Rodin User and Developer Workshop 2009

University of Southampton, 15-17 July 2009

Extended Abstracts

Organisers

Michael Butler, University of Southampton
Stefan Hallerstede, Heinrich-Heine-Universität Düsseldorf
Laurent Voisin, Systerel

www.event-b.org www.deploy-project.eu

Preface

Event-B is a formal method for system-level modelling and analysis. The Rodin Platform is an Eclipse-based toolset for Event-B that provides effective support for modelling and automated proof. The platform is open source and is further extendable with plug-ins. A range of plug-ins has already been developed including ones that support animation, model checking and UML-B.

While much of the continued development and use of Rodin takes place within the DEPLOY Project, there is a growing group of users and plug-in developers outside of DEPLOY. In July 2009, DEPLOY organised a workshop at the University of Southampton to bring together existing and potential users and developers of the Rodin toolset and to foster a broader community of Rodin users and developers. For Rodin users the workshop provided an opportunity to share tool experiences and to gain an understanding of on-going tool developments. For plug-in developers the workshop provided an opportunity to showcase their tools and to achieve better coordination of tool development effort. Moving towards an open source development project will mean that features that cannot be resourced from within the project can be developed outside the project. It will also help to guarantee the longer-term future of the Rodin platform.

This report contains the abstracts of the presentations at the workshop on 16 and 17 July 2009. The workshop was preceded by a tutorial for Rodin Plug-in developers on 15 July.

We would like to acknowledge the support of the School of Electronics and Computer Science at the University of Southampton (especially the organisational work of Maggie Bond), the DEPLOY project and additional government funding.

Michael Butler
Stefan Hallerstede
Laurent Voisin

July 2009

Rodin User and Developer Workshop, 15-17 July 2009
University of Southampton

Programme

Ken Robinson, UNSW, Sydney
System Modelling and Design: Refining Software Engineering

Jean-Raymond Abrial, Consultant
Doing Mathematics with the Rodin Platform

Stephen Wright, University of Bristol
Experiences with a Quite Big Event-B Model

John Colley, University of Southampton
On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

Fangfang Yuan, University of Bristol
Quantitative Design Decisions Measurement using Formal Method

Kriangsak Damchoom / Michael Butler, University of Southampton
An Experiment in Applying Event-B and Rodin to a Flash-Based Filestore

Philipp Ruemmer, University of Oxford
A Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard

Matthias Schmalz, ETH Zurich
Better automated theorem proving in Event-B

Issam Maamria, University of Southampton
Proposal for an extensible rule-based prover for Event-B

Gudmund Grov, Heriot-Watt University
A Proposal for a Rodin Proof Planner & Reasoned Modelling Plug-in

Jens Bendisposto, Heinrich-Heine-Universität Düsseldorf
Using and Extending ProB

Ilya Lopatkin, University of Newcastle
Towards the SAL plugin for the Rodin platform

Kenneth Lausdahl, Engineering College of Aarhus
An Overview of Overture

Michael Butler, University of Southampton
Roadmap for the Rodin Tool

Aryldo G Russo, AeS Group & Research Institute of State of Sao Paulo, Brazil
Formal Methods Outside the Mother Land

Maria Teresa Llano, Heriot-Watt University
System Evolution via Animation and Reasoning

Atif Mashkoor, LORIA, Nancy Université
BRANIMATION

Fredrik Degerlund / Richard Grönblom, Åbo Akademi University
A Framework for Code Generation and Scheduling of Event-B Models

Andy Edmunds, University of Southampton
Code Generation from Event-B

Mohamed Tounsi, Université Bordeaux
B2Visidia: A Tool to Generate a Java Code from Event-B Specifications of Distributed Algorithms

Alexei Iliasov, University of Newcastle
On Event-B and Control Flow

Joris Rehm, LORIA, Nancy Université
A Rodin plugin for quantitative timed models

Renato Silva, University of Southampton
Composition, Renaming and Generic Instantiation in Event-B Development

Abderrahman Matoussi, Universite Paris-Est
Expressing KAOS Goal Refinement Patterns with Event-B

Eduardo Mazza, IMAG, Grenoble
A tool for specifying and validating software responsibility

Mar Yah Said, University of Southampton
Language and Tool Support for Class and State Machine Refinement in UML-B

Colin Snook, University of Southampton,
An EMF Framework for Event-B

James Sharp, University of Surrey
Using CSP Refusal Specifications to Ensure Event-B Refinement

System Modelling and Design: Refining Software Engineering

Ken Robinson
School of Computer Science & Engineering
UNSW
Australia
email: kenr@cse.unsw.edu.au

June 26 2009

1 Overview

This abstract is a precis of a talk that will illustrate aspects of the teaching of Event-B within *The School of Computer Science & Engineering* (*CSE*) at *The University of New South Wales* (*UNSW*) in the second year of our Software Engineering program. It is taught in a course named *System Modelling & Design* and applied within another pair of courses named *Software Engineering Workshops*. This talk will present the objectives and some of the experiences of those courses.

1.1 Motivation for Using Event-B

The ultimate objective for teaching Event-B to students who will graduate as software engineers is to illustrate the advantage of a quantifiable systems design process. This leads to designs about which designers can reason and can obtain verification of the consistency of their designs through the discharge of proof obligations —really mathematical analysis.

The courses are not taught as *Formal Methods*, indeed the author eschews the "formal methods" description for this activity. The courses are motivated and taught as *engineering methods* for the design of systems, especially software systems. We want students to understand and be able to explain the systems they produce. Even though the students are quite competent in programming, experience reveals that they cannot explain their implementations (programs): the only suggestion they can give for verifying those implementations is *testing*. While not denying a place for testing, it is pointed out that their position would not be acceptable in any other engineering profession.

2 Lessons that Must be Learnt

2.1 Not a Silver Bullet

A point that must be emphasised to students is that the strength of the designs they produce is strongly dependent on how they express their designs. There is a real danger that the concept of proof gives the impression that any Event-B model whose proof obligations have been discharged "has been proved to be correct": it needs to be emphasised that all this really demonstrates is consistency and that the consequences could be quite weak.

2.2 Use of Abstraction

Abstraction is essential, not as an exotic mathematical exercise, but as an exercise in presenting the *essential* behaviour that is to be modelled, rather than the detail of how it might be implemented; unfortunately what many students will want to specify. This is one of the hard lessons for students who are used to programming.

2.3 Behaviour not Programming

Even in Event-B you can still "program": meaning that the guards and the actions are determined to achieve the desired behaviour, but the invariant is too weak and doesn't model the constraints that would ensure that behaviour. In the latter case the proof obligations will not express the verification of the desired behaviour of the model.

2.4 Use Many Refinement Layers

As part of the goal of achieving a meaningful model of a system, it will probably be useful to have many levels of refinement in order to build adequately strong invariants and guards and to have greater confidence in the design of the model.

3 The Experience with Event-B

3.1 Contrasting with Classical-B

These courses have been taught in the past using Classical-B. The experience with moving from Classical-B to Event-B has been very positive and rewarding. Strangely, although both versions of B have substantially the same mathematical toolkit, and hence any design that could be expressed in Event-B could be expressed in Classical-B, the experience is quite different. Event-B appears to encourage more abstraction and greater use of refinement. This will be illustrated with examples of developments, some with contrasts to their modelling in Classical-B.

3.2 Implementing Models in the Workshop

In the first semester of the parallel workshop course students work in teams to apply Event-B to the modelling of some system. In the second semester of the workshop the student teams have to implement a prototype of the system that they have modelled in Event-B.

When using Classical-B we used implementation tools provided by the B-Toolkit (B-Core). In 2008 (semester 2) the implementation was done informally but based on a mapping from Event-B models to an object-oriented design. This worked very smoothly in 2008, and could be argued to be more beneficial for the students. Previously, there were some difficulties in understanding how the tool worked and how to refine their model to use the tool, whereas understanding the informal translation is quite simple. This method will be pursued and developed in the coming 2009 semester.

Experiences with a Quite Big Event-B Model

Stephen Wright

Department of Computer Science, University of Bristol, UK

stephen.wright@bris.ac.uk

The Instruction Set Architecture (ISA) of a computing machine is the definition of the binary instructions, registers, and memory space visible to an executing program. Despite there being many ISAs 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: this is a task to which Event-B is well suited. The motivation for the work is the systematic identification of the conditions and behaviors for all possible instruction sequences loaded into the machine, whether part of a correct program or not.

The constructed model may be considered to consist of two parts: a generic description of properties common to most ISAs, and refinement to example ISAs. The generic part is incrementally constructed from an initial very trivial model and the second part then constructs the example ISAs from this. The complete refinement process is demonstrated by the creation and testing of Virtual Machines automatically generated as C source code via a Rodin translation plug-in, which was developed as part of the project.

The technique is demonstrated by refinement to the MIDAS (Microprocessor Instruction and Data Abstraction System) ISA specification. MIDAS is a specification capable of executing binary images compiled from the C language. It is intended to be representative of typical microprocessor ISAs, but using a minimal number of defined instructions in order to make a complete refinement practical. There are two variants: a stack-based machine and a randomly accessible register array machine. The two variants employ the same instruction codes, the differences being limited to register file behavior. Compiler tool chains for each variant have been developed.

Some numbers: the MIDAS ISAs have thirty four instructions; their complete refinements consist of over thirty steps, expanding a single initial event to over one hundred for each variant. This process involves the discharging of nearly five thousand proof obligations. Automatic translation yields over four thousand lines of C source code for each variant.

This presentation will give an overview of the MIDAS project as an example of a model refinement of sufficient scale, depth and detail to be suitable for automatic translation into a usable executable. The place of Event-B model construction within a wider development process is described, including extension of Rodin itself. Various scaling issues are discussed, including editing and manipulation of Event-B notation via the user interface, discharging of the vast number of proof obligations, and the practical limits of Rodin as an Eclipse/Java platform. The modeling techniques used to mitigate some of these scaling issues are described, and promising emerging or proposed Event-B features highlighted. Suggestions for possible future enhancements are made.

On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

John Colley Michael Butler
Electronics and Computer Science, University of Southampton
{jlc05r, mjb}@ecs.soton.ac.uk

Abstract

Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. From the Instruction Set Architecture (ISA) specification, a pipelined microarchitecture is developed that implements the specification. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming. Current verification techniques are predominantly test based within a Register Transfer Level (RTL) simulation and synthesis flow.

One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed. These are termed Read-After-Write (RAW) data hazards. The presence of hazards depends on the instruction mix presented to the microprocessor and psuedo-random test generation techniques have been used in an attempt to achieve adequate test coverage of instruction combinations.

Formal techniques, using both model checking and theorem proving, have been used in microprocessor verification, but as an adjunct to the simulation-based flow. These techniques are applied after the design is completed in the hope of detecting errors not discovered by testing. Higher-level hardware description languages such as Bluespec and Cal, which provide an automatic synthesis route to RTL, can speed up the design process, but it is the verification costs that dominate in the overall flow and the bulk of the verification must still be done at the Register Transfer Level.

Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifes the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. At each refinement step the importance is shown of addressing the inherent simultaneity that characterises the pipelined behaviour and, in particular, the effects that feedback has in pipeline construction. In parallel with this refinement a sequential machine, which has a pipelined structure but without instruction

overlap, is also developed to assist in the process of invariant discovery and to mirror a commonly-used pipeline design technique.

