. Ph.D. thesis, Dept. Comp. Science, Univ. of York (2007)
3. Kam, N., Harel, D., Cohen, I.R.: Modeling biological reactivity: Statecharts vs. boolean logic. In: Proc. Int. Conf. on Systems Biology (ICSB 2001) (2001)
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Resusable Object-Oriented Software. Professional Computing. Addison-Wesley, Reading (1995)

6. Larman, C.: Applying UML and patterns: an introduction to object-oriented analysis and design (1998)
7. Kam, N., Harel, D., Cohen, I.R.: The immune system as a reactive system: Modeling T cell activation with statecharts. In: VLFM 2001 (2001)
8. Kaufman, M., Andris, F., Leo, O.: A logical analysis of T cell activation and anergy. Proc. Natl. Acad. Sci. USA 96(7) (1999)
9. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
10. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof (1996)
11. Amálio, N., Polack, F., Zhang, J.: Zoo specification of T-cell statecharts (2008), http://www-users.cs.york.ac.uk/~fiona/PUBS/zoo-tcell-statecharts.pdf
12. Amálio, N., Stepney, S., Polack, F.: Formal Proof from UML Models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 418–433. Springer, Heidelberg (2004)
13. Saaltink, M.: The Z/EVES system. In: Bowen, J., Hinchey, M. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
14. Zhang, J.: Extending an approach to formalize diagrammatic model: Analysis of Harel's T cell model. Master's thesis, Dept. Comp. Science, Univ. of York (2006)
15. Amálio, N., Stepney, S., Polack, F.: Modular UML semantics: Interpretations in Z based on templates and generics. In: Van, H.D., Liu, Z. (eds.) FACS 2003 Int. Workshop, vol. 284, pp. 81–100. UNU/IIST Technical Report (2003)
16. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. on Softw. Eng. 21(10), 785–798 (1995)
17. Amálio, N., Stepney, S., Polack, F.: A formal template language enabling metaproof. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 252–267. Springer, Heidelberg (2006)
18. Stepney, S., Polack, F.A.C., Turner, H.R.: Engineering emergence. In: ICECCS 2006, pp. 89–97. IEEE Computer Society, Los Alamitos (2006)
19. Liu, J., Tsui, K.: Toward nature-inspired computing. Commun. ACM 49(10), 59–64 (2006)

Integrating Z into Large Projects
Tools and Techniques

Anthony Hall

If we want to use Z to write an overall system specification, we need to integrate it into a rich set of documents written in natural language and domain-specific notations. These documents must be easy to write and read by non-mathematicians.

On a purely practical level, this implies that we want Z to be part of the ordinary documents that are used every day on the project. That means, in practice, that it has to be integrated into Microsoft Word. I describe a tool for writing and checking Z within the Word environment and some progress towards a process for writing the specification and guidelines for its structure.

The de facto standard for writing Z is the LaTeX mark up first introduced by Spivey and now incorporated into the Z standard. Industry, however, does not use LaTeX: it uses Microsoft Word. I have developed, with help from several colleagues, a package of Z tools for Word. This is in day to day use on a large project and is being made freely available. For more information see http://ZWTools.anthonyhall.org

The tool includes:

1. styles for laying out schemas and other Z paragraphs;
2. a Unicode font that includes all the Z symbols;
3. automatic layout of Z paragraphs like the LaTeX equivalent;
4. the ability to enter symbols from a palette or typing in the markup;
5. one-click typechecking, with errors highlighted in the Word document;
6. use before declaration and and specifications distributed over several documents;
7. generation of indexes and cross-references to definition and use of Z names;
8. the ability to hide the Z so the document can be used by non—Z readers;
9. miscellaneous tools such as checking matching brackets.

The tool currently uses Mike Spivey's fuzz as the typechecking engine, but the intention is to open it to other tools and hence to support the Z standard.

It is crucial to realise that in a Z specification, the mathematics is subsidiary to the natural language. A piece of mathematics makes no sense unless we know the intended meaning of each construct. We therefore enforce a rule that an English description must precede the corresponding Z and should be of about the same length. The English and the Z are complementary: the English describes the relevant real-world concept and explains the meaning of every term in the maths; the maths makes precise the relationships between the terms defined in the English. We expect there to be as much informal text and diagrams in the document as there is mathematics. Note in particular that there is no rule like "In case of a discrepancy between English and Z, the Z takes precedence": rather, the rule is that such a discrepancy is an error which must be corrected.

In the full version of this paper I show examples of this style in practice, based on experience in a real project. It is still difficult to write specification documents that are accessible to all stakeholders, but we have made significant progress in integrating formality into large scale project documents.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 337, 2008.

A First Attempt to Express KAOS Refinement Patterns with Event B

Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau

LACL, Université Paris-Est
{abderrahman.matoussi,frederic.gervais,laleau}@univ-paris12.fr

It is now recognised that goals play an important role in requirements engineering process, and consequently in systems development process. Whereas specifications allow us to answer the question "WHAT the system does", goals allow us to address the "WHY, WHO, WHEN" questions [1]. Up to now, the development process associated with formal methods, including Event B, begins at the specification level. Our objective is to include requirements analysis within this process, and more precisely KAOS [2] which is a methodology to implement goal-based reasoning. Existing work [3,4] that combine KAOS with formal methods generate a formal specification model from a KAOS requirements model. We aim at expressing KAOS goal models with a formal language (Event B), hence staying at the same abstraction level. Our work is based on a constructive approach in which Event B models are built incrementally from KAOS goal models, driven by goal refinement patterns [1]. Since a KAOS goal means that a property must be established, the main idea is to represent each goal as a B event and the property as the post-condition of this B event. Up to now, we consider refinement patterns defined with first-order logic. Patterns with LTL temporal logic will be studied in further work. Thus, the general form of the assertion associated to a goal G is $P \rightarrow Q$ (P and Q are predicates, $\rightarrow$ is the logical implication). The *THEN* part of the B event corresponding to G is the translation of this assertion into Event B. At the most abstract level, the guard of the event related to the parent goal is always set to *True* to express that the event is always feasible. The definitive guard is built during the refinement process; i.e. after processing the different sub-goals. Proof obligations of Event B allow most of the KAOS refinement conditions to be verified. However, for some KAOS patterns as the case-driven tactics [2], additional constraints must be identified. Our current work is still partial and we are working on its extensions.

References

1. Darimont, R., van Lamsweerde, A.: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In: SIGSOFT 1996, San Francisco, California, USA, October 1996, pp. 179–190. ACM SIGSOFT, New York (1996)
2. Letier, E.: Reasoning About Agents in Goal-Oriented Requirements Engineering. Ph.D. Thesis (2001), ftp://ftp.info.ucl.ac.be/pub/thesis/letier.pdf
3. Nakagawa, H., Taguchi, K., Honiden, S.: Formal specification generator for KAOS. In: ASE 2007, Atlanta, USA, November 2007, pp. 531–532. ACM, New York (2007)
4. Ponsard, C., Dieul, E.: From Requirements Models to Formal Specifications in B. In: REMO2V 2006, Luxembourg (June 2006)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 338, 2008.

Verification and Validation of Web Service Composition Using Event B Method

Idir Ait-Sadoune and Yamine Ait-Ameur

LISI/ENSMA - Université de Poitiers
Téléport 2 - 1, avenue Clément Ader - B.P. 40109
86960 Futuroscope Cedex - France
{idir.aitsadoune,yamine}@ensma.fr

The Service-Oriented Architecture based on the Web service technology emerged as a consequence of the evolution of distributed computing. One of the key ideas of this technology is the ability to create service compositions by combining and interacting with pre-exisiting services. A service is implemented, described[1], and published by a service provider in a UDDI[2] registry. The service composition is referred to an executable process that interacts with other services accomplishing its functional goal. *Orchestration* and *Choreography*[3] are the processes that allow to schedule the defined services compositions and messages exchanges. There is a wide range of industrial standardization efforts towards providing specification languages for the Web service composition. Among them BPEL (Business Process Execution Language[4]) is the most known and used orchestration language. Our work addresses the composition expressed by the orchestration and its support language BPEL. BPEL allows the designer to represent service compositions by various behavioral properties like services interactions (message exchanges), control flow constraints (sequence, iteration, conditional) or data flow constraints (exchange, modification, evaluation of data expressions).

Our work focuses on the formal verification of the composition of web services. We study the verification and validation of behavioral requirements through, the properties that a services composition shall satisfy in order to achieve its functional goal. These requirements include deadlock freeness, correct manipulation and transformation of data, obeying to rules and to constraints on interactions ordering and termination. This verification is not supported by BPEL like languages although there exists several operational orchestration tools like Orchestra[5] that encode and interpret this language.

We propose to address the problem of services composition verification using proof and refinement based techniques with the event B method. Our approach consists in extracting an event B model from service compositions written in BPEL. Thereafter, the obtained model is enriched with the relevant properties in the INVARIANTS and THEOREMS clauses and events guards. Two development scenarios have been studied.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 339–340, 2008.

References

1. Booth, D., Liu, C.K.: Web Services Description Language Version 2.0. Technical report, W3C Recommendation (June 26, 2007), http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/
2. OASIS: Universal Description, Discovery, and Integration Specification (2003), http://uddi.xml.org/
3. Peltz, C.: Web services orchestration and choreography. Web Services Journal (July 2003), http://www.wsj2.com
4. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Standard (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
5. BSOA: Orchestra v3.0. Technical report, BULL CEDOC (November 2006), http://orchestra.objectweb.org

Stability of Real-Time Abstract State Machines under Desynchronization

J. Cohen and A. Slissenko*

Laboratory for Algorithmics, Complexity and Logic (LACL),
University Paris-East (Paris 12)
61 Av. du Gn. de Gaulle, 94010, Crteil, France
{j.cohen,slissenko}@univ-paris12.fr

Introduction. In our paper TR-LACL-2008–02 (www.univ-paris12.fr/lacl/) we give sufficient conditions that permit to implement a real-time ASM with instantaneous actions (IA-ASM) by an ASM with delayed actions (DA-ASM) with approximate bisimulation of runs. The time is continuous and time constraints are linear inequalities with rational coefficients. As IA-ASM we consider ASM whose programs are blocks of **if** *guard* **then** *blockOfUpdates*. The implementation is an ASM of more general type. It works by 2 phases: backup phase memorizes the values of functions, and update phase makes the updates using the backed up values. Such an implementation implies shifts of time instants and, consequently, of the values of the real-valued functions. The approximation of runs (and, thus approximate bisimulation) is determined by 2 positive parameters (ε, η), where ε bounds time shifts, and η bounds the deviations of real-valued functions. We introduce a notion of (ε, η)-sturdy IA-ASM, and prove that the implementation of any such IA-ASM gives an DA-ASM with (ε, η)-approximately bisimular runs if the delay satisfies some constraints. An interesting point is that the sources of desynchronization that destroy the bisimulation are much more subtle and numerous than one can think a priori. Another conceptual consequence concerns the adequacy of the notion of IA-ASM that was introduced in Gurevich–Huggins (LNCS, vol. 1092, 1996), and later studied in Beauquier–Slissenko, (APAL, 113(1–3):13–52, 2002) for the specification of real-time system.

Our Work in Progress. We study one more question of this kind. Given a general multi-agent ASM with delayed actions, under what conditions its desynchronization gives an approximately bisimular set of runs?

Let $\mathcal{A}$ be such ASM, whose each agent programm is constructed using updates, branching, sequential and parallel composition. Each agent has only external default loop. A delay interval $d_{\mathcal{A}}(X)$ is attributed to each occurrence X of update or of guard of $\mathcal{A}$. When a run arrives at t at the evaluation or execution of X then this action is accomplished by an instant T that is chosen non-deterministically in $t + d_{\mathcal{A}}(X)$.

Given an ASM $\mathcal{A}_0$, its ξ-*desynchronization* $\mathcal{A}_1$ is an ASM with the same program whose delays are ξ-close to the delays of $\mathcal{A}_0$ but bigger.

We give constraints on $\mathcal{A}_0$ and ξ of the same flavor as above, that guarantee that $\mathcal{A}_0$ and $\mathcal{A}_1$ are (ε, η)-bisimular.

* Member of Scholars Club of Saint-Petersburg Division of Steklov Mathematical Institute, Russian Academy of Sciences.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 341, 2008.

XML Database Transformations with Tree Updates

Qing Wang[1], Klaus-Dieter Schewe[2], and Bernhard Thalheim[3]

[1] Massey University, New Zealand
q.q.wang@massey.ac.nz
[2] Information Science Research Centre, Palmerston North, New Zealand
k-d.schewe@xtra.co.nz
[3] Institute of Computer Science, CAU Kiel, Olshausenstr. 40, Kiel, Germany
thalheim@is.informatik.uni-kiel.de

For many years the eXtensible Markup Language (XML) has attracted much research attention from database communities, particularly in the area of query and transformation languages such as XQuery and XSLT. XML documents are usually represented as trees. In order to accommodate the diversity of user requirements, it is desirable to conduct transformations on XML trees at flexible abstraction levels. However, most of current approaches have a fixed abstraction level at which updates must be identified for individual nodes and edges. In this paper we study XML database transformations with structured updates, for example, manipulations on portions of a tree, including deleting, modifying or inserting subtrees, copying contexts, etc, by using Abstract State Machines (ASMs) as it has turned out in [3] to be a universal computation model capturing database transformations.

In this setting, the problem of partial updates [1] will come up again. Essentially, partial updates root in two factors: complex objects and parallel computing. When several parallel computations are executing updates on partial parts of the same complex object, inconsistency of an update set might arise. As trees are typically a kind of complex objects, we believe that XML database transformations provide an interesting paradigm for the study towards partial updates.

The study is carried out under an algebraical framework for XML trees extended from tree algebras by [4,2]. Updates on existing and output XML trees are manipulated in a unifying and parallel manner, and thus the consistency checking on a collection of updates plays a key role in computations.

References

1. Gurevich, Y., Tillmann, N.: Partial updates. Theor. Comput. Sci. 336(2-3), 311–342 (2005)
2. Walukiewicz, I., Bojanczyk, M.: Forest algebras. In: Flum, J., Graedel, E., Wilke, T. (eds.) Logic and Automata. Amsterdam University Press (2007)
3. Wang, Q., Schewe, K.-D.: Axiomatization of database transformations. In: Proceedings of the ASM 2007: The 14th International ASM Workshop (2007)
4. Wilke, T.: An algebraic characterization of frontier testable tree languages. Theor. Comput. Sci. 154(1), 85–106 (1996)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 342, 2008.

Dynamic Resource Configuration & Management for Distributed Information Fusion in Maritime Surveillance

Roozbeh Farahbod and Uwe Glässer

Software Technology Lab, Simon Fraser University, Burnaby, B.C., Canada
{roozbehf,glaesser}@cs.sfu.ca

We propose a highly adaptive and auto-configurable, multi-layer network architecture for distributed information fusion to address large volume surveillance challenges, assuming a multitude of different sensor types on multiple mobile platforms for intelligence, surveillance and reconnaissance. Our focus is on network enabled operations to efficiently manage and improve employment of a set of mobile resources, their information fusion engines and networking capabilities under dynamically changing and essentially unpredictable conditions. Building on realistic application scenarios adopted from the design and development of the CanCoastWatch system [1], we contend that distributed system concepts based on decentralized control mechanisms are crucial for the design of robust and scalable network enabled operations for several reasons.

A high-level model of our network architecture, called *Dynamic Resource Configuration & Management Architecture* (DRCMA) [2], is described in abstract functional and operational terms based on a multi-agent modeling paradigm using the *Abstract State Machine* (ASM) formalism [3]. This description of the underlying design concepts provides a concise yet precise blueprint for reasoning about key system attributes at an intuitive level of understanding, supporting requirements specification, design analysis, validation and, where appropriate, formal verification of system properties prior to actually building the system. Additionally, by building on the CoreASM tool environment (see www.coreasm.org), we also illustrate how to use the ASM formalism and underlying abstraction principles for rapid prototyping of a high-level executable DRCMA model. The result will be a prototype for testing, experimental validation and machine-assisted verification of the key system attributes prior to actually building the system.

References

1. Farahbod, R., Glässer, U., Wehn, H.: CanCoastWatch Dynamic Configuration Manager. In: Proc. of the 14th Int'l Abstract State Machines Workshop (2007)
2. Farahbod, R., Glässer, U., Wehn, H.: Dynamic resource management for adaptive distributed information fusion in large volume surveillance. In: Proc. of SPIE Defense & Security Symposium (2008)
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 343, 2008.

UML-B: A Plug-in for the Event-B Tool Set*

Colin Snook and Michael Butler

University of Southampton,
United Kingdom
{cfs,mjb}@ecs.soton.ac.uk

UML-B is a graphical formal modelling notation that relies on Event-B for its underlying semantics and is closely integrated with the 'Rodin', Event-B verification tools. UML-B is similar to UML but has its own meta-model. UML-B provides tool support, including drawing tools and a translator to generate Event-B models. When a UML-B drawing is saved the translator automatically generates the corresponding Event-B model. The Event-B verification tools (syntax checker and prover) then run automatically providing an immediate display of problems which are indicated on the relevant UML-B diagram. The UML-B modelling environment consists of a UML-B project containing a UML-B model. Four interlinked diagram types (package, context, class and statemachine) are available. Package Diagrams are used to describe the 'refines' and 'sees' relationships between top level components (machines and contexts) of a UML-B project. UML-B mirrors the Event-B approach where static data (sets and constants) are modelled in a separate package called a *'context'*. The context diagram is similar to a class diagram but has only constant data represented by *ClassTypes*, *Attributes* and *Associations*. ClassTypes define 'carrier' sets or constant subsets of other ClassTypes. ClassTypes may own immutable attributes and associations which represent constant functions. The behavioural parts (variables and events) are modelled in a Class diagram which is used to describe the *'machine'*. Classes represent subsets of the ClassTypes that were introduced in the context. The class' associations and attributes are similar to those in the context but represent variables instead of constants. Classes may own *events* that modify the variables. Event *parameters* can be added to an event, providing local variables to be used in the transition's guards and actions. Class events utilise a parameter, *self*, to non-deterministically select the affected instance of the class. State machines may be used to model behaviour. Transitions represent events with implicit behaviour associated with the change of state. Additional guards and actions can be attached to the transition. UML-B retains sufficient commonality with UML for the main goals of approachability to be attained by industrial users. Since UML-B automates the production of many lines of textual B, models are quicker to produce and hence exploration of a problem domain is more attractive. This assists novices in finding useful abstractions for their models. We have found that the efficiency of UML-B and its ability to divide and contextualise mathematical expressions assists novices who would otherwise be deterred from writing formal specifications. UML-B is also a useful visual aid for more experienced formal methods users.

* This work was carried out under the EU projects, Rodin [IST-511599] and ICT project Deploy [IP-214158].

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 344, 2008.

BART: A Tool for Automatic Refinement*

Antoine Requet

ClearSy
Parc de la Duranne
320, avenue Archimde
Les Pliades III - Bt A
13857 AIX EN PROVENCE CEDEX 3 - France
antoine.requet@clearsy.com

1 Extended Abstract

Refining a B specification into an implementation can be a complex and time consuming process. This process can usually be separated in two distinct parts: the specification part, where the refinement is used to introduce new properties and specification details, and the implementation, where refinement is used to convert a detailed B specification into a B0 implementation. The first part requires human interaction, since it corresponds to writing the specification. However, the implementation part is more mechanical, and usually corresponds to apply known refinement schemes.

The BART tool aims to provide helps for this second part of the B development, by automatically refining machines or refinements to B0 implementations.

To refine a specification, the tool uses rules describing refinement patterns using pattern matching. Those rules allows the refinement of both data and algorithm: abstract variables are refined to concrete variables, and the substitutions used within the abstract machines are refined into equivalent B0 substitutions.

A set of default refinement rules is provided with the tool, however, users can also write new refinement rules to handle more complex refinements. Rules can be customized safely, as the proof of the generated machines still has to be performed. So, an incorrect refinement rule will lead to an incorrect refinement, which will be detected during the proof.

Using an automatic refinement tool provides several benfits: the most obvious is that it automatises repetitive tasks. However, it is also a way of reusing refinement patterns, and capitalizing on refinement experience, and can also simplify the proof of the generated components by using well-known refinement patterns. The BART tool is currently in development and will be integrated in the next major version of Atelier B.

* This work has been funded by french "Agence Nationale de la Recherche" ANR-06-SETI-015.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 345, 2008.

Model Checking Event-B by Encoding into Alloy
(Extended Abstract)*

Paulo J. Matos and João Marques-Silva

Electronics and Computer Science, University of Southampton
{pocm,jpms}@ecs.soton.ac.uk

Current day systems are ever more detailed and complex leading to the necessity of developing models that abstract unimportant implementation details while emphasizing their structure. Until recently it was only possible to perform temporal model checking in an Event-B model by converting the model to B-method and then using ProB [1]. More recently, a prototype ProB plugin [2] for the RODIN tool has been developed. Nevertheles, encoding Event-B to Alloy allows building on top of the Alloy model finding engine therefore benefiting from all of its optimizations. An extended version of this work is in [3].

There are three aspects to the encoding: encoding of model structures, expressions, and predicates (which are straightforward). The execution model needs to be emulated by the final Alloy model. A signature "State" keeps track of all the state variables that are ordered in time using the ordering module. Events are predicates and facts define not only the initial state but also that one event is triggered per state. Expressions are the hardest part to encode. There is not only a myriad of complex expressions in Event-B but given that Alloy uses only flat relations, some Event-B expressions that introduce relations with nested sets generate many Alloy expressions. Some expressions are straightforward as they have Alloy counterparts, others need to be defined by small functions. Function expressions are encoded as relations and then facts can be added to the model as to assure the semantics is preserved.

The motivation for our work is to allow users of the Event-B language to exploit the accumulated experience from the development of the Alloy tools. The resulting Alloy model can serve to find counterexamples to false invariants and translate them back to Event-B. Future work entails the automatic generation of the encoding and its integration with the RODIN platform. The tool to be developed can then be extended to use other backends besides Alloy.