To illustrate the method, the register-register arithmetic instruction of a typical System-on-Chip (SoC) microprocessor is chosen that can exhibit RAW data hazards with overlapping execution. A technique, termed forwarding, where intermediate values are fed back to a stage that needs them, is employed in modern microprocessors to provide a very efficient means of managing RAW hazards. Debugging the forwarding logic has, however, been found to be difficult and expensive. With the introduction of appropriate invariants it is shown that the concrete, pipelined refinement will not preserve these invariants unless the RAW hazards are detected and managed appropriately.

The concrete Event-B model implements forwarding in a way that corresponds directly to the techniques used in microprocessor design and is proved, automatically, in the Rodin environment to be a correct refinement of the abstract ISA specification. The concrete model also has a direct correspondence to an equivalent hardware description in the high-level languages Bluespec and Cal, which like Event-B are based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

Quantitative Design Decisions Measurement using Formal Method

Presentation abstract

Fangfang Yuan[*]

Computer Science Department
University of Bristol
Fangfang.Yuan@bristol.ac.uk

As design complexity rapidly grows, verification consumes an ever increasing part of the time invested into the design of a system. Design decisions are made to optimise architectures in order to gain high performance or reduce power consumption. However, the impact of design decisions on verification is rarely considered when these decisions are being made. In the hardware community up to 70% of the design effort has reportedly been spent on verification. In practice, many verification projects take much longer than anticipated; delays in design verification are often the reason for delayed tape-out of a circuit. It is of great importance to at least make designers aware of the verification effort behind design options.

While methods are available to assess how much more performance gained or how much less power is consumed by a particular design choice, it is remarkable how little knowledge there is on estimating the verification effort needed to gain confidence in the functional correctness of a design. Of particular interest to us were the approaches that lead to the power estimation techniques which today underpin the tools used for low power design. We replicated this approach, and transferred the method to make design decisions verification-aware.

In principle, all microprocessor designs show a certain degree of similarity. Therefore, we estimate the verification effort for a new design, based on the comparison of the new design's features to a reference model for which the verification effort is known. Since current simulation-based methods do not provide quantitative and objective comparability, we used Event-B as a tool which supports a rigorous method that is also generic and extendible for different design option comparisons. In addition, it also supports hierarchical modelling to allow early design exploration, i.e. the comparisons of design choices at higher abstraction levels, well before the final concrete model can be described. Last but not least, by using Event-B and Rodin the reference model is also correct by construction due to the property-preserving refinement enforced in this framework.

Briefly, the reference model is built via a series of refinement steps, where each step splits an instruction set until each instruction falls into a unique concrete subset. During the refinements, the key point was to leave an open choice point at each level of abstraction to accommodate different design options. To enable quantitative comparisons, the number of events is used as an indicator

[*] Studying for a PhD under the supervision of Dr. Kerstin Eder

for verification effort; this is based on the observation that in simulation-based verification a certain number of tests would be associated with each event in the model. Using the reference model, the impact of different design decisions, such as multi-destination instructions and constant registers, on verification effort can be compared at the appropriate levels of abstraction.

Additionally, a Rodin plug-in for collecting statistics, such as the number of events, number and type of POs, etc, has been developed. Further research could be done on investigating how leaving things undefined in a design, which is very common in practice, impacts verification effort. Investigations are also being made into the development of a more automatic method to generate a processor model; the one I developed was a purely manual effort. During the model development I found that visualising the hierarchy description was helping me to present the features of the hierarchical model. Therefore, a visualisation plug-in also interests me a lot.

An Experiment in Applying Event-B and Rodin to a Flash-Based Filestore

Kriangsak Damchoom and Michael Butler
University of Southampton

June 10, 2009

1 Introduction

According to [8], Hoare and Misra outline the importance of experimenting in formal methods which involves the application of theories and tools in order to push forward scientific progress in formal methods. Experiments assist us to understand the strengths and weaknesses of theories and tools. We have chosen a flash-based filestore as a case study for our experiments. This case study was proposed as a challenging system by Joshi and Holzmann [9] in 2005. As presented in [9], the challenge is how to deal with failures that may occur while performing file or flash operations. For example, how do we cope with fault-tolerance when flash instructions being performed fail, or when power loss?

In our formal development, the flash specification we chose is the Open NAND Flash Interface (ONFi) proposed in [10]. This specification is open and is mostly referenced by researchers who are working in this area. Physical characteristics of the flash device constrain the way it is used. Thus, physical characteristics and failure management are important to be concentrated. Reading and writing content of any file from and to the flash device are consistent with an abstract model of a file system.

Our experiment presented here is the development of a verified refinement chain for a flash-based filestore using Event-B and Rodin. This experiment is an extension of the work we presented in [7] where we outlined a model of a tree-structured file system. The extension we address here consists of replacing the abstract file system by the flash specification and dealing with fault-tolerance. Our contribution are evidence of the applicability of Event-B and Rodin to a flash file store.

2 Methodologies

An incremental refinement was employed as our strategy to develop a model of a flash-based file system. The refinement was used in two different approaches, horizontal or *feature augmentation* and vertical refinements or *structural refinement* [6]. The horizontal development was mainly presented in [7]. The vertical refinement is the focus of this paper. Horizontal refinement is aimed at introducing new requirements or properties which are not addressed in the initial model or may be postponed to other levels. Thus, in each refinement step, additional state variables and related events might be extended to incorporate those augmented features. The system models will be enlarged gradually when new properties are added. On the other hand, the purpose of structural refinement is to replace an abstract structure with more design details in each refinement step down to an implementation. This kind of refinement may involve data refinement, event decomposition and machine decomposition.

2.1 Horizontal refinement

Horizontal refinement was used in an incremental way to construct a model of an abstract file system. The model was begun with an abstract tree structure. After that, new features – files and directories, file content, permissions and other properties such as name and created date – were gradually introduced in refinement steps. We eventually achieved five-layered specification describing an abstract file system. We regard the full chain of horizontal refinements as representing *the specification*, not just the most abstract level. That is, the specification is the abstract level plus a series of feature augmentations.

The event-extension feature [1] provided by the tool was mainly used to construct the refinement chain of the model. In each refinement step, when new features or properties were introduced, the related events were extended by adding more details corresponding to those features. The extension may involve adding parameters, guards and actions.

2.2 Vertical refinement

Structural refinement, which is the focus of this experiment, was used to relate the abstract file system with the specification of the flash interface layer. The event-decomposition technique presented in [4] was used to decompose the events *readfile* and *writefile* into sub-events in order to map them with *page-read* and *page-programme* interfaces provided by the flash interface layer. The decomposition is based on the assumption that the content of the file is read from or written into the storage one page at a time.

Decomposing those two events were done in two refinement steps. After that, the flash specification which is based on the ONFi standard was introduced in the following step. The abstract content of each file was replaced by the content on the flash array which is represented by the *fat* table. This table is a mapping of each file to a table that maps each logical page-id of the file to its corresponding row address in the flash. The corresponding row address represents the location (in the flash) on which the content of that page is.

2.3 Machine decomposition and Further refinements

The machine decomposition technique [4] was applied to decompose the machine of the last refinement into two machines representing the specification of the file system layer and the flash interface layer. The reason we do this is to carry out further refinements of the flash model separately from the file system model. The machine decomposition we applied here follows the style described in [4]. Namely, variables are partitioned and sub-machines interact with each other via synchronisation over shared parameterised events.

After we have done a machine decomposition, the flash model was refined individually by adding more details of the flash specification. For example, each LUN has at least one page register used for buffering data. Writing a page is done in two phases. The first is writing the given data into a page register within the selected LUN and the second is programming the data on the page register into the flash array at the given row address. Event *block-erase* is also specified. This event has the effect of erasing the given block in order to be reused (or rewritten). To reclaim a dirty block, the block that contains obsolete data and may has one or more pages with valid data, all valid pages within the block must be relocated. After relocation has been completed, the given block becomes obsolete and ready to be erased.

Note that the *wear-levelling* process [2] is an important feature that has not been completed in our development yet. It is in our on-going work.

3 Conclusion and Discussion

Feature augmentation was a mechanism used for constructing a model of an abstract file system which was presented in [7]. Instead of specifying everything in one level that may give rise of proof difficulty, we decided to partition the whole system features into sub-features which were chosen to be introduced in refinement steps. We have found that this approach makes model easier to construct and prove. Additionally, we also have found that the event-extension feature provided by the Rodin tool makes model easier to construct and modify. Namely, some modifications can be made at the abstract levels individually without an affecting of modifying the concrete levels. This

[1]The extension feature is available in the Rodin toolset release 0.9.0 or higher.

[2]A technique used for prolonging the life time of flash memory covering reclaiming and erasing blocks within a flash chip.

is in contrast to while we were developing the model of [7] using the Rodin tool release 0.8.2 with no event-extension.

As evidence of applying the event-decomposition technique to our case study, we have found that the event-decomposition technique is very effective for breaking an atomic event. It can be applied to other work whose events may require to decompose in order cope with fault-tolerance or concurrency. An atomic event can be partition into sub-events that can be performed in an interleaved fashion.

The machine decomposition was used to separate part of the flash interface layer from the file system layer. The purpose is to deal with further refinements of the flash interface layer separately. Sine those two layers are in different place and interact with each other via the shared parameterised events. Based on this evidence, we believe that machine decomposition is applicable for other developments whose specification involves a number of sub-modules that can be partitioned and refined individually. Recently, the Rodin dose not provide any tool to decompose machines directly, we still need to decompose machines manually using the editor of the Rodin tool. Thus, in the future, it would be useful if a machine-decomposition tool could be developed.

In each step of iteration of modelling, modification and proof, POs generated by the tool were used as guidelines for modelling and reasoning about the model. For example, they were used to determine which gluing invariant should be added to the machine, which event guard should be added in order to strengthen the model, as well as which form of expressions should be used to make prove easier. That means this technique let us get a higher number of automatic proof.

Although proof statistics show a high degree of automatic proof (93%), some improvements still be required. Main tool is automatic proof and retaining proofs when models change but ML proofs require to be reproved every time the models change. This wastes a lot of time if there are many POs to be discharged.

Lastly, additional improvement that can be added to the language is the separation of external and internal parameters. The separation would make events more readable and easier to specify interactive systems. Another improvement, providing generic theories such as tree-theory would be useful for specifying and reasoning about systems that require tree manipulation.

References

[1] Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. To be published by Cambridge University Press (2009)

[2] Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260. Springer (2006)

[3] Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamentae Infomatica. 1001–1026 (2006)

[4] Butler, M.: Decomposition structures for Event-B. In: Intergrated Formal Methods, iFM2009. LNSC, vol. 5423, pp. 20–38. Spinger (2009)

[5] Butler, M., Yadav, D.: An Incremental development of the Mondex system in Event-B. Formal Aspects of Computing 20(1):61–77 (2008)

[6] Butler, M., Abrial, J.-R., Damchoom, K., Edmunds, A.: Applying Event-B and Rodin to the filestore. VSRNet, ABZ 2008 (2008)

[7] Damchoom, K., Butler, M., Abrial, J.-R.: Modelling and proof of a tree-structured file system in Event-B and Rodin. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 25-44. Springer (2008)

[8] Hoare, T., Misra, J.: Verified software: theories, tools, experiments; Vision of a Grand Challenge project (2005)

[9] Joshi, R. and Holzmann, G. J.: A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005)

[10] Hynix Cemicondutor et al.: Open NAND Flash Interface Specification, Revision 2.0. Technical report, ONFI (Feb. 2008), http://www.onfi.org

Title: A Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard

Philipp Ruemmer, Daniel Kroening
University of Oxford

Abstract:

Sets, lists, and maps are elementary data structures frequently used both in programs and in (declarative) specifications. Program analysis tools therefore need to decide verification conditions containing variables of such types. Unfortunately, the SMT-Lib standard, which is the format most commonly used to integrate theorem provers and SMT-Solvers into program verification systems, does not provide sets, lists, or maps as primitive datatypes. The solution normally chosen for verification systems is to encode these datatypes into arrays or uninterpreted functions, which has several drawbacks: it is not possible to employ dedicated decision procedures for these theories, much structure of the verification problems is lost, and the encodings often exploit prover-specific features. We therefore propose a new theory of finite sets, lists, and maps for the SMT-Lib standard.