References

1. Leuschel, M., Butler, M.J.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
2. Ligot, O., Bendisposto, J., Leuschel, M.: Debugging Event-B Models using the ProB Disprover Plug-in. In: Proceedings of AFADL 2007 (June 2007)
3. Matos, P.J., Marques-Silva, J.: Model checking Event-B by encoding into Alloy. Computing Research Repository abs/0805.3256 (May 2008)

* This work is partially supported by EPSRC grant EP/E012973/1, and by EU grants IST/033709 and ICT/217069.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 346, 2008.

A Roadmap for the Rodin Toolset*

Jean-Raymond Abrial[1], Michael Butler[2],
Stefan Hallerstede[2], and Laurent Voisin[3]

[1] ETH Zurich, Switzerland
jabrial@inf.ethz.ch
[2] University of Southampton, United Kingdom
{mjb,sth}@ecs.soton.ac.uk
[3] Systerel, France
laurent.voisin@systerel.fr

Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.

The Rodin Platform[1] is an Eclipse-based toolset for Event-B that provides effective support for refinement and mathematical proof. Key aspects of the tool include support for abstract modelling in Event-B, support for refinement proof, extensibility of the functionality and open source development. To support modelling and refinement proofs Rodin contains a modelling database surrounded by various plug-ins: a static checker, a proof obligation generator, automated and interactive provers. The extensibility of the platform has allowed for the integration of various plug-ins such as a model-checker (ProB), animators, a UML-B transformer and a LaTeX generator. The database approach provides great flexibility, allowing the tool to be extended and adapted easily. It also facilitates incremental development and analysis of models. The platform is open source, contributes to the Eclipse framework and uses the Eclipse extension mechanisms to enable the integration of plug-ins.

In its present form, Rodin provides a powerful and effective toolset for Event-B development and it has been validated by means of numerous medium-sized case studies. Naturally further improvements and extensions are required in order to improve the productivity of users further and in order to scale the application of the toolset to large industrial-scale developments. A roadmap has been produced which outlines the planned extensions to the Rodin toolset over the coming years. The roadmap[1] covers the following issues: model construction; composition and decomposition; team-based development; extending proof obligations and mathematical language; proof and model checking; animation; requirements handling and traceability; document management; automated model generation.

* The continued development of the Rodin toolset is funded by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu. The toolset was originally developed as part of the project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

[1] Available from www.event-b.org

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 347, 2008.

Exploiting the ASM Method for Validation & Verification of Embedded Systems*

A. Gargantini[1], E. Riccobene[2], and P. Scandurra[2]

[1] DIIMM, Università di Bergamo, Italy
[2] DTI, Università di Milano, Italy

SystemC (built upon C++) [2] is an IEEE industry-standard language for *system-level* models, specifically targeted at architectural, algorithmic, transaction-level modelling. Recently, a further improvement has been achieved by trying to combine SystemC with lightweight software modelling languages like UML to describe system specifications. In accordance with the design principles of the OMG's *Model-driven architecture* (MDA), we defined a model-driven design methodology for embedded systems [3] based on the UML 2, a SystemC UML profile (for the HW side), and a multi-thread C UML profile (for the SW side), which allows UML modelling of the system at higher levels of abstraction (from a functional level down to RTL level).

Currently, we are working on complementing this methodology with a *formal analysis process* for high level system validation and verification (V&V) which involves the Abstract State Machine (ASM) formal method.

The V&V toolset we are building is based on the ASMETA toolset [1]. The analysis process starts with the automatic mapping of the SystemC-UML model of the system into a corresponding ASM model (written in AsmetaL), which provides the basis for analysis. Several activities can be then executed in parallel: (a) simulation of ASM models by AsmetaS; (b) scenario-based validation allowing the designer to describe possible behaviours of the system as scenarios and test them within the AsmetaV validator; (c) automatic test-case generation from the model by the ATGT tool; (d) conformance testing of the implementations with respect to their specification by transforming test and/or validation scenarios in SystemC test cases. We plan to support formal verification by model checking. This requires transforming ASM models into inputs for model checkers, for example SPIN, and specifying the desired properties in temporal logic.

We have been testing our analysis methodology on case studies taken from the standard SystemC distribution. Thanks to the ease in raising the abstraction level using ASMs, we believe our approach scales effectively to industrial systems.

References

1. The ASMETA toolset (2006), `http://asmeta.sf.net/`
2. The Open SystemC Initiative, `http://www.systemc.org`
3. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A Model-driven co-design flow for Embedded Systems. In: Advances in Design and Specification Languages for Embedded Systems (Best of FDL 2006) (2007)

* This work is partially supported by STMicroelectronics.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 348, 2008.

Tool Support for the *Circus* Refinement Calculus

A.C. Gurgel, C.G. de Castro, and M.V.M. Oliveira

Departamento de Informática e Matemática Aplicada, UFRN, Brazil

Circus combine both data and behavioural aspects of concurrent systems using a combination of CSP, Z, and Dijkstra's command language. Its associated refinement theory and calculus distinguishes itself from other such combinations. Using a refinement calculus, we can correctly construct programs in a stepwise fashion. Each step is justified by the application of a refinement law, possibly with the discharge of proof obligations (hereafter called POs). The manual application of the refinement calculus, however, is an error-prone and hard task.

We present CRefine[1], tool that supports the use of the *Circus* refinement calculus. It is a considerable extension to an earlier prototype. First, we updated the *Circus* parser to fix a couple of bugs of its earlier version. We have also added facilities to manage developments like undoing and redoing refinement steps, and saving and opening developments. Furthermore, GUI facilities like pretty-printing, filtering applicable laws according to the selected program, classification of laws, adding comments, and printing the development are also available. Finally, CRefine automatically discharges some POs. In our experience, they represent over 60% of the POs generated in a development.

CRefine's interface contains a menu and three main frames: refinement, proof obligations, and code. The refinement frame shows all the steps of the refinement process. This includes law applications and retrieving the current status of an action or process (collection). The proof obligations frame lists the POs that were generated by the law applications, indicates their current state (i.e. checked valid or invalid, or unchecked), and associates each one of them to the law application that originated it in the refinement frame. Finally, the code frame exhibits the overall *Circus* specification that has been calculate so far.

The majority of the *Circus* refinement laws proposed so far are included in CRefine. However, we intend to use a specific parser for refinement laws to dynamically load them. It is also in our agenda to allow CRefine users to define and use tactics of refinement within CRefine bringing a profit in effort. In a near future, users will also be able to modularise their development by making sub-developments within larger developments. The most interesting piece of future work is the automatic (or iterative) discharge of the POs that are not automatically discharged. For that, we will integrate CRefine with a theorem-prover.

CRefine can be a useful tool in the development of state-rich reactive systems. Our initial intention was to develop an educational tool and use it in teaching formal methods. However, during the implementation and tests, we noticed that it may as well be useful in the development of industrial-scale systems. Empirical verifications in a near future will verify this statement. For instance, we are currently developing case studies that are related to the oil industry.

[1] This work is financially supported by CNPq: grant 550946/2007-1.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 349, 2008.

Separation of Z Operations

Ramsay Taylor

Dept.of Computer Science, Regent Court, University of Sheffield, S1 4DP, UK
ramsay@dcs.shef.ac.uk

1 Introduction

Machine code and assembly language programs are structured using various branches and decision points, but between these they contain blocks of instructions that are simply sequentially composed. Most work on formal program analysis has focused on the behavior of the branch points — primarily because composing the blocks of sequential code to determine their overal effect on the system is often intellectually trivial. This processs is also computationaly simple, but it is not computationally trivial. The aim of this work is to produce a system of rules that can be efficiently implemented and allow us to determine the overal behaviour of sequentially composed operations.

To identify those sequential compositions that are trivial we will use techniques inspired by Separation logic[2, 1]. Separation logic itself is a very general, abstract collection of higher order logic statements but the simple observation at the heart of separation logic will be used: if two operations refer to completely disjoint parts of the state space they can be reasoned about independently.

Here we will not present anything with the generality and elegance of separation logic. Nor will we present a complete solution to analysing sequential composition in Z. The aim is to present some techniques that are very easy to implement and that will identify those operation compositions that are trivial. These can be processed syntactically, before a more serious theorem proover is applied.

The approach taken is in the spirit of separation logic. If two operations are sequentially composed but it can be shown that the effects of the first in no way influence the effects of the second then the effect of the composition is just the syntactic combination of the two — a new schema for the composition can be created by concatenating all the declarations and predicates together after some very simple pruning.

References

[1] Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Proceedings of POPL 2001 (2001)
[2] Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare (1999)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 350, 2008.

BSmart: A Tool for the Development of Java Card Applications with the B Method

D. Déharbe, B.E.G. Gomes, and A.M. Moreira

Federal University of Rio Grande do Norte; Natal, RN; Brazil
{david,bruno,anamaria}@consiste.dimap.ufrn.br

A smart card is a portable computer device able to store data and execute commands. Java Card [1] is a specialization of Java, providing vendor inter-operability for smart cards, and has now reached a *de facto* standard status in this industry. The strategic importance of this market and the requirement for a high reliability motivate the use of rigorous software development processes for smart card aware applications based on the Java Card technology. The B method [2] is a good candidate for such process, since it is a formal method with a successful record to address industrial-level software development. In [3,4], we proposed two versions of a Java Card software development method (called BSmart) based on the B method. The main feature of these methods is to abstract the particularities of smart card systems to the applications developers as much as possible. This abstract presents the current version of a tool, also called BSmart, to support the method. The tool provides the automatable steps required by the method and some guidelines and *library machines* that are useful during the development process. It includes B development tools (type checker, PO generator) and specific BSmart tools (refinement generator, Java Card translator, etc.). In this approach, the card services specifier only needs to apply some refinement steps to his abstract (implementation platform independent) B specification. The generation of these refinements adapts the specification to Java Card standards and introduces platform specific aspects gradually. Finally, from the concrete B refinements the Java Card code implementing the services provided by the card will be automatically generated by the tool. The tool also provides the generation of a Java API for the host-side application from the original abstract specification, encapsulating all the communication protocol details. Thus, the client application code can then be developed in a completely platform independent way. The definition of the method is in a mature stage, and our attention is now focused on the implementation of more robust versions of the BSmart tools and packaging them in a user-friendly environment. The integration of verification and animation tools is also planned for a next release of the tool.

References

1. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Reading (2000)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University (1996)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 351–352, 2008.

3. Gomes, B., Moreira, A.M., Déharbe, D.: Developing Java Card applications with B. In: SBMF, pp. 63–77 (2005)
4. Deharbe, D., Gomes, B.G., Moreira, A.M.: Automation of Java Card component development using the B method. In: ICECCS, pp. 259–268. IEEE Comp. Soc., Los Alamitos (2006)

From ABZ to Cryptography
(Abstract)

Eerke A. Boiten

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK
E.A.Boiten@kent.ac.uk

Three Steps from the Ideal

Ideally correctness is by construction; post-hoc verification is second choice; verification of *proofs* is the next step down. In the application area of *modern cryptographic protocol verification*, the latter would be viewed as serious progress.

Modern Cryptographic Protocols and Security

A modern cryptographic protocol may have the following properties:

- its functionality is clear, but its security definition incomplete;
- it contains explicit probabilistic elements;
- its notion of security (correctness) is approximate, and relative to computational resources available for an attack against it;
- its security is proved relative to some problem being hard;
- primitives cannot be implemented compositionally.

All this means that the standard techniques and good intentions of formal methods do not work straight out of the box. Many approaches to bridging the gap between formal methods and modern cryptography exist – but none of these are too close in spirit to the ABZ world.

An ABZ Bottom-Up Approach

We would need to extend our toolbox in a few dimensions:

approximation. Notions of correctness which are not exact but "close enough".
probability. In formalisms, and in notions of refinement.
action refinement. From single step world change to communication protocols.
attacks. Multiple instances of protocols, attacks, computationally bounded?
compositionality. Of cryptographic primitives, to allow algebraic reasoning.

All of this makes up a large research agenda to chip away at. Watch this space for a planned new EPSRC Network and new research in several of these areas.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 353, 2008.

Using ASM to Achieve Executability within a Family of DSL

Ileana Ober and Ali Abou Dib

IRIT – Université Paul Sabatier Toulouse
118, route de Narbonne 3 1062 Toulouse- France
{Ileana.Ober,aboudib}@irit.fr

Abstract. We propose an approach to achieve interoperability in a family of domain specific language based on the use of their ASM semantics and of the category theory. The approach is based on the construction of a unifying language of the family, by using categorical colimits. Since the unifying language is obtained by construction, translators to this one are obtained easily. These are the premises for using ASM tools for symbolically executing systems made of components specified in domain specific languages of a same family.

Our work starts from a case study that we developed with colleagues from the French Space Agency (CNES). This case study revealed their need to deal with a set of related – yet different – domain specific languages in remotely controlled satellites. Here, one main challenge is to handle the heterogeneity. Our thesis is that the interoperability within a family of DSLs can be rigorously tackled, by using a categorical approach. Classical results in category theory, allow us to obtain *by construction* the formal semantics of a *unification language* as well as *translators* from the source DSLs to this *unification language*. Moreover, *properties* established in the context of DSLs and expressed as invariants, pre-conditions or post-conditions in the algebraic structure can be *transferred* to the unification language.

In order to get to a framework with symbolic execution, we apply a similar approach on the ASM specifications of programs specified in DSLs of a same family of languages. On the theoretical side, the set of ASM specifications of the languages in the family does not lead to the category of algebraic specifications. Therefore, we have to identify a good category on which to reason on. Existing results on *especs* show that it is possible to use categorical results and existing tools based on categories, with specifications using state machines.

We started on the practical side, by doing small experiments consisting in translating by hand ASM toy specifications in CoreASM, corresponding to specifications in source DSLs into the algebraic form accepted by Specware, in order to calculate pushouts for unifying them.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 354, 2008.

Using Satisfiability Modulo Theories to Analyze Abstract State Machines (Abstract)

Margus Veanes[1] and Ando Saabas[2]

[1] Microsoft Research, Redmond, WA, USA
margus@microsoft.com
[2] Institute of Cybernetics
Tallinn University of Technology, Tallinn, Estonia
ando@cs.ioc.ee

We look at a fragment of ASMs used to model protocol-like aspects of software systems. Such models are used industrially as part of documentation and oracles in model-based testing of application-level network protocols. Correctness assumptions about the model are often expressed through state invariants. An important problem is to validate the model prior to its use as an oracle. We discuss a technique of using Satisfiability Modulo Theories or SMT to perform bounded reachability analysis of such models. We use the Z3 solver for our implementation and we use AsmL as the modeling language.

Protocols are abundant; we rely on the reliable sending and receiving of email, multimedia, and business data. But protocols, such as the Windows network file protocol SMB (Server Message Block), can be very complex and hard to get right. Model programs have proven to be a useful way to model the behavior of such protocols and it is an emerging practice in the software industry to use model programs for documentation and behavioral specification of such protocols[1] [2], so that different vendors understand the same protocol in the same way. The step semantics of model programs is based on the theory of ASMs with a rich background universe. Correctness assumptions about the model are often expressed through state invariants. It is important that the model is validated before it is used as a specification or an oracle. We describe a technique [3] of using satisfiability modulo theories or SMT to perform bounded reachability analysis of a fragment of model programs and extend the work through improved handling of quantifier elimination and extended support for background axioms. We use the SMT solver Z3[4] and we use AsmL as the modeling language.

[1] W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of windows protocol documentation. In *First International Conference on Software Testing, Verification and Validation, ICST*, Lillehammer, Norway, April 2008.

[2] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[3] M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE'08*, LNCS. Springer, 2008.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 355, 2008.

Formal Verification of ASM Models Using TLA$^+$

Hocine El-Habib Daho and Djilali Benhamamouch

Department of Informatics, University of Oran, Algeria
dahoh@yahoo.com

Abstract. The notion of Abstract State Machines(ASMs) handles a practical new approach for modeling and analysing various kinds of discrete dynamic systems. In the context of the verification problem of ASM models, formal verification techniques based on variants of restricted first-order temporal logic have been used to verify correctness of restricted forms of ASM specifications. In this spirit, the current work shows how the state-based logic of TLA$^+$can be employed to formally reason about dynamic systems formalised in terms of ASMs.

Reasoning about ASMs within the TLA$^+$-Logical Framework. In our ongoing research work[2], we propose to adopt Lamport's Temporal Logic of Actions (TLA)[3] to formally reason about ASM specifications of dynamic systems[1]. TLA is a state-based logic which provides the means for describing transition systems and formulating their properties in a single logical formalism, equipped with a set of proof rules for reasoning about safety and liveness properties. The operational behavior of an ASM specification is directly defined by TLA-formulas and the TLA-proof techniques can be applied to formally prove the correctness of ASM specifications. In particular, we provide some basic rules to translate ASM models into TLA$^+$specifications. TLA$^+$is a formal specification language based on Zermelo-Fränkel set theory, first-order logic and the linear-time temporal logic TLA[3]. The TLA$^+$ framework offers a potential mathematical framework into which ASM model elements are directly translated to their most natural equivalents in TLA$^+$. The applicability of the proposed TLA$^+$approach is illustrated by the formal correctness proofs of both Lamport's bakery algorithm and a token ring algorithm both formalised in terms of ASMs[2]. Futur work will concentrate on the development of a model translator, namely ASM2TLA$^+$ translator, to perform the translation of an ASM model into a TLA$^+$model which can be verified automatically using the TLA$^+$model checker called TLC[3].

References

1. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
2. El-Habib Daho, H., Benhamamouch, D.: Verifying the correctness of ASM Programs using TLA$^+$. Technical Report, Department of Informatics (January 2008)
3. Lamport, L.: Specifying Systems: The TLA$^+$Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2003)

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 356, 2008.

DIR 41 Case Study

How Event-B Can Improve an Industrial System Specification

Christophe Metayer[1] and Mathieu Clabaut[2]

[1] Systerel
christophe.metayer@systerel.fr
[2] Systerel
mathieu.clabaut@systerel.fr

1 Context and Goals

Some RATP[1] units are responsible for evolution and maintenance of an automated interlocking specification document.

In order to improve their paper and pencil process, RATP asked Systerel if Event-B could be useful to them. An eight month study was launched whose main goal was to help RATP improving their confidence in their interlocking specification, by applying an Event-B approach on rewriting their requirement document.

2 Description of Work and Results

The work was planned to be highly iterative and composed of the following tasks:
Requirement specification rewriting with the organization and the wording of our choice. This document was to be approved by the domain experts and to serve as a reference for the modeling task.
Refinement plan design
Modeling the system and proving it correct.

The achieved model contains fifteen levels of refinement and allowed us to exhibit four potential safety flaws (with an expected low probability of occurrence) and many implicit hypotheses.

Several paths of improvement emerge from this study. They range from modeling process and techniques to the way of involving domain experts in the whole process.

3 Conclusion

Modeling a system with Event-B proved to be very interesting for pointing out potential safety flaws and for capturing and proving the global safety rationale.

This way of modeling allows a B expert with little knowledge in an industrial domain to quickly grasp the domain core concepts. It is still very difficult to involve the domain experts in the whole process and we have high expectations that model animation would improve this.

[1] French organization in charge of Paris transportation, which supported this study.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 357, 2008.

FDIR Architectures for Autonomous Spacecraft: Specification and Assessment with Event-B

Jean-Charles Chaudemar[1], Charles Castel[2], and Christel Seguin[2]

[1] ISAE-DMIA, Toulouse, France
[2] ONERA-DCSD, Toulouse, France

Abstract. On-board Fault Detection, Isolation and Recovery (FDIR) systems are considered to ensure the safety and to increase the autonomy of spacecrafts. They shall be carefully designed and validated. Their implementation involves a relevant knowledge of items like functions and architectures of the system, and a fault model in relation with these items. Thus, the event-B method is well suited to correctly specify and validate on-board safety architectures.

This paper focuses on the FDIR concept presentation and the use of event-B for formalising and for refining the FDIR concept.

The paper is organised as follows: after a short presentation of on-board FDIR concept strongly bounded with autonomy architecture concept, we suggest activities enabling to implement FDIR concept. Then, we present the framework of formal modelling that we will use to describe our architecture and the properties related to this architecture. We illustrate our approach by modelling more specifically a safety architecture pattern that includes a primary functional component and a redundant one, under the hypothesis of no common fault. The safety property to be met is: "one single fault shall not lead to the total loss of the function". The last section of the paper deals with the objectives for the future work.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 358, 2008.

Object Modelling in the SystemB Industrial Project

Helen Treharne, Edward Turner, Steve Schneider [1], and Neil Evans[2]

[1]Department of Computing, University of Surrey
[2]AWE plc Aldermaston

Abstract. The SystemB project is a two year project at the University of Surrey, funded by AWE plc, and is concerned with bridging the areas of formal methods and object modelling. The project is focused on the CSP ‖ B integrated formal method and increasing its level of tool support so that CSP ‖ B models of Executable UML (xUML) systems can be constructed automatically. The CSP ‖ B models will subject the xUML model to formal analysis prior to generating executable code. We are currently developing a CSP ‖ B model generator within the xUML toolsuite provided by Kennedy Carter Ltd. xUML is used within AWE and we will initially focus on reasoning about xUML state machines. Actions within xUML state machines are defined using the Action Specification Language (ASL). ASL is more low level than the Object Constraint Language; they can execute concurrently, and can also be used in operation definitions. Hence it is a challenge to model formally. In the extended abstract we provide an overview of one ASL to AMN translation pattern being developed and highlight the role of B in the project.

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, p. 359, 2008.

Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification

Cliff B. Jones and Ken G. Pierce

School of Computing Science, Newcastle University, UK

Abstract. This paper presents a novel formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms (ACMs). Although the correctness of the "4-slot algorithm" has been shown elsewhere, earlier developments are by no means intuitive. The aims of this paper include both the presentation of an understandable (yet formal) design history and the establishment of another way of "splitting (software) atoms". Using the "fiction of atomicity" as an aid to understanding the initial steps of development, the top-level specification is developed to code. The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; the atomicity assumptions implied by rely/guarantee conditions are realised by clever choice of data representation. The development method herein is compared with other approaches –in a spirit of cooperation– as the authors believe that constructive comparison elucidates many of the finer points in the "4-slot" specification/development and of parallel programs in general.