Proposal for an extensible rule-based prover for Event-B

Issam Maamria, Michael J. Butler, Andrew Edmunds and Abdolbaghi Rezazadeh
e-mail:im06r, mjb, ae2, ra3@ecs.soton.ac.uk

University of Southampton

Abstract

Event-B [3] is a formalism for discrete system modelling. Key features of Event-B include the use of set theory as a modelling notation, the use of refinement to model systems at different levels of abstraction, and the use of mathematical proof to verify consistency between refinement levels. Event-B provides two constructs to model systems; contexts describe static properties (constants and carrier sets) whereas machines model dynamic behaviour (variables and events). Proof obligations are generated to verify the consistency of models with regards to some well-defined criteria [3].

The Rodin [4] platform is the software tool that accompanies Event-B. It is developed as a collection of Eclipse plug-ins to benefit from the modularity of the Eclipse architecture. Our work will exploit this important feature of Rodin to develop a mechanism by which the mathematical language of Event-B can be augmented with new datatypes and definitions of new expression operators (e.g., symmetric difference) and new basic predicates (e.g., reflexivity of a relation). The mechanism will also allow the prover infrastructure to be easily extensible with new proof rules (inference rules and rewrite rules) without the detour through writing Java code as is the current practice. This work will go a long way in realising the mathematical extensions envisaged in [2].

In line with the Rodin approach to modelling using two constructs and generating proof obligations to verify consistency, we propose a new construct we shall call *Theory*. This new construct was partially inspired by Isabelle's [1] theories where the user can specify datatypes, functions (amongst other things) alongside theorems to be proved. The Isabelle theory can also make contributions to the prover by extending the set of simplification rules it can apply. The theory construct in Event-B will achieve the two objectives outlined previously, and will be developed in a similar fashion to contexts and machines.

Keywords: *Event-B, rule-based prover, rewrite rules, inference rules, mathematical extensions*

References

[1] *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, London, UK, 2002.

[2] J.-R. Abrial, M. Butler, M. Schmalz, S. Hallerstede, and L. Voisin. Proposals for Mathematical Extensions for Event-B, 2009.

[3] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.*, 77(1-2):1–28, 2007.

[4] Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.

A Proposal for a Rodin Proof Planner & Reasoned Modelling Plug-in

Gudmund Grov[1], Andrew Ireland[2], Michael Butler[3], Alan Bundy[4] & Cliff Jones[5]
[1]`G.Grov@hw.ac.uk` [2]`A.Ireland@hw.ac.uk` [3]`mjb@ecs.soton.ac.uk`
[4]`bundy@inf.ed.ac.uk` [5]`cliff.jones@ncl.ac.uk`
[1,2]Heriot-Watt University [3]University of Southampton
[4]University of Edinburgh [5]University of Newcastle

We propose two plug-ins for the Rodin toolset. Firstly, we aim to create a *proof planning plug-in*, which we believe will enhance proof automation by exploiting common patterns of reasoning. Secondly, the reasoning patterns will be extended with modelling patterns, allowing us to explore the interplay between reasoning and modelling, i.e. *reasoned modelling*. In addition to proof automation, this *reasoned modelling plug-in* will provide modelling guidance based on the reasoning process.

1 Proof planning in Rodin

Event-B promotes an incremental style of design where the activities of modelling and analysis are closely intertwined. Therefore Event-B and Rodin provide an ideal system for our *reasoned modelling* research, which will explore the interplay between reasoning and modelling.

Our starting point is the notion of *proof planning*, a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [4]. A proof plan provides a way of representing common patterns of reasoning, patterns that provide heuristic guidance for proof search. A novel feature of proof planning is the ability to analyse and patch proof-failures, known as the *proof critics* mechanism [9].

One of proof planning's biggest success stories is proof by mathematical induction [5]. Induction is essential for reasoning about inductively defined data types (which are part of the Rodin roadmap [2]). Specific successes are: **mathematical induction:** program verification, synthesis, and optimisation; hardware verification; correction of faulty specifications; **non-inductive proof:** summing series; limit theorems; **automatic proof patching:** conjecture generalisation, lemma discovery, induction revision, case splitting, loop invariant discovery.

We believe Event-B models contain many reasoning patterns, thus a proof-planning plug-in will enhance proof automation within Rodin. Note that proof planning has a track record in promoting tool integration [10, 12].

2 Reasoned modelling in Rodin

Proof plans capture reasoning patterns, and proof critics capture reasoning patches, like lemma speculation and conjecture generalisation. In an Event-B development, there is often a strong interplay between the modelling and reasoning perspectives. For example, the patching of a model is guided by analysing failed proof obligations. This is illustrated in Abrial's "Cars on a Bridge" example [1], where modification of events (guards and actions) and invariants are guided by failed proof obligations. In the Mondex smart-card case study [7], gluing invariants are discovered in an iterative process where each modification is a result of the analysis of failed proof obligations.

We thus propose a new paradigm which explores the interplay between modelling and reasoning. This will be achieved by extending proof plans and proof critics to incorporate both modelling and reasoning patterns. We call this *reasoned modelling*.

In addition to proof centric patches, like lemma speculation and conjecture generalisation, a *reasoned modelling critic* will also suggest changes to models. We expect to capture modelling heuristics in such critics, however, full automation will not be possible or desirable, since there can be many suggestions and only the user (modeller) will have the required insight of the intended behaviour or requirement. The plug-in will provide high-level modelling suggestions to the user, based on failures and analysis of (low-level) proof obligations. Heuristics, can be used to prioritise suggestions. Note that proof planning has a track record in using proof failure to evolve models [6, 11].

In addition to containing reasoning patterns, a *reasoned modelling plan* will contain modelling patterns. One example is the discovery of gluing invariants. Such a plan may, as in [7], cause intermediate refinements. Thus, a plan may contain local and global changes, in one or many machines and contexts. Moreover, it will contain information on both how to model and how to verify the corresponding proof obligations. Also, due to the flexibility of our approach, we believe we can encode a rich set of general refinement patterns. Note that since reasoned modelling is centred around modelling *and* reasoning, it deviates from other suggested Event-B design and refinement patterns, like [3, 8], which have a clear modelling focus.

References

[1] Jean-Raymond Abrial. *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To be published.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. A Roadmap for the Rodin Toolset. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*, page 347. Springer, 2008.

[3] Jean-Raymond Abrial and Thai Son Hoang. Using Design Patterns in Formal Methods: an Event-B Approach. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[4] Alan Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In *9^{th} International Conference on Automated Deduction (CADE'09)*, pages 111–20. Springer-Verlag, 1987.

[5] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling – Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[6] Alan Bundy and Michael Chan. Towards ontology evolution in physics. In W. Hodges, editor, *Proceedings of Wollic 2008*, LNCS. Springer-Verlag, July 2008.

[7] Michael J. Butler and Divakar Yadav. An Incremental Development of the Mondex System in Event-B. *Formal Aspect of Computing*, 20(1):61–77, 2008.

[8] Alexei Iliasov. Refinement Patterns for Rapid Development of Dependable Systems. In *EFTS '07: Proceedings of the 2007 workshop on Engineering Fault Tolerant Systems*, page 10, New York, NY, USA, 2007. ACM.

[9] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNAI*, pages 178–189, St. Petersburg, Russia, July 1992. Springer Verlag.

[10] Andrew Ireland, Bill J. Ellis, Andrew Cook, Rod Chapman, and John Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[11] F. McNeill and A. Bundy. Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution. *IJSWIS*, 3(3):1–35, 2007. Special issue on ontology matching.

[12] Konrad Slind, Mike Gordon, Richard Boulton, and Alan Bundy. System Description: An Interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *CADE 15*, volume 1421, pages 134–138, Lindau, Germany, July 1998. Springer.

Using and Extending ProB[⋆]

Jens Bendisposto and Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto, leuschel}@cs.uni-duesseldorf.de

Writing a formal specification for real-life, industrial problems is a difficult and error prone task, even for experts in formal methods. When specifying a formal model it is crucial to validate, that the model's behaviour corresponds to the requirements and to our expectations to preseve us from additional costs of changing at a late point of the development. Animation and visualization can help us to gain confindence that the model satisfies our requirements.

For the B-Method, we presented the Prolog based animator and model checker ProB, which aims to support the writing of formal specifications. ProB allows a user to gain confidence that the specification does meet the requirements. The user can check the presence of desired functionality and inspect the behavior of a specification.

We developed a plug-in version of ProB allowing to animate and model-check Event-B models. The plug-in version so far supports the most often used subset of the features provided by the tcl/tk version of ProB. In addition, it offers to export the models for usage in the tcl/tk version.

The architecture of the ProB plug-in was designed to be as open as posible for later extensions by tool developers. All activities are encapsulated in commands. Developers can write their own commands (sometimes they require to change the Prolog side of ProB) or combine existing commands. The implementation of the commands delivered with the plug-in makes heavy usage of combining basic commands. For instance, the command to explores a state calls a number of basic commands such as a command to get the values for the state variables, to get the status of the invariant (broken or not) or to get information about timeouts.

Our presentation gives an short introduction into using the ProB plug-in for Rodin. We will demonstrate how to install and use the plug-in as well as give useful hints for users that are new to ProB. We will also give a quick overview together with pointers, how the tool can be extended and used by other plug-ins.

⋆ 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).

Towards the SAL plugin for the Rodin platform

Ilya Lopatkin Alexei Iliasov
Alexander Romanovsky

School of Computing Science
Newcastle University
{Ilya.Lopatkin, Alexei.Iliasov, Alexander.Romanovsky}@newcastle.ac.uk

June 12, 2009

SAL [4] is a model-checking framework combining a range of tools for reasoning about information systems, in particular concurrent systems, in a rigorous manner. The core of SAL is a language for constructing system specifications in a compositional way. The SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers.

The overall aim of our work is to investigate the potential applications of SAL in a combination with the Rodin platform, a development tool for the Event-B formalism. Unlike SAL, Rodin relies mostly on theorem proving for model analysis. We are looking for a way of complementing the Rodin platform with a plugin that would automate some of the more difficult tasks such as liveness, deadlock freeness and reachability analysis.

1 Translating Event-B into SAL

During our initial experiments we have translated a number of Event-B models into the input language of SAL and verified them. The benchmark for our efforts is ProB[1, 2]. Since ProB is Prolog-based we had started with an expectations of achieving some perfomance advantage for a considerable subset of problems.

Figure 1 presents the comparative performance of the ProB Event-B model checker and SAL run on the result of transforming the same model into SAL. The first model is a synthetic benchmark based on bubble sort algorithm. The other four models are the examples bundled with ProB distribution. These demonstrate the translation and the performance of some of the most "inconvenient" parts of Event-B syntax for SAL: sets, functions, relations and operators on them such as union, intersection, cardinality and etc.