1 Introduction

This paper is intended to contribute to methods of developing parallel programs; in particular it extends the repertoire of ways of "splitting (software) atoms safely". To do this, it addresses an intricate parallel program to illustrate the novel aspects of an approach to the development of parallel programs.

The general case for developing programs from abstractions is taken as read (cf. [Jon90, Abr96]). The VDM literature uses the terms "operation decomposition" and "data reification" for design steps of sequential programs and provides detailed proof obligations to justify such steps. Even if –as here– what is being created is a rational reconstruction of a design, the resulting documentation offers clarity and captures a design history to inform subsequent modification. Research on rely/guarantee conditions (see Section 2.2 below) extends the formal tools to cover classes of shared-variable concurrent programs. As has been repeatedly made clear in the literature, "compositionality" is essential to derive real pay off from a "posit and prove" approach.

More recently, research has looked at using a "fiction of atomicity" as an additional abstraction [Jon03] in the specification of parallel programs; the corresponding development notion is sometimes referred to as "splitting (software) atoms safely"; one example of this approach is Jones' transformation rules for "pobl" as in [Jon96].

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 360–377, 2008.

This paper uses rely and guarantee conditions in reasoning about "splitting atoms". In particular, the example illustrates the combination of rely/guarantee reasoning with data reification outlined in [Jon07].

Although this paper offers comparisons (see Section 6), it is quite specifically not competitive. In fact, the intention is to write a longer joint journal paper with Abrial and Cansell whose rather different approach [AC05] fits into the evolving research on "Event-B" which is being pursued in the EU "Deploy" project in which the first author is also a player. Moreover, the first author co-supervised Neil Henderson's PhD and encouraged the view that each of the approaches used in [Hen04] threw different light on the intricate algorithm that has also been chosen for the current paper.

The application chosen concerns "Asynchronous Communication Methods" — specifically, the four-slot implementation of ACMs devised by Hugo Simpson [Sim97] — see Section 3. The algorithm is ingenious and its correctness by no means obvious.

The real message of the current paper is however the (generic) approach outlined; a key test is whether the reader gains insight by reading the development below. In each major section, there is a sub-section that restates the methods used so that it is clear what the reader can take from the specific example to other specification and design challenges.

2 Background Material

This section *briefly* sets out state-of-the-art methods; any reader who is unfamiliar with these areas should consult the cited publications.

2.1 Data Reification

For many systems, data abstraction is key to achieving a concise and perspicuous specification. An algorithm might be easy to specify or describe in terms of tractable mathematical objects; its implementation might have to represent the abstraction in a complex way (possibly to achieve performance). Separation of these issues results in clearer documentation of design histories. The preferred development rule in VDM [Jon90] works where the chosen reification (representation of the abstraction) can be described using a "retrieve function" that is a many-to-one mapping from the representation back to the abstraction. This is possible where the abstraction is free from "implementation bias".

The simple VDM reification rule basically checks that (starting with a representation state) composing the retrieve function with the post condition of an abstract operation gives the same result as composing the post condition of the operation on the representation with the retrieve function. (There are restrictions to pre conditions –but here they are minimal– and an obligation to prove "adequacy" of a representation. All of this is explained in [Jon90, Chapter 8].)

There are however situations where the abstraction retains information to express potential non-determinacy and this information is superfluous in a step of

development where the non-determinacy is reduced. In a sense this is "intentional bias". In such situations it is necessary to use the development rule introduced by Tobias Nipkow in [Nip86, Nip87] that expresses a general relation between the abstraction and its reification.

For an exhaustive discussion of "data refinement" see [dRE99]; for a historical account of the development of the VDM rules see [Jon89].

2.2 Rely/Guarantee Thinking

Just as pre conditions simplify a designer's task by limiting the starting states in which the specified object is to be deployed, rely conditions indicate assumptions that the developer is allowed to make about the expected interference to a (shared-variable) concurrent program. Similarly, guarantee conditions can be compared to post conditions in that both are constraints on the behaviour of the created program.

VDM's operation decomposition rules for sequential programs have always used post conditions that relate the final state to the initial state (this is in contrast to many approaches that try to get by with predicates of the final state). Both rely and guarantee conditions are also relations between two states.

The general idea of documenting and reasoning about interference has many embodiments; some of the references include [Jon81, Jon83a, Jon83b, Jon96] but a number of other theses extend the basic idea.[1] A notable extension to cover progress arguments is [Stø90]. As the title of this section suggests, the approach is seen as a general way of thinking and reasoning about the design of concurrent systems rather than a specific set of rules. In fact, the general approach can also be applied to communication-based concurrency.

Once again, de Roever provides an encyclopaedic treatment in [dR01]; a particularly valuable contribution is the clear identification of the fact that rely/guarantee thinking achieves "compositionality".

A more recent development is the link made in [Jon07], between the achievement of a rely/guarantee specification and the designer's ability to find an appropriate data representation. This observation throws light on several older developments and is crucial to the design step in Section 5 below. Essentially, an abstraction is used that could be said to be using the "fiction of atomicity". The splitting of operations that have to be atomic on the abstraction is made possible by judicious choice of representation. So, for example, a variable whose monotonic reduction would imply locking can be represented by an expression involving the minimum of two values each of which can only be updated by one of two parallel processes.

2.3 Event Decomposition

Jean-Raymond Abrial's extension of his "B" approach [Abr96] to "event-B" is described in [AC05]. Guarded events are assumed to be executed atomically;

[1] See an on-line attempt to keep track of the literature at:
http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf

selection as to which event can be executed is non-deterministic if multiple guards evaluate to **true**. As such, this approach is completely different from that of rely/guarantee thinking (although Section 2.4 notes a common concern). The approach in [AC05] to increasing concurrency (or "splitting atoms") is to decompose events. When one "splits" an event into sub-events it has to be shown that all but one "refine skip".

There are a number of elegant examples of the use of this approach: Abrial and Cansell have also tackled the "4-slot" implementation of ACMs and have been kind enough to let us see their development as supported by the RODIN tools [Rod08]. Some further comments relating [AC08] to the material in this paper are made in Section 6.3.

2.4 Tracking Execution with Variables

There are two roles that variables can play that are close to "pseudo instruction counters". The shorthand term "phasing" is used in this paper to refer to either role.

In the (rely/guarantee) approach it is sometimes necessary to delineate different interference in different phases of a program. One way of handling this is by using pseudo-instruction counters and implications whose left-hand side makes appropriate case distinctions. One objective below is to show that using control constructs like "semicolon" provides another way of representing changing assumptions.

The other use of pseudo instruction counters is vividly illustrated in Abrial's event refinement approach. The order of execution of the events with true guards in a given set is non-deterministic. In situations where the correctness depends on a constrained order, pseudo instruction counters are tested in guards and set in the corresponding events.[2] The Abrial/Cansell approach is discussed in Section 6.3.

2.5 Status of the Proofs

The authors have checked all of the proof obligations required in the development below. A technical report version of this paper will add appendices that make outline proofs available for scrutiny. The second author's thesis will present proofs at the level of formality used in [Jon90]. Plans to attempt machine checked proofs are currently being considered.

3 ACMs and Their Specification

"Asynchronous Communication Methods" (ACMs) address an extremely interesting application scenario. First, imagine two process that are independently

[2] This is reminiscent of the proof of the Boehm/Jacopini theorem that "goto" statements can be avoided.

timed in the sense that they are not synchronised in any way (thus "asynchronous"); furthermore, suppose that one process produces values that are to be "communicated" to the other (one writes and the other reads); the key requirement is that communication must be achieved with *no delay* to either process. So it is *not*, for example, possible to use a conventional shared variable –access to which is controlled by some device such as semaphores– since one process could be delayed waiting for a lock to be released. To sharpen the issues, it might be useful to think of *Value* below as being large — something that certainly can't be changed in one machine cycle ("atomically"). ACMs are used in important high speed communication situations such as passing values from sensors to flight control software.

A number of non-obvious consequences follow from the asynchronous essence of ACMs. The simplest is that it is certainly valid for the reader to see the same value multiple times if it cycles faster than the writer. The more complicated consequences are shown once a formal specification has been given.

3.1 A Specification

The issue of a formal statement of what behaviour a valid ACM is allowed to exhibit is itself non-trivial and different approaches are already distinguished at this starting point. (Alternatives are discussed in Section 3.3.) Sections 4 and 5 present a formal development of a well-known –and extremely ingenious– implementation of ACMs but it is clearly necessary to offer a formal starting point for such a development. The aim here is to provide a way of specifying ACM behaviour with which a user can feel comfortable.

It would fit the "splitting atoms" programme nicely if it were possible to present a specification using of a simple (atomic) variable. Unfortunately, this is not an appropriate abstraction because it does not show the full potential behaviour of an ACM. In particular, a read operation could start and –before it delivers a value– several write operations could start and complete. Alternatively, a write operation could start and –before it completes– several different read operations could start and complete. It is necessary to show which values can be delivered to the receiving process. A straightforward way to do this is to distinguish between *start-Read*/*end-Read* and *start-Write*/*end-Write*.[3] An underlying state to characterise these operations (in VDM notation[4]) could be:

$$\begin{array}{lll} \Sigma^a :: & \mathit{data\text{-}w} & : \mathit{Value}^* \\ & \mathit{fresh\text{-}w} & : \mathbb{N} \\ & \mathit{hold\text{-}r} & : \mathbb{N} \end{array}$$

$$\textbf{inv}\ (\mathit{mk\text{-}}\Sigma^a(\mathit{data\text{-}w}, \mathit{fresh\text{-}w}, \mathit{hold\text{-}r})) \triangleq \mathit{hold\text{-}r} \leq \mathit{fresh\text{-}w}$$

[3] For those who feel queasy about this in a specification, Section 3.3 discusses alternatives. Furthermore, the approach of the current section can be proved to fit with such specifications.

[4] Remember that types in VDM are restricted by invariants; so, for example, quantifying over Σ^a only includes records that satisfy its invariant.

The idea here is that *data-w* retains all values written; *start-Write* first stores a new value but only *end-Write* releases it for access by updating *fresh-w*. Conversely, *start-Read* notes the index of values that must be regarded as "fresh" and *end-Read* makes a non-deterministic choice of an index between the *hold-r* and the value of *fresh-w* at the time of completion of the read. (The suffixes of the variable names indicate whether the reader or writer can change their values; this shows straight away that there are no variables written to by both "sides".)

It is obviously necessary to initialise the state. Most authors who give formal presentations do this by assuming that a value x has been written *then* read once. This can be shown as:

$\sigma_0^a = mk\text{-}\Sigma^a([\mathtt{x}], 1, 1)$

with the following pseudo-code:

while true do
  start-Write(*v*: *Value*): *data-w* ← *data-w* $\curvearrowright$ [*v*];
  end-Write(): *fresh-w* ← **len** *data-w*
od
while true do
  start-Read(): *hold-r* ← *fresh-w*;
  end-Read()*r*: *Value*: *r* ← *data-w*(*i*) **for some** $i \in \{hold\text{-}r..fresh\text{-}w\}$
od

The code ensures that old values cannot be read. Although *end-Read* might not select the newest item in the sequence, a value only becomes old when a newer item is returned. Since *start-Read* sets *hold-r* to the value of *fresh-r* before the choice is made and *hold-r* is never greater than *fresh-r*, the read process cannot return an old value (though the same value may be returned more than once).

Figures 1, 2 and 3 give possible executions of the code (giving the operation name and corresponding final sate). Figure 1 is a simple sequential write and read: y is added to *data-w*, marked as fresh and subsequently read. In Figure 2, the read begins before the write ends and the read yields x.

start-Write(y)	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 1, 1)$
end-Write()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 2, 1)$
start-Read()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 2, 2)$
end-Read()	.. $r = \mathtt{y}$

Fig. 1. Sequential case

start-Write(y)	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 1, 1)$
start-Read()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 1, 1)$
end-Read()	.. $r = \mathtt{x}$
end-Write()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 2, 1)$

Fig. 2. Interleaved case

start-Read()	.. $mk\text{-}\Sigma^a([\mathtt{x}], 1, 1)$
start-Write(y)	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 1, 1)$
end-Write()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}], 2, 1)$
start-Write(z)	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}], 2, 1)$
end-Write()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}], 3, 1)$
end-Read()	.. $r \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}$
start-Read()	.. $mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}], 3, 3)$
end-Read()	.. $r = \mathtt{z}$

Fig. 3. Non-deterministic case

The more complex case in Figure 3 shows the non-determinism of the read operation. By the time *end-Read* is ready to return a result, three possible values are available and one will be selected non-deterministically. Note however that a subsequent read can return neither `x` nor `y` because *hold-r* is updated to the value of *fresh-w* at the start of the read.

The pseudo-code above is brief and offers the intuition of what can happen but for the development that follows, this needs to be presented as formal (VDM) specifications of the four operations. These are straightforward (see Figure 4)[5].

Write(*v*: *Value*)
  start-Write(*v*: *Value*)
    wr *data-w*
    post $data\text{-}w = \overleftarrow{data\text{-}w} \frown [v]$
  end-Write(*v*: *Value*)
    rd *data-w*
    wr *fresh-w*
    pre $data\text{-}w(\mathbf{len}\ data\text{-}w) = v$
    post $fresh\text{-}w = \mathbf{len}\ data\text{-}w$

Read()*r*: *Value*
  local $hold\text{-}r\text{:}\,\mathbb{N}$
  start-Read()
    wr *hold-r*
    rd *fresh-w*
    post $hold\text{-}r = fresh\text{-}w$
  end-Read()*r*: *Value*
    rd $data\text{-}w, fresh\text{-}w$
    post $\exists i \in \{hold\text{-}r..fresh\text{-}w\} \cdot r = data\text{-}w(i)$

Fig. 4. Specification in terms of four sub-operations

In these operation specifications, the standard VDM style of marking the read/write access is used. This proves particularly valuable below when interference is considered. One non-standard extension is also used and that is the declaring *hold-r* as local to the two *Read* operations. This is essentially marking it as invisible to the two *Write* operations. (The efficacy of these markings is addressed in the (draft) thesis of the second author.)

Notice that rely/guarantee conditions are not *yet* necessary because the four operations are assumed to be atomic. The fiction of atomicity is used to achieve a simple specification. At this point, the specifications imply big assumptions about atomic update of *data-w*; this is addressed in the following sections. Perhaps of more interest is the decision to use semicolon as a tool in specifications — again, this is addressed below.

[5] As an aside: It would be reasonable to assume that a *Read* operation will run in less time than a *Write* — in this case it would be impossible for multiple *Write*s to complete within the time of a *Read* — such an assumption can slightly simplify solutions. This assumption is not made here (nor in most other papers).

Note that *pre-end-Write* is required to pass information between the two write processes. The proof showing that this is implied by *post-start-Write* is immediate.

3.2 Splitting Atoms in Σ^a

As observed, the operations in the preceding section are assumed to execute atomically. The process of "splitting atoms" can begin by considering the overlap of *Read* and *Write* sub-operations. This could be very difficult to describe. Indeed, the cleverness of the final code is all about finding a way to do this safely whilst achieving "asynchroneity". Here, we postpone the key property that there is no delay to either process since it can be achieved by further splitting of atoms.

For now, Figure 5 contains exactly the post conditions of the preceding section and adds rely and guarantee conditions that represent the possible interference. Notice first that the state Σ^a is unchanged. Rely/guarantee assertions are easy to add because all that is necessary is to make sure that results required in the post conditions cannot be subverted by interference.

It is a simple task to check that the rely and guarantee conditions in the two threads are consistent. The work involved is almost syntactic because of the limitations on read and write access marked in the VDM operation specifications. For example, both *rely-start-Write* and *rely-end-Write* follow immediately from the fact that neither *Read* component has write access to the relevant variables.

An astute reader might be very worried that massive assumptions are being made here about what can be changed atomically. Such assumptions have to be eliminated in subsequent development. What is achieved here is to show that the splitting atoms development idea can provide an intuitive understanding of extremely delicate code.

The details of the rely and guarantee operations are, here, made much simpler to write because of the way that the sub operations are ordered (by semicolon). Were one to try to record a specification of an entire *Read* and *Write* operations, they would be festooned with implications. The structure of the program (e.g. that *Write* cannot interfere with *Write*) simplifies the specifications of the sub-operations.

3.3 Alternative Specifications

The most surprising decision in the specification used here is the retention of values (in *data-w* of σ^a) that can no longer be accessed. Henderson goes to pains to delete "old" values after they have been overtaken by subsequent reads. The cost in [Hen04] is that both *Read* and *Write* need to have a record of where the other process is in its execution. True, this record keeping is eliminated in the subsequent development; but so are our superfluous values. In both cases, the same technical rule comes to the rescue. Our current view is that leaving the extra values results in a clearer specification.

Abrial and Cansell [AC08] start from a specification in terms of the traces of reading and writing. It is inherent in the ACM problem –rather than a criticism of

$Write(v\colon Value)$
  $start\text{-}Write(v\colon Value)$
    rd $fresh\text{-}w$
    wr $data\text{-}w$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    guar $\{1..fresh\text{-}w\} \lhd data\text{-}w = \{1..fresh\text{-}w\} \lhd \overleftarrow{data\text{-}w}$
    post $data\text{-}w = \overleftarrow{data\text{-}w} \frown [v]$
  $end\text{-}Write(v\colon Value)$
    rd $data\text{-}w$
    wr $fresh\text{-}w$
    pre $data\text{-}w(\textbf{len}\ data\text{-}w) = v$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    post $fresh\text{-}w = \textbf{len}\ data\text{-}w$

$Read()r\colon Value$
  $start\text{-}Read()$
    rd $fresh\text{-}w$
    wr $hold\text{-}r$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r}$
    post $hold\text{-}r \in \{\overleftarrow{fresh\text{-}w}, fresh\text{-}w\}$
  $end\text{-}Read()r\colon Value$
    rd $data\text{-}w, fresh\text{-}w, hold\text{-}r$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r} \wedge \forall i \in \{hold\text{-}r..\overleftarrow{fresh\text{-}w}\} \cdot data\text{-}w(i) = \overleftarrow{data\text{-}w}(i)$
    post $\exists i \in \{hold\text{-}r..\overleftarrow{fresh\text{-}w}\} \cdot r = \overleftarrow{data\text{-}w}(i)$

Fig. 5. Specification of sub-operations on Σ^a with rely/guarantee

their specification– that pinning down the exact behaviour is somewhat messy: in essence, they have to reflect the points at which operations start and end. In the journal version of this paper a proof will be added that the initial "specification" in Section 3.1 satisfies their specification. This then leaves the user to decide which is the most intuitive way of understanding ACM behaviour.

3.4 Summary of Specification Methods Used

The ideal of the "fiction of atomicity" would be to abstract from all of the details of ACMs by using a single atomically accessed variable as an abstraction. Since this does not describe all of the possible behaviours, one has to think harder to obtain a starting specification. The choice here is to make a minimal split of the two parallel processes each into two sub-operations whose behaviour is composed sequentially ("by semicolon"). This "phasing" is of course algorithmic detail in a specification but is claimed to offer a reasonably intuitive description of the permissable behaviours of an ACM. The same phasing idea pays off handsomely when the move is made to specifications with rely and guarantee conditions: if the same essential properties were to be presented for the whole of say *Write*,

there would have to be ghost variables to track the phase and implications to present the information about the separate phases as a single predicate. The current authors recognise the arguments for a specification in terms of traces but believe phasing is sometimes easier to understand.

The rely and guarantee conditions themselves are fairly standard. Checking that they are consistent between the two parallel threads is made almost trivial by judicious choice of frame markings.

4 Retaining Less History

The first real reification is to an intermediate representation in which it is possible to retain fewer *Value*s than in Σ^a. Not only can Σ^i get away with fewer values, it is also clear that it might be possible to lock only parts of its *data-w* component and thus increase concurrency. Thus this step moves towards the idea of multiple slots without being specific as to how many there must be to make the algorithm work. Essentially, a careful data reification step is bringing in some of the design decisions without going all the way to Simpson's code. Rely/guarantee conditions are again used to investigate the requirements.

4.1 The Data Representation

The state representation for this reification is:

$$\begin{array}{lll}\Sigma^i :: & \mathit{data\text{-}w} & : X \xrightarrow{m} \mathit{Value} \\ & \mathit{fresh\text{-}w} & : X \\ & \mathit{hold\text{-}r} & : X \\ & \mathit{hold\text{-}w} & : X\end{array}$$

$$\textbf{inv}\ (\mathit{mk}\text{-}\Sigma^i(\mathit{data}, \mathit{fresh}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w})) \triangleq \{\mathit{fresh}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w}\} \subseteq \textbf{dom}\ \mathit{data}$$

At this step of development, the (indexing) set X is arbitrary. In order to show the initial state assume that $X \in \{\alpha, \beta, \ldots\}$. Then:

$$\sigma_0^i = \mathit{mk}\text{-}\Sigma^i(\{\alpha \mapsto \mathtt{x}\}, \alpha, \alpha, \alpha)$$

4.2 Relating Σ^i to Σ^a

The fact that the chosen state in the specification (Σ^a) cannot be "retrieved" from the representation as in the simple VDM reification rule means that the connection between elements of Σ^a/Σ^i has to be given by a relation:

$$r : \Sigma^a \times \Sigma^i \to \mathbb{B}$$

$$\begin{array}{l} r(\mathit{mk}\text{-}\Sigma^a(\mathit{data\text{-}w}^a, \mathit{fresh\text{-}w}^a, \mathit{hold\text{-}r}^a), \\ \qquad\qquad \mathit{mk}\text{-}\Sigma^i(\mathit{data\text{-}w}^i, \mathit{fresh\text{-}w}^i, \mathit{hold\text{-}r}^i, \mathit{hold\text{-}w}^i)) \triangleq \\ \quad \textbf{rng}\ \mathit{data\text{-}w}^i \subseteq \textbf{elems}\ \mathit{data\text{-}w}^a \wedge \\ \quad \mathit{data\text{-}w}^a(\mathit{fresh\text{-}w}^a) = \mathit{data\text{-}w}^i(\mathit{fresh\text{-}w}^i) \wedge \\ \quad \mathit{data\text{-}w}^a(\mathit{hold\text{-}r}^a) = \mathit{data\text{-}w}^i(\mathit{hold\text{-}r}^i) \end{array}$$

Because the representation here has (potentially) less information than the abstraction, it is necessary to use the refinement rule given in [Nip86, Nip87]. For the current case it is necessary to show for each operation that:

$$r(\sigma_1^a, \sigma_1^i) \wedge post^i(\sigma_1^i, \sigma_2^i) \;\Rightarrow\; \exists \sigma_2^a \in \Sigma^a \cdot post^a(\sigma_1^a, \sigma_2^a) \wedge r(\sigma_2^a, \sigma_2^i)$$

Generalisations of this rule to add inputs or outputs are obvious.