In these models, we used a classical representation of sets in predicates adopted from [5]. Reasoning on timings we obtained during our experiments, we drew preliminary conclusions about efficiency of SAL model checker on our

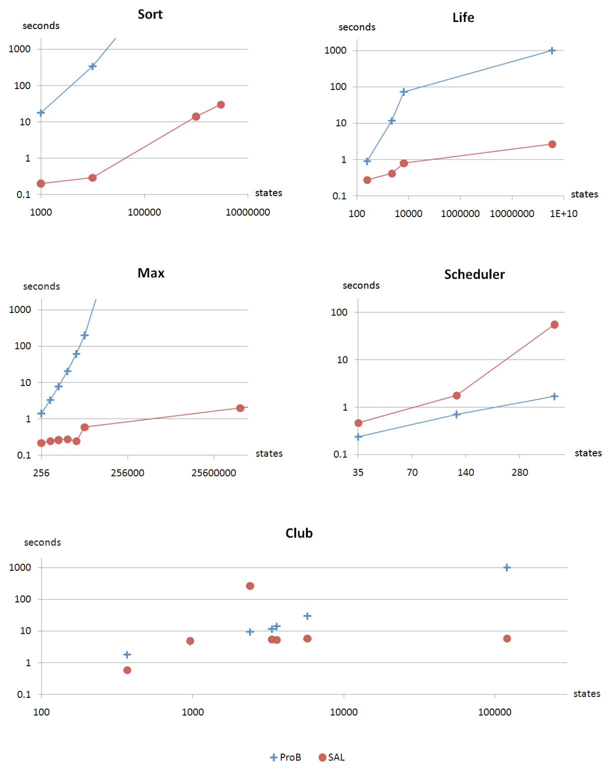


Figure 1: Comparison of ProB and SAL model checkers

models and provided a number of optimisations which led to a significant performance benefit for the models operating on sets.

2 Ongoing work

Our ongoing work is focusing on incorporating the SAL model checker into the Rodin platform. A number of steps are being performed to achieve this. We are aiming at developing a nearly complete mapping of Event-B to the SAL input language. We do not consider it practical to attempt to cover the whole of the Event-B mathematical language due to semantic gap between two. SAL doesn't natively support the full set of Event-B constructs, so it makes reasonable to consider only the part which is feasible and gives benefits in comparison with existing tool. Therefore, we intend for our tool to cooperate with the ProB model checker so that models that cannot be handled with SAL are automatically handled by ProB.

The result of developing the language mapping would be an automated translation of Event-B models into SAL. The next step is in providing a user with a meaningful feedback from the tool.

Based on experience obtained during our manual experiments, we drew a general approach of translation of the main Event-B model elements such as events, invariant, variables, etc [3]. Our current work is automation of such translation. At the appendix we provide a very simple example showing the input and output models of our translator. We intend to develop it further into a model checking plugin for the Rodin platform.

References

[1] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[2] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[3] D. Plagge, M. Leuschel, I. Lopatkin, A. Iliasov, A. Romanovsky. SAL, Kodkod, and BDDs for Validation of B Models. Lessons and Outlook. To be presented at *CAV'09*, June 2009.

[4] The SAL website. `http://sal.csl.sri.com`.

[5] G. Smith and L. Wildman. Model checking Z specifications using SAL. In *ZB*, pages 85–103, 2005.

Appendix

```
MACHINE M0
SEES c0

VARIABLES
  var1
  var2

INVARIANTS
  inv1   :    var1∈N
  inv2   :    var1>3
  inv3   :    var2∈N1
  inv4   :    var2>var1

EVENTS
  INITIALISATION
  BEGIN
    act1   :    var1:=4
    act2   :    var2:=5
  END

  evt1
  ANY a
  WHERE
    grd1   :    a∈N
    grd2   :    a>2
    grd3   :    var1<10
  THEN
    act1   :    var1:=var1+a
  END
END
```

```
translated: context =
begin
  main: module =
  begin
    local var1:{var1:NAT|var1>3}
    local var2:NAT1

    initialization
      var1=4;
      var2=5;

    transition
    [
      ([](a:{a:NAT|a>2}): evt1:
          var1<10 -->
            var1'=var1+a;
      )
    ]
  end;

  invariant: theorem main |- G(
    var2>var1
  );
end
```

A roadmap for the Rodin toolset*

Jean-Raymond Abrial Michael Butler Stefan Hallerstede
Laurent Voisin

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. Keep aspects of the are support for abstract modelling in Event-B; support for refinement proof; extensibility; open source. To support modelliing 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

Formal Methods: Outside the Mother Land

Aryldo G Russo Jr.

AeS Group & Research Institute of State of São Paulo (IPT),
agrj@aes.com.br

Abstract. The use of formal methods has constantly increased, although with basically two constraints: their use has been concentrated mostly in Europe, their Mother land and they have been used only by big companies which are in charge to develop some kind of safety critical applications, what, in a first look seems correct. The aim of this paper is to present the usage stage of formal methods in other parts of the world, mainly South America, and Far East. A personal comparison of some formal method tools, namely: Atelier B[1], RODIN[2], and SCADE[3]is also presented. The comparison methodology is based on three different points of view: capability, I mean, how these tools can satisfy project constraints, usability, basically, what's the difficulty the user faces when trying to use the tool, and adequacy to the current development process.This work describes also real applications in industry, sometimes not the formal method usage itself, but how the formal method culture can drasticaly helps on the development process. Finally, some of the gaps in industry wishes that could be fulfilled by some applications are sorted.

1 Introduction

The primary objective of this paper is to present the curent State of Practice of Formal Methods in coutries outside Europe, namely, Brazil and Korea. In this sense, I would like to present it as the utilization of formal methods in general, and, moreover, not only the application of one method or another, but how the principles that guide the formal methods usage can help in the software development process.

But, before talking directly about the subject of this paper, I'd like to give some background information about the reason I started to work with Formal Methods, and my involvement in academia. Then, I will present a general scenario of how these methods are being used nowadays in the places I meintioned before. In the remaining of this paper, I will present some industrial areas where we can find already some use of formal methods.

Finally, I will present a comparison of three tools, namely, AtelierB[1], RODIN[2] and SCADE[3]. This comparison is based on three aspects, tool capability, usability and adaptation to the current development process. I will show also, some real application of these tools, and the work that was performed to change the way that industry was used to think about software development, even in safety critical areas.

At the end, I will present some gaps that, from my personal point of view, can be fulfilled with some new or in development phase, plugins and language extensions.

1.1 The AeS Group

The AeS Group has developed railway sub-systems since 1998. Among the systems developed by the group, the door system became one of the most important in the railway market due mainly to the architecture used (modular, and with distributed processing) and, since this kind of system deals with human lives, the strong concern of the group with reliability and safety.

Due to the advances in technology, many safety functions that were handled by hardware are now responsibility of the embedded software. This fact triggered motivation to use formal methods in standards relevant to software safety [4]. Some standards can be followed to increase the safety level of an equipment. One of them is the IEC 61508 [5]. This standard presents four levels of safety, the so called Safety Integrity Level - SIL, and above level 2, a formal specification is required or suggested to achieve a certain level of completeness, robustness, and safety, that grows as the level grows. The goal of using formal methods is to produce an unambiguous and consistent specification which is as complete, error-free and without contradictions as possible, however simple to verify.

Nowadays, AeS Group has also the support of DEPLOY project and some universities like University of Southampton, and University of York. Of course, AeS Group is also supported by companies like ClearSy and Esterel.

1.2 General scenario

In order to picture out the differences in formal method application outside Europe, I will give you some information about the current software engineering process that is being applied at this moment in the process of safety-related application development.

Basicaly, the software development process presented in the IEC 61508[5] is well known in South American companies, but as the time to market is, normaly, extremaly short, those recomendations are put aside, and the craft process is followed. This process is basically the reception of the primary specification, the coding phase is made relying on the personal expertise, and the tests are performed as few as possible. It's already a good scenario to use formal methods and try to better the process without changing the manual tasks.

Talking about Far East, those process are barely known. As presented in [6], the adoption of the recomendations referenced in the software development process are in it's infancy phase, meaning that even the standard understanding are not clear enough.

Based on this view, it's crystal clear that is not possible to go directly to the pure application of formal methods. First, it's necessary to create a better culture of software development process.

2 Where formal methods (could be) are used

Many different industrial areas, where safety and reliability issues are highly important characteristics, have been using, or at least have tried formal methods in order to increase their confidence that those requirements are met. Those industries are, mainly, Nuclear[7], Medical devices[8], Avionics, Aerospacial and transportation [9]. Some examples are the emergency contention measures in nuclear power plants, health support devices in medical applications, automatic pilot on avionics, positioning systems in aerospacial and signaling systems in tranportation just to cite a few.

This means that there is plenty of space for the adoption of supporting tools that could help either the development process (either system or software) in the sense of automatizing some parts of it, and also, in some cases, for speeding up those development tasks difficult to perform, while the developer uses his efforts in other more conceptual phases.

In order to change this scenario, the distance between mathematical notation and the normal procedures used so far has to be shortened, and for that some highly desired characteristics should be included in the current tools in order to reflect the activities that are normally performed in those industries.

Fortunately, it might not be so difficult as, at least, the development model that has been adopted in those industries (V model**??**) is not different from the model used in a formal model development.

3 Tool comparison

In order to verify how the current tools can be modified to reflect the industrial needs, I prepared a brief comparison of some existent tools. I have restricted this comparison to some tools that I know better and that have been used in my application field, that is, railways application. Those tools are, Atelier B, RODIN and SCADE.

3.1 Chart comparison

Aspect	**capability**	**usability**	**adaptation**	***Results***
AtelierB	2	1	2	*5*
RODIN	2	2	1	*5*
SCADE	2	3	3	*8*

Table 1. Comparsion table

4 Experiences

Basically, my experiences in formal methods are both as a practitioner and as a researcher. In the last 3 years I've been trying to introduce formal methods in the projects I have worked on, and I can say that even if they can not fulfill all industrial needs they can help a lot to better model the development process and the resultant product (or software).

Despite all odds, and as pointed in [10], it was not necessary to have someone with strong knowledge in mathematics, altough the basic concepts were needed. Moreover, it was not necessary a big team in none of the described projects in order to sucefully cary on the project. In the second project I cited before, just one person did all the work.

In all of these projects, the most dificult task, and the one that took more time was the requirement elicitation and analysis. Even if it not direct related to formal methods, as it has to be carried on does matter the process you adopt, the goal to build a formal model helps during the classification and elaboration of each requirement forcing them to be complete and non ambiguous.

At the end, the time (and money) that is spent in the earlier phases of the development process is greater than in a normal development, but the time (and much money) that is spent in tests and rework are definitely less. In the case of the second example (Door system), even using the formal methodology only as a support tool, the resulting test cases were much more effective, and the period of tests was shortened by 2 months (from 6 months to 4 months).

Based on these experiences, and others, I summarize in section 5 some features that I think could be included in RODIN platform.

5 Gaps or needs

In this section it is summarized some of expectations about the future of supporting tools and point out some characteristics that is presumed as necessary. Most of them are being prepared, but even though some key points for each one might be pointed out.

– *Requirements* - It's a fact that requirement problems are responsible for more than 40% of the total problems in a project **??**. With this in mind, this is the most important feature that should be integrated in RODIN platform.
– *Traceability* - Even if it's related also to requirements, I think RODIN platform might have also capability to perform this task
– *intermadiate languages* - This is something that's already been done by UMLB plugin, but I think that one interesting feature is missing. Besides the ability to create state machines, for example, the ability to execute these models would be gratefully appreciated. With that, we would be able to verify if our assumptions are correct, with no need to go inside the proof obligations.
– *test case generation* - This, in my opinion, is one of the biggest gaps in industry right now.

6 Conclusion

As it's more a positional paper than a research paper itself, I will present my personal conclusions to you. The application of formal methods in industry is growing, however most of the times as a result of some projects involving academia and industry, like DEPLOY project.