4.3 Specifications of the Sub-operations

The specifications of the four sub-operations over the Σ^i states are shown in Figure 6. There is masses of non-determinism here — in fact, one valid implementation is to have $X = \mathbb{N}$ and retain the whole sequence as in Section 3.

The post condition of *start-Write* clearly shows that we need at least three slots in order to avoid "race conditions" on individual *Values*.[6]

Write(*v*: *Value*)
  local *hold-w*: X
  start-Write(*v*: *Value*)
    rd *hold-r*, *fresh-w*
    wr *data-w*
    rely $\mathit{fresh\text{-}w} = \overleftarrow{\mathit{fresh\text{-}w}} \wedge \mathit{data\text{-}w} = \overleftarrow{\mathit{data\text{-}w}}$
    guar $\{\overleftarrow{\mathit{hold\text{-}r}}, \mathit{hold\text{-}r}\} \lhd \mathit{data\text{-}w} = \{\overleftarrow{\mathit{hold\text{-}r}}, \mathit{hold\text{-}r}\} \lhd \overleftarrow{\mathit{data\text{-}w}}$
    post $\mathit{hold\text{-}w} \in (X - \{\mathit{fresh\text{-}w}, \overleftarrow{\mathit{hold\text{-}r}}, \mathit{hold\text{-}r}\}) \wedge$
      $\mathit{data\text{-}w} = \overleftarrow{\mathit{data\text{-}w}} \dagger \{\mathit{hold\text{-}w} \mapsto v\}$
  end-Write(*v*: *Value*)
    rd *data-w*
    wr *fresh-w*
    pre $\mathit{data\text{-}w}(\mathit{hold\text{-}w}) = v$
    rely $\mathit{fresh\text{-}w} = \overleftarrow{\mathit{fresh\text{-}w}} \wedge \mathit{data\text{-}w} = \overleftarrow{\mathit{data\text{-}w}}$
    post $\mathit{fresh\text{-}w} = \mathit{hold\text{-}w}$

Read()*r*: *Value*
  start-Read()
    rd *fresh-w*
    wr *hold-r*
    rely $\mathit{hold\text{-}r} = \overleftarrow{\mathit{hold\text{-}r}}$
    post $\mathit{hold\text{-}r} \in \{\overleftarrow{\mathit{fresh\text{-}w}}, \mathit{fresh\text{-}w}\}$
  end-Read()*r*: *Value*
    rd *hold-r*, *data-w*
    rely $\mathit{hold\text{-}r} = \overleftarrow{\mathit{hold\text{-}r}} \wedge \mathit{data\text{-}w}(\mathit{hold\text{-}r}) = \overleftarrow{\mathit{data\text{-}w}}(\mathit{hold\text{-}r})$
    post $r = \mathit{data\text{-}w}(\mathit{hold\text{-}r})$

Fig. 6. Rely/guarantee specifications on Σ^i

[6] The argument why that is not enough is set out in [Hen04] and is not repeated here.

4.4 Justifying This Step

The technical report version of this paper contains proofs of the (initialisation, and) four sub-operations in an appendix. These proofs will be presented formally in the second author's forthcoming PhD thesis.

Checking the coherence of the rely and guarantee conditions between the two sub-operations of *Read* and of *Write* is somewhat more work than in Section 3 but the effort required is still drastically reduced by the read and write frames (coupled with the **local** in *Write*).

4.5 Summary of Development Methods Used in This Stage

The justification of the data reification from Σ^a to Σ^i cannot be done using the simpler of the two rules in the VDM literature but the rule from Nipkow's thesis covers the (possible) reduction in the size of the state space and this rule has been included in VDM since [Jon90, §9.3]. The use here is technically interesting; in fact, its availability makes possible the choice of development from Σ^a to Σ^r via Σ^i. Such careful choice of design strategy is essential but is perhaps the hardest part of the method to reduce to general rules.

Another key point only sees its completion in Section 5 and that is the use even at this step of rather bold atomicity assumptions. Without Simpson's clever data representation it might be impossible to achieve atomic update (on a reasonable machine architecture) without locking and it is made clear in Section 3 that this is not allowed in ACMs. Such roadblocks (leading to backtracking) cannot be ruled out by any method whether formal or informal.

There are key links from this section to the second author's upcoming PhD thesis. In particular, one sees even more clearly in this section than the last how rely and guarantee conditions are simpler to express because of the read and write frames. Furthermore, without "phasing", there would be much more to write with implications all over the place.

5 The Four-Slot Representation

In purely formal terms, the task remaining after Figure 6 (which uses Σ^i) is to find a representation that admits atomic changes in a sensible machine. The crucial contribution of Simpson's "4-slot" algorithm is to achieve control over where the reader and writer find or change values with only two single bit control variables. This is where the link between "splitting atoms" and reification (cf. [Jon07, §3]) comes into play. These control variables keep the reader and writer from "colliding" while never delaying each other. This is the ingenuity in Hugo Simpson's contribution and there is absolutely no claim here that the formalism is a substitute for such design inspiration. Used by a designer (which it wasn't), formalism can establish that proceeding to the next design step leaves no hostage to fortune on correctness; used as here, the formalism can provide a clear understanding, documentation (and appreciation) of an intricate piece of code.

5.1 The Data Representation

The size of the domain of the *data-w* field of Σ^i is not constrained. Simpson's "4-slot" approach shows that the domain need only have cardinality four. Furthermore, he shows that treating the *data* map as two pairs (P below) of two slots (indexed by S below) makes their bookkeeping atomic.

So the essential difference between Σ^i of Section 4 and Σ^r here is that the general index set X of the former is represented here as a pair (P, S). The other changes are to control variables whose role becomes clear in Section 5.2.

$$\begin{array}{lll} \Sigma^r :: & \mathit{data\text{-}w} & : P \times S \xrightarrow{m} \mathit{Value} \\ & \mathit{pair\text{-}w} & : P \\ & \mathit{pair\text{-}r} & : P \\ & \mathit{slot\text{-}w} & : P \xrightarrow{m} S \\ & \mathit{wp\text{-}w} & : P \\ & \mathit{ws\text{-}w} & : S \\ & \mathit{rs\text{-}r} & : S \end{array}$$

where:

$$P, S = \mathit{token}\text{-}\mathbf{set}$$

These two sets can be identical and each has two elements: $P = S, \mathbf{card}\, P = 2$, with an inverter function, ρ (for "reverse")[7], such that $\rho(i) \neq i$.

5.2 Justifying the Step from Σ^r to Σ^i

Figure 7 reflects the differences between the two state spaces. The justification of this step of development requires showing that the combination of index values in Σ^r can be used to justify the properties of X etc. in Σ^i. (This is the process described in [Jon07, §3].) Thus one shows that each of the conditions of Figure 7 corresponds to those of Figure 6. This follows from:

Σ^i	represented in Σ^r by
$\mathit{data\text{-}w}^i$	$\mathit{data\text{-}w}^r$
fresh^i	$(\mathit{pair\text{-}w}^r, \mathit{slot\text{-}w}^r(\mathit{pair}^r(\mathit{pair\text{-}w}^r)))$
$\mathit{hold\text{-}r}^i$	$(\mathit{pair\text{-}r}^r, \mathit{slot\text{-}w}^r(\mathit{pair}^r(\mathit{pair\text{-}r}^r)))$
$\mathit{hold\text{-}w}^i$	$(\mathit{wp\text{-}w}^r, \mathit{wp\text{-}s}^r)$

The proofs will be given in an appendix to the technical report version of this paper.

5.3 The Code

It is straightforward to show that the code in Figure 8 satisfies the specifications of the sub-operations in Section 5.2.

[7] Many authors use $\mathbb{B}$ for P and then employ negation — to us, this is a coding trick!

$Write(v\colon Value)$
  local $wp\text{-}w\colon P$
  local $ws\text{-}w\colon S$
  $start\text{-}Write(v\colon Value)$
    rd $pair\text{-}r, slot\text{-}w$
    wr $data\text{-}w$
    rely $slot\text{-}w = \overleftarrow{slot\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    guar $\{(\overleftarrow{pair\text{-}r}, slot\text{-}w(\overleftarrow{pair\text{-}r}), (pair\text{-}r, slot\text{-}w(pair\text{-}r)\} \lhd data\text{-}w =$
      $\{(\overleftarrow{pair\text{-}r}, slot\text{-}w(\overleftarrow{pair\text{-}r}), (pair\text{-}r, slot\text{-}w(pair\text{-}r)\} \lhd \overleftarrow{data\text{-}w}$
    post $wp\text{-}w = \rho(\overleftarrow{pair\text{-}r}) \wedge ws\text{-}w = \rho(slot\text{-}w(wp\text{-}w)) \wedge$
      $data\text{-}w(wp\text{-}w, ws\text{-}w) = v$
  $end\text{-}Write()$
    wr $pair\text{-}w, slot\text{-}w$
    rely $pair\text{-}w = \overleftarrow{pair\text{-}w} \wedge slot\text{-}w = \overleftarrow{slot\text{-}w}$
    guar $slot\text{-}w(pair\text{-}r) = \overleftarrow{slot\text{-}w}(pair\text{-}r)$
    post $slot\text{-}w(wp\text{-}w) = ws\text{-}w \wedge pair\text{-}w = wp\text{-}w$

$Read()r\colon Value$
  local $rs\text{-}r\colon S$
  $start\text{-}Read()$
    rd $pair\text{-}w, slot\text{-}w$
    wr $pair\text{-}r$
    rely $slot\text{-}w(pair\text{-}r) = \overleftarrow{slot\text{-}w}(pair\text{-}r) \wedge pair\text{-}r = \overleftarrow{pair\text{-}r}$
    post $pair\text{-}r = \overleftarrow{pair\text{-}w} \wedge rs\text{-}r = \overleftarrow{slot\text{-}w}(pair\text{-}r)$
  $end\text{-}Read()r\colon Value$
    rd $pair\text{-}r, data\text{-}w$
    rely $pair\text{-}r = \overleftarrow{pair\text{-}r} \wedge data\text{-}w(pair\text{-}r, rs\text{-}r) = \overleftarrow{data\text{-}w}(pair\text{-}r, rs\text{-}r)$
    post $r = data\text{-}w(pair\text{-}r, rs\text{-}r)$

Fig. 7. Final (Σ^r) rely/guarantee specification of code

$Write(v\colon Value)$
  local $wp\text{-}w\colon P$
  local $ws\text{-}w\colon S$
    $wp\text{-}w \leftarrow \rho(pair\text{-}r)$;
    $ws\text{-}w \leftarrow \rho(slot\text{-}w(wp\text{-}w))$;
    $data\text{-}w(wp\text{-}w, ws\text{-}w) \leftarrow v$;
    $slot\text{-}w(wp\text{-}w) \leftarrow ws\text{-}w$;
    $pair\text{-}w \leftarrow wp\text{-}w$

$Read()r\colon Value$
  local $rs\text{-}r\colon S$
    $pair\text{-}r \leftarrow pair\text{-}w$;
    $rs\text{-}r \leftarrow slot\text{-}w(pair\text{-}r)$;
    $r \leftarrow data\text{-}w(pair\text{-}r, rs\text{-}r)$

Fig. 8. Code for Simpson's algorithm

5.4 Summary of Development Methods Used in This Stage

Finally, the usefulness of the intermediate data abstraction becomes clear in this step: it is relatively easy to see the pair/slot mapping as a way of simplifying a mapping from the arbitrary set X. Moreover, the whole thrust of "splitting atoms safely" is clear in this step.