It's clear that outside Europe, formal methods usage is still incipient, and more effort in showing the benefits of that use is needed. In order to facilitate this approach we need tools that do not scare the customer in a first sight, otherwise the fear not to perform a good job will be always greater than the possibility of creating better products.

If these barriers could be broken, I think that the use of formal methods would spread out really fast.

If the introduction of the features I mentioned before could be a reality, it would be a great step for this project.

If the managers are open mind, and admit waiting a bit more in the begining of the development to see real results, (light or heavy) formal methods application could be a lot cost-effective and can, at the end, decrese the costs of the whole project by decreasing the costs in test and maintenance phases.

7 Aknowledgements

I'd like to thank to all univiersities and companies cited in this paper for their support providing me recomendations and tools.

I'd like to thank Mr. Michael Butler that, on behalf of DEPLOY team, for the invitation to present this paper in the Rodin Workshop.

Finally I'd like to thank to Mr. Hrvoje Belani, who devoted his time to review this paper and mostly for his comments which help me to include substantial points in this work.

References

1. ClearSy: Atelierb 1
2. Butler, M., Hallerstede, S.: The rodin formal modelling tool. deploy-eprints.ecs.soton.ac.uk 1
3. Esterel: Getting started with scade. (Sep 2007) 1–148 1
4. Bowen, J.P., Stavridou, V.: The industrial take-up of formal methods in safety-critical and other areas: A perspective. In: FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe. Volume 670 of Lecture Notes in Computer Science., Odense, Denmark, Springer (1993) 183–195 2
5. Commission, I.E.: IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission Standards (1998) 2
6. Hwang, J., Jo, H., Jeong, R.: Analysis of safety properties for vital system communication protocol. Electrical Machines and Systems, 2007. ICEMS. International Conference on (2007) 1767–1771 2

7. Abrial, J.: Formal methods: Theory becoming practice. Journal of Universal Computer Science (Jan 2007) 3
8. Jetley, R., Iyer, S., Jones, P.: A formal methods approach to medical device review. COMPUTER (Jan 2006) 3
9. Lecomte, T., Servat, T., Pouzancre, G.: Formal methods in safety-critical railway systems. Proc. Brazilian Symposium on Formal Methods: SMBF (Jan 2007) 3
10. Bowen, J., Hinchey, M.: Ten commandments of formal methods ten years later. COMPUTER (Jan 2006) 4

System Evolution via Animation and Reasoning

Andrew Ireland, Maria Teresa Llano and Rob Pooley
School of Mathematical and Computer Sciences
Heriot-Watt University

The development of correct abstract models is a creative task and one of the major challenges in the construction of design models. This, along with the fact that software systems tend to evolve makes the challenge of writing good abstract models even harder. Our research addresses this challenge. We are exploring different ways of giving support to the user on design decisions, more specifically, on decisions about UML-B designs. In order to do so, we are investigating the role of patterns and anti-patterns within UML-B models, i.e., common patterns of design behaviour and design flaws, respectively. Also, we will investigate the way in which formal reasoning can inform UML-B designs about defects in the models, so that through this feedback (unproven proof obligations, counter-examples, etc.) we can suggest potential improvements to the design.

1 Illustrative Example

In this example we use the contract net protocol [2] to illustrate how the results of reasoning techniques can be used to suggest improvements to UML-B designs. We base our UML-B model on the Event-B specification of the protocol developed in [1]. The contract net system is a protocol for distributed negotiation processes. Its purpose is to find an agent or group of agents (*the participants*) that best fit the requirements to complete a task that is proposed by another agent (*the initiator*). The initiator begins the process by sending a call for proposals to the other agents in the system. The agents that receive the request can answer by sending a proposal to the initiator. Then, the initiator selects a proposal or group of proposals to be in charge of completing the task. The participants are informed about the selection and then the communication finishes when the participants inform the initiator that the task has been completed.

In order to illustrate the behaviour of the protocol we use activity diagrams. While activity diagrams are not currently part of the UML-B profile, we believe that the integration of a diagram that allows the modelling of the internal behaviour of Event-B specifications is beneficial. In particular, such diagrams can help users in having a better understanding of the systems they develop as well as benefit the analysis and feedback given to UML-B designs. In Figure 1 we present a small fragment of the activity diagram for the refinement of the contract net protocol proposed in [1]. This fragment of the model contains the reception of acceptance and rejection messages of the proposals sent by the participants.

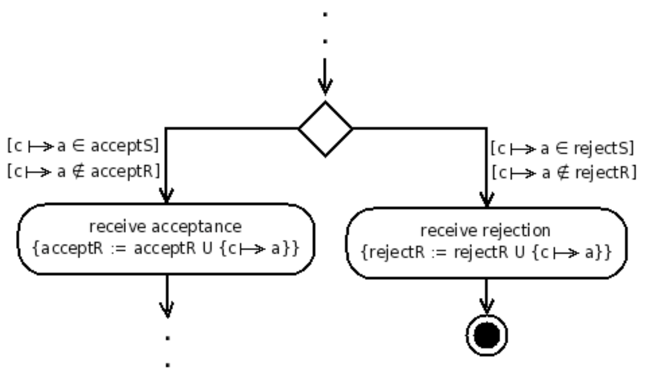


Figure 1: The contract net protocol - Acceptance and rejection of messages.

A proposal of a participant is either *accepted* or *rejected*, which means that the sets *acceptR* and *rejectR* should be disjoint, i.e. $acceptR \cap rejectR = \emptyset$ is introduced as an invariant of the model. As the model stands, both *receive acceptance* and *receive rejection* events may violate the invariant. For instance, assuming $c \mapsto a$ is a member of *rejectR*, it is possible for the event *receive acceptance* to be triggered since it does not evaluate that $c \mapsto a$ should not be part of the set *rejectR*. This represents a bug in the model.

There are different alternatives that can be suggested in order to correct this bug. One of these alternatives is to request the user to confirm if the invariant is indeed the correct one. If confirmation about the correctness of the invariant is received, another alternative is the strengthen of the guards of the *receive acceptance* and *receive rejection* events. The generation of guards strengthening can be determined via proof failure analysis. This alternative is shown in Figure 2. In this model a guard is added to each event in order to eliminate the possibility of invalidating the invariant, i.e., in the *receive acceptance* event $c \mapsto a$ should not be a member of the *rejectR* set (i.e. $c \mapsto a \notin rejectR$) and in the *receive rejection* event $c \mapsto a$ should not be a member of the *acceptR* set (i.e. $c \mapsto a \notin acceptR$).

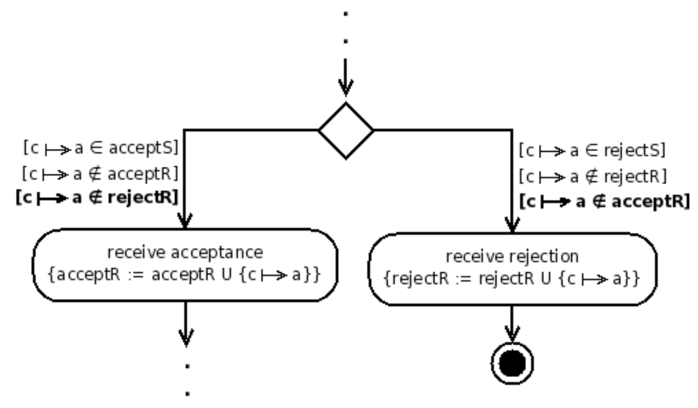


Figure 2: Alternative model of the acceptance and rejection of messages: Guards strengthening.

Eliminating the design bug requires user interaction, since only the user knows what is intended. However, the interaction can take place at the level of alternative models rather than failed proof obligations. In this example we have suggested two alternatives, one is the verification of the invariant and the other one is a modification of the model in the form of an activity diagram. However, as mentioned before, there are several alternatives that can be suggested; therefore, the final decision is left to the user.

2 Conclusions

We have shown by an illustrative example how formal techniques can be used to inform the design, and how we can use their feedback to give suggestions to the user at a more abstract level than failed proof obligations. As alternative paths of research, we are also interested in helping UML-B users generate and patch invariants for their models since this has proved to be a hard task in the development of UML-B designs. Furthermore, as explained before, we want to study the use of patterns and anti-patterns to define abstract system-level models. Finally, we also believe that by having a UML diagram that models the internal behaviour of the system, one may be able to find inconsistencies in user specifications.

References

[1] E. Ball and M. Butler. Event-b patterns for specifying fault-tolerance in multi-agent interaction. *Methods, Models and Tools for Fault Tolerance*, LNCS 5454:104–129, 2009.

[2] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Computers*, 29(12):1104–1113, 1980.

BRANIMATION [⋆]

Atif Mashkoor, Jean-Pierre Jacquot

LORIA – DEDALE Team – Nancy Université
Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr

1 The problem

Brama is one of animation tools for Event-B specifications supported by Rodin Platform. It enables quick validation of models. While animating an Event-B specification with Brama, sometimes we stumble upon some technical issues which prevent its execution. The situations where Brama cannot animate a specification can be arranged in a typology of five typical cases:

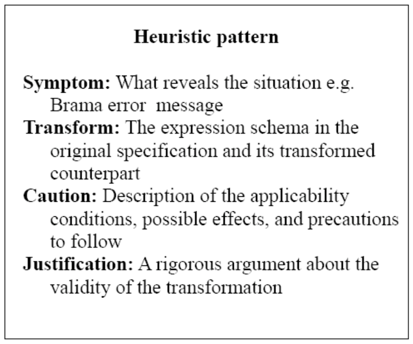
1 Brama does not support the finite clause in axioms
2 Brama must interpret quantifications as iterations
  2.1 Brama only operates on finite sets
  2.2 Brama cannot compute finite sets defined in comprehension with nested quantification
  2.3 Brama explicitly requires typing information of all those sets over which iteration is performed in an axiom
3 Brama cannot compute dynamic functional bindings in substitutions
  3.1 Brama does not support dynamic mapping of variables in substitutions
  3.2 Brama does not support dynamic function computation in substitutions
4 Brama does not compute functions defined analytically
  4.1 Functions with analytical definitions in context cannot be computed in events
  4.2 Functions using case analysis can not be expressed in a single event
  4.3 Invariants based on function computations can not be evaluated
5 Brama has limited communication with its external graphical animation environment

2 The solution

For each situation, we have defined a "heuristic" to transform the original specification into one that can be animated [1]. The heuristics are described following a rigid pattern as shown by fig 1.

We design the heuristics to preserve the behavior of the specification as specified by original model, and not its formal properties. So if some of the proof obligations can not be discharged, this is still acceptable.

[⋆] This is a proposal for tool/plug-in development


Heuristic pattern

Symptom: What reveals the situation e.g. Brama error message
Transform: The expression schema in the original specification and its transformed counterpart
Caution: Description of the applicability conditions, possible effects, and precautions to follow
Justification: A rigorous argument about the validity of the transformation

Fig. 1. The heuristic pattern

3 The observations

We experiment our heuristics on two case studies: a formal domain model of land transportation [2] and a situated multi-agent platooning system [3]. From these two experiences, we come to know that:

– Most well written specifications need to go through this transformational process in order to be animated correctly.
– Once heuristics are applied some of the proofs are impossible to discharge.
– We should not introduce these heuristics early during the specification phase before animation as they will further complicate the already complex text.
– It is a good idea to animate each refinement step like verification to gain confidence in your specification. Problems then can be detected and fixed right on the spot.

4 The need for tool

Though the proposed heuristics solve the aforementioned Brama problems, yet their manual application is tedious, cumbersome and may be error prone if not applied carefully. Therefore a plug-in/tool is required which can apply these transformations automatically to specifications. We don't want this tool to be highly intelligent or sophisticated, it can only performs basic functions, for example, removal of finite clause, provision of typing information, event replications, etc.

References

1. Mashkoor, A., Jacquot, J.P.: Incorporating Animation in Stepwise Development of Formal Specification. Research Report INRIA-00392996, LORIA, Nancy, France (2009)
2. Mashkoor, A., Jacquot, J.P., Souquières, J.: B événementiel pour la modélisation du domaine: application au transport. In: Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09), Toulouse, France (2009) 1–19
3. Lanoix, A.: Event-B specification of a situated multi-agent system: Study of a platoon of vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE Computer Society (2008) 297–304

A Framework for Code Generation and Scheduling of Event-B Models

(Extended abstract)

Fredrik Degerlund[1,2] and Richard Grönblom[2]

[1)]Turku Centre for Computer Science
[2)]Åbo Akademi University, Dept. of Information Technologies
Joukahainengatan 3-5, FIN-20520 Åbo, Finland
{`fredrik.degerlund`, `richard.gronblom@abo.fi`}

1 Introduction

Currently, software is often developed using methods that are not mathematically strict. A large number of different formal methods for software development have been proposed, of which a subset are based on reasoning about how program statements modify the state space of the program. Examples of such state-based formal methods are the Action Systems formalism [5], the B-Method [1] and Event-B [9]. Action systems are convenient for reasoning about programs, including parallel ones, on a theoretical level, whereas the B-Method puts more emphasis on practical details such as a more fixed syntax, a specific mathematical language, etc. A number of tools that enable computer-aided modelling using the B-Method have also been developed [4, 6]. The Event-B language has its roots in the B-Method, but is more suitable, e.g., for modelling concurrent software. Tool support for Event-B has been/is being developed within the EU projects RODIN and DEPLOY, in the form of the RODIN platform [2].

Whereas executable programs can relatively easily be created from Atelier B (B-Method) models, code generation and execution has not yet been as well studied for Event-B and the RODIN platform. One factor that makes it harder to generate and execute code from Event-B, compared to the B-Method, is the fact that concurrent programs can be modelled natively in Event-B. There have, however, been some early attempts to make Event-B code generation available. For example, a small code generator plug-in (Event-B to C) has been written by S. Wright [11]. We are currently working on code generation support in the same spirit as the work of S. Wright, but which would, among other things, accept a larger subset of the Event-B language, take scheduling into consideration to a higher degree, etc. Our framework consists of two parts: a code-generation plug-in and a scheduler for the generated code. In the rest of this abstract, we describe these tools more closely.

2 Code Generation and Scheduling

The first part of our framework is a code generation plug-in for the RODIN platform. The code generator accepts a subset of the Event-B language and translates the constructs into C/C++ code. This approach is similar to the approach taken in the code generator of Atelier B, where a translatable subset of B called B0 is defined. The plug-in of S. Wright also accepts a subset of Event-B, but we want to accept a more general set of constructs for translation. We refer to this subset of Event-B as Event-B0. If the RODIN model consist of constructs not included in Event-B0, it first has to be transformed (refined) to adhere to this subset before code can be generated.

When considering code generation and execution for the B-Method and Event-B, we especially want to point out one crucial difference between the languages. Whereas the B-Method was traditionally intended for modelling sequential programs, Event-B also supports concurrent models. Whereas mere translation into C/C++ functions from B-Method models can be considered adequate, or at least acceptable, it would be leaving part of the problem unsolved in the case of Event-B. This is because C/C++ functions, as such, do not take scheduling or concurrent execution into account. The generated code can hardly be considered complete without taking a stand on how and when to execute the functions. Because of this, our plug-in generates code in such a way that it is closely tied to our second tool, the scheduler, which orchestrates the execution.

Our scheduler is inspired by, and shares some code base with, an earlier prototype scheduler called ELSA [7], which was designed for B Action Systems [10]. B Action Systems is a way of modelling action systems using the B-Method, and since action systems support concurrency, this also had to be considered in ELSA. The B-Method is in that regard similar to B Action Systems, whereby techniques deployed in ELSA are also relevant for Event-B. Our solution is to include scheduling code together with the generated code (which is translated to functions). This code is responsible for keeping track of common variables within the functions, termination of the program, etc. The scheduling can, thus, deduce which individual functions may be called at what time, including which functions to schedule simultaneously. To implement parallelism in practice, we use the MPICH2 library [3] which adheres to the MPI (Message Passing Interface) [8] standard. Using this technique, parallelism can be achieved both on SMP (symmetric multiprocessing) platforms and over networks.

References

[1] J.-R. Abrial. *The B-Book: assigning programs to meanings.* Cambridge University Press, New York, USA, 1996.

[2] J.-R. Abrial, S. Hallerstede, F. Mehta, C. Métayer, and L. Voisin. Specification of basic tools and platform, 2005. Rodin Deliverable D10. EU-project RODIN (IST-511599), http://rodin.cs.ncl.ac.uk/deliverables.htm.

[3] Argonne National Laboratory. Mpich2. http://www-unix.mcs.anl.gov/mpi/mpich.

[4] B-Core (UK) Ltd. The b-toolkit. http://www.b-core.com/ONLINEDOC/BToolkit.html.

[5] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM Press.

[6] ClearSy. Atelier b. http://www.atelierb.societe.com.

[7] F. Degerlund, M. Waldén, and K. Sere. Implementation issues concerning the action systems formalism. In David S. Munro, Hong Shen, Quan Z. Sheng, Henry Detmold, Katrina E. Falkner, Cruz Izu, Paul D. Coddington, Bradley Alexander, and Si-Qing Zheng, editors, *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'07)*, pages 471–479. IEEE Computer Society Press, 2007.

[8] MPI Forum. Mpi documents. http://www.mpi-forum.org/docs.

[9] C. Métayer, J.-R. Abrial, and L. Voisin. Event-b language, 2005. Rodin Deliverable D7. EU-project RODIN (IST-511599), http://rodin.cs.ncl.ac.uk/deliverables.htm.

[10] M. Waldén and K. Sere. Reasoning about action systems using the B-method. *Formal Methods in System Design*, 13(1):5–35, May 1998.

[11] S. Wright. Automatic generation of c from event-b. In *Integration of Model-based Formal Methods and Tools 2009 (IM_FMT'2009), Workshop at IFM'2009, Informal Proceedings*, 2009. http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/.

Code Generation for Event-B with Intermediate Specification

Andy Edmunds and Michael Butler
University of Southampton

July, 2009

Abstract

The Event-B method and tools [1] provides a formal approach to modelling systems, and incorporates the notion of refinement. The work that we present bridges the abstraction gap between the lowest level of Event-B refinement and a working implementation, similar to B0 in classical-B [3]. We introduce an approach using an intermediate specification which allows a developer to specify processes with interleaving (non-atomic) operations. The specification of access to shared data is via a monitor abstraction with atomic procedure calls. Each process has a single non-atomic operation facilitating interleaving behaviour; the sub-clauses of a non-atomic operation consist of labelled atomic clauses. We use the clause labels as program counters, in a style similar to that of the $^+$CAL Algorithm Language [6]. The Labelled atomic clauses map to events guarded by a program counter (the label is used as the program counter). Using this approach we are able to introduce various non-atomic implementation level constructs to specify branching and looping behaviour. The intermediate specification is an abstraction, in that it hides implementation details of locking and blocking, and provides the developer with a clear view of atomicity.

We show how non-atomic operations are given Event-B semantics, and how each labelled clause maps to an atomic event. Automatic translation of an an intermediate specification will give rise to an Event-B model and source code of the chosen target language. Our work progresses to specifications that use object oriented techniques, with our approach called Object-oriented Concurrent-B (OCB) [4]. We describe how we model object-oriented implementations using the OCB notation. The techniques we use to model instantiation are similar to that of UML-B and U2B [7, 8]. For our initial investigations we choose Java 1.4 [2] as a target for working programs. However the techniques we employ could be applied to other object-oriented languages. Java's built-in synchronization mechanism is used to provide mutually exclusive access to data. The Java program will have atomicity that corresponds to the formal model and OCB clauses, and makes use of synchronized method calls. There is a close correspondence between the OCB model structure and the resulting Java classes.

An extension to OCB has been developed in which a number of objects can be updated within a single atomic clause; this is facilitated by

new Java SDK 5.0 [5] features such as non-blocking lock acquisition. The extension allows a process direct access to variables of a shared object using dot notation; and also allows multiple procedure calls in a single atomic clause. To facilitate Transactional-OCB we introduce implementation specific features to Event-B atomic actions (reminiscent of those of classical-B); such as a sequential operator, and atomic branching and looping.

References

[1] J.R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[2] M. Campione, K. Walrath, P. Chan, R. Lee, J. Kanerva, J. Gosling, B. Joy, G. Steele, G. Bracha, Technical Advisors - K. Arnold, T. Lindholm, and F. Yellin. The Java Language Specification - Second Edition, 2000.

[3] ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.

[4] A. Edmunds and M. Butler. Linking event-b and concurrent object-oriented programs. In *Refine 2008 - International Refinement Workshop*, May 2008.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition.* Addison-Wesley, 2004.

[6] L. Lamport. The $^{+}$cal algorithm language. In E. Najm, J.F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, page 23. Springer, 2006.

[7] C. Snook and M. Butler. *U2B - A tool for translating UML-B models into B*, volume UML-B Specification for Proven Embedded Systems Design. Springer, 2004.

[8] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 2006.

B2Visidia: A Tool to Generate a Java Code from Event-B Specifications of Distributed Algorithms

Mohamed Tounsi and Mohamed Mosbah
LaBRI Laboratory, Université Bordeaux 1
351 Cours de la Libération,
33405 Talence France
tounsi@labri.fr mosbah@labri.fr

Abstract

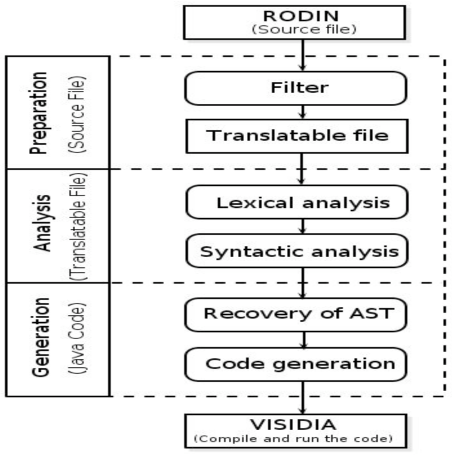
Generally, the design and the proof of distributed algorithms is a very difficult task due to the lack of knowledge of the global state and the non determinism in the execution of the processes. Formal methods like Event-B can guarantee that these algorithms run correctly and efficiently. However, the specification of distributed algorithms in the Event-B language remains a high level description of distributed algorithms. In this paper, we propose a general approach to translate an Event-B specification of a distributed algorithms into a Java implementation. Our approach is implemented with a tool called B2Visidia that allows the designer to directly manipulate programs of distributed algorithms specified in Event-B.