6 Conclusions

This section both summarises the general methodological messages of the paper and offers brief descriptions of some other recent justifications of Simpson's algorithm. In making such comparisons, the authors are not trying to be competitive but to use this intricate algorithm to indicate what insight can be given by various approaches.

6.1 Summary

As made clear at the outset, ACMs are complex; Simpson's algorithm is ingenious; and its correctness requires delicate reasoning. The material in Figures 5–7 is key to providing an intuitive grasp of the correctness. The authors hope that the reader finds this a clear design rationale. (The material pre Figure 5 is really there to provide an intuition of the behaviour.)

However, the intention was not to add yet another correctness argument of one specific algorithm but instead to use this development to illustrate how a number of ideas can be used in concert to move from a "fiction of atomicity" using a development approach that can be called "splitting (software) atoms safely". The notes in Sections 3.4, 4.5 and 5.4 can be summarised as:

- The authors present an understandable and tractable reworking of the "4-slot" algorithm, with a clear design history.
- The "fiction of atomicity" is a good place to begin.
- While rely/guarantee conditions allow us to reason about the interference, a clever data reification is required (which Simpson gives us).
- Rely/guarantee reasoning is greatly simplified by the use of frames and phasing arguments.

6.2 Brief Comments on Henderson's Development

Henderson's research (in particular, his thesis [Hen04]) has been a key information source. Interestingly, he uses broadly the same set of technical tools as in the current paper. In spite of this, the presentation here looks very different.

First, Henderson's specification attempts to retain a minimal list of *Value*s that could potentially be returned by a *Read*. As mentioned in Section 3.3, a cost for this is a pair of "ghost variables" that inform the *Read* operation in which phase the *Write* operation is executing (and *vice versa*). These variables can be eliminated in reification because Henderson also uses "Nipkow's rule".

The current authors hold the (biased) view that the specification here is clearer but there would be little difficulty in proving they describe the same behaviour and the choice can be left to the "customer".

A more pervasive difference results in part from the recent development (cf. [Jon07]) of the link between atomicity refinement and data reification. In Section 5 of the current paper, the preceding interference specifications are achieved by capitalising on Simpson's four-slot representation.

The reader is also referred to [HP02] and [PHA04]; the second of these addresses the delicate issue of "meta-stability" of the control bits.

6.3 Comparison with Event Decomposition

The "event decomposition" method described in, for example, [AC05] is extremely interesting because it is general. Attention has already been drawn above to its use of a "pseudo instruction counter" which is related to the "phasing" idea used here. They avoid any need for rely and guarantee conditions by preserving the atomicity of events at any level of development. This achieves a considerable economy of rules.

The current authors do wonder whether the interesting development of Simpson's algorithm in [AC08] indicates that the atomicity constraint might require a series of difficult-to-invent steps. But their forthcoming publication will admit wider comparison (and by people unbiased by being authors of either approach). As indicated, it is the hope of the current authors that a comparison paper might be written together with Jean-Raymond Abrial and Dominique Cansell.

6.4 Comparison with "Separation Logic"

Another exciting development in research on concurrent code has been the recent developments around "concurrent separation logic". At this time, researchers in Newcastle, London and Cambridge are discussing ways of combining the best features of both separation logic and rely/guarantee reasoning. For example, the second author's thesis builds the bridge with the read/write frames here. There is not space here to do this research full justice; but an excellent recent reference (from which other citations can be found) is [Vaf07].

During the writing of this paper, Richard Bornat sent us current work on Simpson's algorithm. The title of [BA08] alone should indicate why this is exciting. Again, the availability of this in published form will admit proper unbiased comparison.

Acknowledgments

The authors are grateful to Jean-Raymond Abrial and Dominique Cansell for sharing ongoing work in this area. Similarly, the preview of the paper by Richard Bornat and Hasan Amjad is gratefully acknowledged even though time has not yet permitted a full comparison. Thanks also go to Peter O'Hearn, Hongseok

Yang, Viktor Vafeiadis and Matt Parkinson for general and ongoing discussions on development methods for concurrency.

Of course, the original inspiration of the specific algorithm comes from Hugo Simpson's contribution. Neil Henderson and Steve Paynter made us aware of the challenge of this tiny but intriguing problem.

Our research is supported by the EPSRC Platform Grant on "Trustworthy Ambient Systems" and EU FP7 "DEPLOY project".

References

[Abr96] Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)

[AC05] Abrial, J.-R., Cansell, D.: Formal construction of a non-blocking concurrent queue algorithm. Journal of Universal Computer Science 11(5), 744–770 (2005)

[AC08] Abrial, J.-R., Cansell, D.: Development of a comcurrent program (2008) (private communication)

[BA08] Bornat, R., Amjad, H.: Inter-process buffers in separation logic with rely-guarantee. Formal Aspects of Computing (private communication) (submitted, 2008)

[dR01] de Roever, W.P.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press, Cambridge (2001)

[dRE99] de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and Their Comparison. Cambridge University Press, Cambridge (1999)

[Hen04] Henderson, N.: Formal Modelling and Analysis of an Asynchronous Communication Mechanism. PhD thesis, University of Newcastle upon Tyne (2004)

[HP02] Henderson, N., Paynter, S.E.: The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 350–369. Springer, Heidelberg (2002)

[Jon81] Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (June 1981); Printed as: Programming Research Group, Technical Monograph 25

[Jon83a] Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland, Amsterdam (1983)

[Jon83b] Jones, C.B.: Tentative steps toward a development method for interfering programs. Transactions on Programming Languages and System 5(4), 596–619 (1983)

[Jon89] Jones, C.B.: Data reification. In: McDermid, J.A. (ed.) The Theory and Practice of Refinement, pp. 79–89. Butterworths (1989)

[Jon90] Jones, C.B.: Systematic Software Development using VDM, 2nd edn. Prentice Hall International, Englewood Cliffs (1990)

[Jon96] Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods in System Design 8(2), 105–122 (1996)

[Jon03] Jones, C.B.: Wanted: a compositional approach to concurrency. In: McIver, A., Morgan, C. (eds.) Programming Methodology, pp. 1–15. Springer, Heidelberg (2003)

[Jon07] Jones, C.B.: Splitting atoms safely. Theoretical Computer Science 357, 109–119 (2007)

[Nip86] Nipkow, T.: Non-deterministic data types: Models and implementations. Acta Informatica 22, 629–661 (1986)

[Nip87] Nipkow, T.: Behavioural Implementation Concepts for Nondeterministic Data Types. PhD thesis, University of Manchester (May 1987)

[PHA04] Paynter, S.E., Henderson, N., Armstrong, J.M.: Ramifications of metastability in bit variables explored via Simpson's 4-slot mechanism. Formal Aspects of Computing 16(4), 332–351 (2004)

[Rod08] Rodin.: Rodin tools can be downloaded from SourceForge (2008), `http://sourceforge.net/projects/rodin-b-sharp/`

[Sim97] Simpson, H.R.: New algorithms for asynchronous communication. IEE, Proceedings of Computer Digital Technology 144(4), 227–231 (1997)

[Stø90] Stølen, K.: Development of Parallel Programs on Shared Data-Structures. PhD thesis, Manchester University (1990), Available as UMCS-91-1-1

[Vaf07] Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)

ABZ2008 VSR-Net Workshop

Jim Woodcock[1] and Paul Boca[2]

[1] University of York
[2] London Southbank University

In 2004, the UK Computing Research Committee initiated a number of *Grand Challenges* aimed at stimulating long term research in key areas of computing science. One of the challenges (GC6) focuses on Dependable Systems Evolution. GC6 has two central principles: theory should be embodied in tools, and tools should be tested against real systems. The goal is to produce a Verifying Compiler (a suite of integrated tools) and a repository of verified software.

The results of the first VSR pilot project, conducted in 2006, on verifying the Mondex system, using different formalisms and tools, are the first artifacts to be deposited in the repository. A number of other pilot studies are underway in various parts of Europe, USA, Brazil, Canada, and China. This is a truly international initiative, and the UK is a major contributor.

In 2005, Joshi and Holzman from the Jet Propulsion Laboratory at Caltech suggested the specification and verification of a POSIX-compliant filestore interface to flash memory—see `http://www.cs.york.ac.uk/circus/mc/abz` for a good overview of the challenge problem. The filestore has strict fault-tolerance requirements that make it suitable for use by forthcoming NASA missions. This was suggested to the ASM, B and Z communities as an interesting problem to work on, and some rose to this challenge by submitting papers to the ABZ 2008 conference (see the paper by Eunsuk Kang and Daniel Jackson, entitled *Formal Modeling and Analysis of a Flash Filesystem in Alloy*, in these proceedings).

VSR-net is approaching the end of its three years of ESPRC funding, and this workshop provides a summary of the various achievements to date and a forum to discuss how to move forward with the Challenge. There is an opportunity for reflection too: Tony Hoare gives his perspective on how the Challenge has evolved and how it differs from what he originally envisaged.

The workshop features a number of technical talks, and we give a few highlights here. Cliff Jones, University of Newcastle, a leading figure in the verification area, will give an invited talk entitled *Splitting atoms with rely/guarantee-conditions coupled with data reification*. In this talk Jones describes a novel, and intuitive formal development of an implementation of Simpsons *4-slot algorithm*. The paper accompanying this talk appears elsewhere in these proceedings).

Jim Woodcock, University of York, in his talk *Progress on the Verified Software Repository* discusses past, present and future challenge problems. Three of the areas Woodcock covers (Mondex, Cardiac Pacemaker, and the Verified Filestore) are represented at the workshop:

– Richard Banach (University of Manchester). *The Mondex Purse Requirements and Retrenchments.*

E. Börger et al. (Eds.): ABZ 2008, LNCS 5238, pp. 378–379, 2008.

- John Fitzgerald (University of Newcastle). *Approaches to the Pacemaker Challenge Problem.*
- Michael Butler (University of Southampton). *Applying Event-B and Rodin to the filestore.*
- Eunsuk Kang (MIT). *Counterexample Detection, Core Extraction and Simulation: Three Analyses Applied to a Flash File System Model.*

These cover different approaches and formalisms (*e.g.*, VDM, Event-B, Retrenchment) and different tools (Rodin and Alloy).

There are no formal proceedings for this workshop. However, the organizers are planning a book based on this and other VSR-net workshops, which they hope will be published in 2009.

Author Index