1 Introduction

In order to reduce complexity of distributed algorithms and construct proofs of correctness, it is necessary to use formal methods. Particularly, Event-B is a well suited method for the specification of distributed algorithms [7, 8, 5]. It is supported by a powerful tool called ”RODIN” [9]. It also maintains very well the refinement techniques that can transform an abstract, non-deterministic, specification into a concrete, deterministic system, in several stages. However, the mathematical aspect makes it very difficult to understand and to use it, especially for non experienced designers. Also, formal proof correctness of distributed algorithms are often long, hard and tedious. Simulation can help to understand and to discover many errors quickly and easily in algorithms before delving into a correct proof. In this context, we propose a new approach to implement and to visualize Event-B specifications of distributed algorithms. More precisely, we have developed a general method and a tool called B2Visidia to generate a Java implementation of a distributed algorithm specified in Event-B, which will run in the Visidia environment [4, 2, 3]. B2Visidia is based on the formal method Event-B and the Visidia environment for visualizing and experimenting distributed algorithms formally proved. We think that, the most important idea behind the construction of this tool is to allow the test and the debug of the specification of distributed algorithms and to provide a library of proved algorithms under Visidia which is a platform for implementing and visualizing distributed algorithms. Also, it allows to generate a Java implementation of a large class of distributed algorithms specified in Event-B. We use the high-level encoding of distributed algorithms by graph relabeling systems [6]. Moreover it offers an easy way for Event-B designer to generate a Java code.
In order to automatically translate Event-B in other languages, the most important work has been proposed by Stephen Wright [10]. It consists of generating a C code from Event-B with a multi-phased translation process. In this work, the author proposes a tool named B2C that has been developed as a plug-in to ”RODIN”.

2 Global architecture

Translating an Event-B model in a concrete language (such as Java language) is not possible in one shot. Since, Event-B specifies any system without taking into account its implementation. B2Visidia makes it possible to translate an Event-B specification (with a few added annotations) into a Java code for Visidia. As presented in the following figure, our approach includes three stages: the initial step consists in preparing the source file stored as an XML file in the Rodin platform. The goal is to generate a simple and translatable text file that holds the useful parts for the translation.



To this end we have chosen Tom [1] which is a language and a software environment very well-suited for programming various transformations on trees/terms and also it can be used to match and rewrite XML documents. Once the file has been re-written, we perform, in the second step a lexical and a syntactic analysis of the translatable file to build an Abstract Syntax Tree (AST) of the algorithm specification. Finally, we generate a code for Visidia. In this step, we use suitable rules to perform the conversion of AST nodes and generate the corresponding Java code.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom manual. Technical report, INRIA CNRS, 2008.

[2] M. Bauderon, S. Gruner, and M. Mosbah. A new tool for the simulation and visualization of distributed algorithms. *MFI'01*, 1:165–177, mai 2001. Toulouse, France.

[3] M. Bauderon, S. Gruner, Y. Mtivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. *Second International Workshop on Graph Transformation and Visual Modeling Techniques, ENTCS*, 50(3):229–239, juillet 2001. Crete, Greece.

[4] Michel Bauderon and Mohamed Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. *Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation)*, 72(3):13–24, 2003.

[5] Sentot Kromodimoeljo, Bill Pase, Mark Saaltink, Dan Craigen, and Irwin Meisels. The eves system. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, pages 349–373, London, UK, 1993. Springer-Verlag.

[6] I. Litovsky, Y. Métivier, and Éric Sopena. Different local controls for graph relabelling systems. *Mathematical System Theory*, 28:41–65, 1995.

[7] Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Laboratory for Computer Science, Cambridge, MA, May 25, 2002.

[8] Ib Sorensen and David Neilson. B: towards zero defect software. *The Kluwer International Series In Engineering And Computer Science*, pages 23–42, 2001.

[9] Laurent Voisin. Public versions of basic tools and platform. Public Document Project IST-511599, ETH Zurich, 29 October 2007.

[10] Steve Wright. Automatic generation of c from event-b. In *Workshop on Integration of Model-based Formal Methods and Tools*. http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/im_fmt2009_proceedings.html, February 2009.

On Event-B and Control Flow

A. Iliasov
Newcastle University

1 Overview

Event-B is a general-purpose specification language. However, it tackles some problems less successfully than others. One such class is the problems with rich control flow properties. Control flow is a not a natural part of an Event-B specification. The next event to be executed is selected non-deterministically among the currently enabled events of a machine. The information about event ordering has to be embedded into guards and event actions. This results in an entanglement of control flow and functional specification with an additional downside of extra model variables. This leads to an increased complexity and reduced readability of a model. One way to decouple control flow from functional specification is to define control flow information in a dedicated language, as an extension of a conventional Event-B model.

There are a number of reasons to consider an extension of Event-B with an event ordering mechanism:

- for some problems the information about event ordering is an essential part of requirements; it comes as a natural expectation to be able to adequately reproduce these in a model;
- explicit control flow may help to prove properties related to event ordering;
- sequential code generation requires some form of control flow information;
- since event ordering could restrict the non-determinism in event selection, model checking is likely to be more efficient for a composition of a machine with event ordering information;
- there is a potential for a machine editor presenting a visual machine layout based on control flow information;
- realizing such a mechanism could be an initial step towards bridging the gap between high-level workflow languages and Event-B.

The purpose of adding flow information to a machine is to express event ordering, on top of the information already contained in event guards. One convenient way to achieve this is to define such ordering information in the form of a *next function* - a function that for a given event returns the possible next events. Its general form is as follows:

$$NF : Event \nrightarrow \mathbb{P}(Event)$$

This simple construct is capable of expressing notions such as sequential composition, event choice and loop[1]. As an example, let us consider the following instance of a next function: $NF_1 = \{first \mapsto \{second\}, second \mapsto \{first,' stop\}\}$.

[1]A more general form $NF : \mathbb{P}(Event) \nrightarrow \mathbb{P}(\mathbb{P}(Event))$ is needed to support parallel composition. Due to space restrictions the discussion is limited to a simpler sequential case.

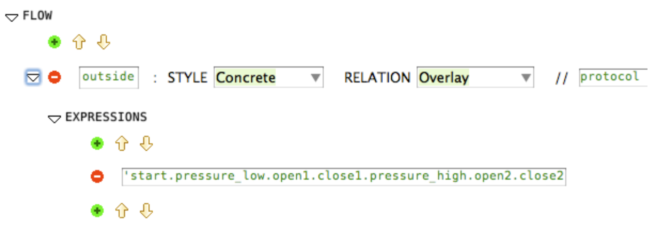

```
FLOW
outside : STYLE Concrete   RELATION Overlay   // protocol
  EXPRESSIONS
    'start.pressure_low.open1.close1.pressure_high.open2.close2
```

Figure 1: The Flow section in the RODIN Platform Event-B Editor.

The function mentions three events and only two of them are in the function domain. According to this definition, event $first$ is followed by event *second* (sequential composition) and event *second* is followed by $first$ or $'stop$ (the choice construct is embedded into the loop construct). A nicer syntactic alternative is $*(first.second).'stop$. Here, dot (.) denotes sequential composition and $*(...)$ stands for a possibly terminating loop. There are also choice $a|b$ and non-terminating loop $**(...)$ operators. It is easy to give precise semantics to this syntax sugar in the terms of properties of a corresponding next function.

Events starting with $'$ bear special meaning. $'start$ is a shortcut for event INITIALISATION, $'stop$ is an assumed termination event. Its guard is the negation of the disjunction of all event guards and its body is undefined. Finally, event $'skip$ is used to complete partial expressions (e.g., e is interpreted as $e.'skip$). An Event-B machine, not constrained with a flow expression, is understood to comply with the following next function: $'start.*(e_1|e_2|\ldots|e_k).'stop$. Informally, the function is interpreted as a potentially terminating loop made of the choice on all the machine events.

A flow of a machine may not be an arbitrary expression. It must agree with the definition of machine events and respect a number of consistency conditions. All of these are either computed from the definition of a flow and a machine specification or proved by discharging a number proof obligations, much like it is done for machine consistency and refinement. The exact set of proof obligations, however, depends on the role a flow plays in a specification. So far, we have identified two flags that define four varying flow roles. A flow may be *abstract* or *concrete*; a specification with a concrete flow may be implemented by a sequential program. A flow may be *overlaid* or *equivalent*; an equivalent flow does not constrain the event ordering prescribed by event guards.

There is a basic consistency proof obligation that shows that a next function expressed agrees with an Event-B specification on deadlocks and divergencies. Proving that a flow is concrete or equivalent requires further proof obligations. The case of a concrete overlaid flow is enough to *drive* a machine and is normally a prerequisite for generating an implementation in an imperative programming language.

A proof-of-concept implementation was realised in the form of RODIN Platform plugin. The plugin extends the machine editor to provide the means for entering flow expressions and also contributes new proof obligation needed to establish flow consistency, implementability and equivalence (Figure 1). A number of challenges were identified such as the need for additional hypothesis and scalability issues for some proof obligations.

A Rodin plugin for quantitative timed models*

Joris Rehm
joris.rehm@loria.fr - LORIA - Nancy Université

We propose to develop a Rodin[1] plug-in that experiments a systematic use of a refinement pattern. The goal of this pattern is to help in modeling of timed system in Event-B. By timed system we mean system with quantitative temporal constraints and properties.

The user (of the plug-in) will see and modify an Event-B machine augmented with a new operator S. This unary operator over an event name e gives the delay elapsed since the latest triggering of the event e. By using this operator in the invariant, the user will be able to write and prove temporal properties over the events. By using this operator in the guard of an event f, the user will be able to specify temporal constraints over the duration $S(e)$ (which becomes here the delay between the event e and f). For the possible constraints, we plan to consider lower time bounds ($l \leq S(e)$ in guard of f) and upper time bound ($S(e) \leq u$ in guard of f). The upper time bound is a bound within the event f must obligatory occurs, it is not just a possibility. In this text, the usages of the operator S is called the annotations, it is an extension of the syntaxe of the B models. The annotations can only appear in invariant or guards.

Our pattern is an Event-B model that encodes the behaviour of the operator S (we call this model the pattern model). The annotations given by the user define how to refine the pattern model and how to obtain the behaviour of S needed for a particular augmented model. The goal of our plug-in is to generate a normal Event-B model that is the studied (augmented) model where the annotations are replaced with the superposition of the refined pattern model.

To explain briefly what is the idea of the pattern, we show below an augmented model (on the left) and the generated result (on the right). This small model define a light that can be on ($lo = TRUE$) or off ($lo = FALSE$); the light can be switch on by a button (the event *on*) and goes automaticaly off (event *off*) after a delay between $c - d$ and $c + d$. You can see the temporal annotations in the elements named lb_off (Lower Bounds) and up_off (Upper Bounds). The S operator can also appear in the invariant, for example we can write $c + d < S(on) \Rightarrow lo = FALSE$.

EVENTS on $\widehat{=}$ **Begin** act1: $lo := TRUE$ **End**	**EVENTS** on $\widehat{=}$ **Begin** act1: $lo := TRUE$ act2: $s\_on := 0$ **End**

*This work was supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

[1]http://www.event-b.org

```
off ≙
  When
    grd1: $lo = TRUE$
    lb_off: $c - d \leq S(on)$
    ub_off: $S(on) \leq c + d$
  Then
    act1: $lo := FALSE$
  End
```

```
off ≙
  When
    grd1: $lo = TRUE$
    lb_off: $c - d \leq s\_on$
  Then
    act1: $lo := FALSE$
  End
tic ≙
  Any s
  Where
    grd1: $0 < s$
    ub_off: $lo = TRUE$
              $\Rightarrow s\_on + s \leq c + d$
  Then
    act1: $s\_on := s\_on + s$
  End
```

To generate the model the plug-in will have to: declare a new variable s_e for all events which appears in the operator S; reset this variable s_e to zero in the event e; replace all $S(e)$ by s_e for all e; generate a *tic* event that increments the s_e clocks (for all e) and add in the guard of *tic* the predicate : $GUARD(f) \Rightarrow s\_e + s \leq u$ for all upper bound $S(e) \leq u$ in a event f, with $GUARD(f)$ being the guard of the event e (without temporal annotations).

For the user interface, we plan to specialize the standard "edit" editor of Rodin. The user will be able to add his annotations and when the machine is saved, the plugin will generate a normal machine. The proof will be done over the proof obligations of this generated machine.

For more information about our pattern see [3], another example of pattern for time can be see in [2]. Our notion of (refinement) pattern is the same than the Action/Reaction pattern (see the chapter 3 of [1]). As related work, we can also cite the works about automatic refinement with tools like Bart[2]. There are some common elements between this tool and our work, the major difference is that Bart transforms a B machine to a B machine (more close to an implementation), where our augmented model is not a normal B machine.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2009.

[2] Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for event B development. In *B 2007: Formal Specification and Development in B*, volume 4355/2006, pages 140–154. Springer, January 17-19 2007.

[3] Joris Rehm. Pattern Based Integration of Time applied to the 2-Slots Simpson Algorithm. In *Integration of Model-based Formal Methods and Tools - IM_FMT'2009 - in IFM'2009*, Düsseldorf Allemagne, 02 2009.

[2]`http://www.tools.clearsy.com/index.php5?title=BART_Project`

Composition, Renaming and Generic Instantiation in Event-B Development

Renato Silva, Michael Butler

Dependable Systems and Software Engineering Group
Department of Electronics and Computer Science
University of Southampton

Abstract

Modelling large scale systems usually results in large and complex models that become hard to manage and cumbersome. A possible solution is to apply some techniques that ease those difficulties. Examples are the Composition and Generic Instantiation techniques. Composition is the process by which it is possible to combine different sub-systems into a larger system. Independent modelled sub-components can be used instead of creating a large complex model using one single component . The advantage of this approach is to deal with the complexity of the sub-components individually (usually through independent refinements) without thinking about the other sub-components. The proofs are also (in general) simpler since they just refer to the individual properties of the sub-component. The sub-components, after some possible sub-developments (refinements), can be reused by creating a larger component through composition.

We use Event-B as a formal method to model our systems, which is based on the classical B, created by J.R. Abrial. Event-B models contain variables whose values are assigned when events are enable (first order logic predicate guards) and actions are executed. Properties of the models are expressed by invariants and axioms. We propose a shared event composition, where synchronised events from different sub-components are merged. When all the guards from all the synchronised events are enabled, parameters are merged and the actions of each event are executed in parallel. Event-B has the same semantics structure and refinement definitions as Action Systems. There is a correspondence between parallel composition in CSP and event based parallel composition for Action System. Extending that same correspondence to event based parallel composition in Event-B, the failure-divergence definition in CSP can be applied to Event-B and as a consequence, the monotonicity property is shared. Which means that the sub-components can be refined independently since the composition is still be preserved.

The Generic Instantiation is another technique that deals with reusability of components: a template or pattern is used to create more specific instances. The instances inherit properties from the pattern and personalise it by renaming or replacing those properties. We propose the instantiation of Event-B machines and we use contexts as parameterisation of instantiated machines. Renaming/refactoring plugin is used for renaming and replacing the specific properties. If the proofs in the pattern are already discharged, they do not need to be addressed in the instantiated machines. On the other hand, as part of the parameterisation, the assumptions must be assured and satisfied by the instantiation. A possible solution is to instantiate the axioms (assumptions) into theorems. We show an example where the generic instantiation is applied.

A demonstration of the prototype of the Composition plug-in will be provided when applied to a simple example. Also a Renaming plug-in will be shown and described as a necessity when composing sub-systems into a larger system.

Expressing KAOS Goal Refinement Patterns with Event-B

Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau

LACL, Université Paris-Est
{abderrahman.matoussi,frederic.gervais,laleau}@univ-paris12.fr

Employing formal methods like B [1] for complex systems specification is steadily growing from year to year. They have shown their ability to produce such systems for large industrial problems. With formal methods, an initial mathematical model is refined in multiple steps, until the final refinement contains enough detail for an implementation. This initial model is derived from the user requirements (requirements analysis). As this formal development chain matures, the major remaining weakness in the development chain is the gap between textual or semi-formal requirements and the initial formal specification. In fact, the validation of this initial formal specification is very difficult due to the inability for customers to understand formal models, to link them with initial requirements. Consequently, the gap between the requirements phase and the formal specification phase gets larger and larger and the reconciliation seems more and more difficult.

Our objective is to bridge this gap using the goal-based requirements engineering method KAOS [4, 5] and the Event-B formal method [2] by including the requirements analysis phase in the software development associated with the formal methods. Contrary to other requirements methods such as i* [7], KAOS is promising in that it can be extended with an extra step of formality which can fill in the gap between requirements and the later phases of development. The choice of Event-B is due to its similarity and complementarity with KAOS. Firstly, Event-B is based on set mathematics with the ability to use standard first-order predicate logic facilitating the integration with the KAOS requirements model that is based on first-order temporal logic. Secondly, both Event-B and KAOS have the notion of refinement (constructive approach). Finally, KAOS and Event-B (Contrary to the classical B) have the ability to model both the system and its environment. For these various reasons, the proposed approach, which is very briefly presented in [6], aims to prove the KAOS requirements model and to establish formal links between this model and the Event-B specification of a system. Since goals play an important role in requirements engineering process and provide a bridge linking stakeholder requests to system specification [8], the proposed approach comes down to automatically derive Event-B specifications from KAOS goal model rather than from KAOS requirements model as a whole. Consequently, we show that it is possible to express KAOS goal models with formal method like Event-B by staying at the same abstraction level. However, it is not possible to verify that both models are equivalent. In fact, the Event-B expression of the KAOS goal model allows us to give it a precise semantics.

To achieve our objective, we formalize (with Event-B) two basic KAOS patterns that analysts use to generate a KAOS goal hierarchy: (i) *OR refinement* which simply specifies alternative means to satisfy a goal; (ii) *AND refinement* which means that the conjunction of the subgoals is a sufficient condition to achieve the parent goal. Moreover, we present the Event-B formalization of some KAOS refinement patterns such as *the milestone refinement pattern* [3] which consists in identifying milestone states that must be reached to achieve the target predicate. We think that these formal design patterns or proof-based design patterns will be very useful and explores the fact that the Event-B method provides a framework for developing generic models of systems.

The main contribution of our constructive approach, driven by goals, is that it establishes a bridge between the non-formal and the formal worlds as narrow and concise as possible. Moreover, this bridge balances the tradeoff between complexity of rigid formality (Event-B) and expressiveness of semi-formal approaches (KAOS). However, a number of future research steps are ongoing. Further work will consist in applying the approach on a number of case studies in order to support non-functional goals. This would address issues of conflict between these goals, which does not exist between functional goals. At tool level, we plan to develop a connector between KAOS toolset and the RODIN[1] open platform.

References

1. J.R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.
2. J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B'98*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, April 1998. Springer-Verlag.
3. R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *SIGSOFT '96*, pages 179–190, San Francisco, California, USA, October 1996. ACM SIGSOFT.
4. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE 2001*, pp. 249–263, Toronto, Canada, August 2001. IEEE Computer Society.
5. E. Letier. Reasoning About Agents in Goal-Oriented Requirements Engineering. Phd Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, Mai 2001.*ftp://ftp.info.ucl.ac.be/pub/thesis/letier.pdf*
6. A. Matoussi and F. Gervais and R. Laleau A First Attempt to Express KAOS Refinement Patterns with Event B. In *ABZ 2008*, pages 338, London, UK, September 2008. Springer.
7. E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pages 226-235, 1997. IEEE Computer Society Press.
8. W.N. Robinson and S. Pawlowski. Surfacing Root Requirements Interactions from Inquiry Cycle Requirements Documents. In *The Third IEEE International Conference on Requirements Engineering (ICRE'98)*, pages 82–89, Colorado Springs, CO, USA, 1998. IEEE Computer Society Press.

[1] http://rodin-b-sharp.sourceforge.net

Title: A tool for specifying and validating software responsibility

Abstract: The objective of ANR project LISE (Liability Issues in Software Engineering) is to precisely define software responsibility and to use this definition as a basis for liability agreement. Presently, the LISE project uses Event B notations as the basis to define the system responsibility model (interface components, electronic evidences, parties and their liabilities) and as a way to check software liability properties. At first we will present the current state of our responsibilities model. Next we give a draft for a tool, based on graphical representation and animation, that aims to help parties to elaborate liability contracts.

Language and Tool Support for Class and State Machine Refinement in UML-B

Mar Yah Said, Michael Butler, and Colin Snook

ECS, University of Southampton, Southampton, SO17 1BJ, UK
(mys05r,mjb,cfs)@ecs.soton.ac.uk

UML-B is a 'UML-like' graphical front end for Event-B. It adds support for object oriented modelling concepts while retaining the Event-B modelling concepts. In the continuity of the work on UML-B, we strengthen its refinement concepts. Development in Event-B is done through refinements of an abstract model. Since Event-B is reflected in UML-B, the abstraction-refinement concepts must also be catered for in UML-B. UML-B introduced the new concept of refinement, where model complexity is managed by introducing more detailed versions of a machine. We extend this refinement concept by introducing the notion of refined classes and refined state machines. Together with these notions, several refinement techniques in UML-B are defined. The UML-B drawing tool and Event-B translator are extended to support the refinement concepts. A case study of an auto teller machine (ATM) is presented to demonstrate the notion of refined classes and refined state machines.

The motivation for refined classes (and inherited attributes) come from performing refinement in Event-B. The notion of refined classes in UML-B reflect the refinement of variables in Event-B. A refined class is one that refines a more abstract class and an inherited attribute is one that inherits an attribute of the abstract class. The motivation for refined state machines (and refined states) come from combining the state machine hierarchy in UML-B with refinement in Event-B. The essential concept is that state machines are refined by elaborating an abstract state with nested sub-states. A refined state machine is one that refines a more abstract state machine and a refined state is one that refines a more abstract state.

This work also introduces five refinement techniques which are, adding new attributes and associations to a refined class, adding new classes in a refinement, elaborating state, elaborating transition and moving a class event to a refined class or a new class in a refinement.

A case study based on an auto teller machine (ATM) was undertaken to validate the extension of UML-B with regards to the notion of refined classes and refined state machines. The ATM case study also demonstrates the use of the above five refinement techniques. There are seven machine levels for the ATM UML-B development. These machines are linked by a refinement relationship. The state machine refinement in the second, third and fourth refinements introduced additional levels in the state machine nesting hierarchy. The approach of elaborating states with sub-states in refinement, as illustrated by the ATM case study, supports an incremental refinement approach. The hierarchical structure of nested state machines also supports modular reasoning by localising the invariants required for refinement proofs.

An EMF Framework for Event-B

Colin Snook, University of Southampton,
Fabian Fritz, Heinrich-Heine University,
Alexei Iliasov, University of Newcastle

The Eclipse Modelling Framework (EMF) provides supporting infrastructure for building tools and other applications based on a structured data model. The EMF contains generative tools and runtime support for the implementation of a model repository based on a meta-model of the domain. Also provided are a set of adapter classes that enable viewing and command-based editing of the model. Further support for other model manipulation tasks has been added to the EMF. For example, support for model transformation, comparison and merging is provided.

During its initial development, the Rodin platform did not utilise EMF technology because of fears about performance. This prevents integration with other EMF based modelling tools. To completely re-base the Rodin platform on the EMF retrospectively would require extensive re-work of the Rodin verification tools which are tightly integrated with Rodin's bespoke repository format. Our solution is to provide an EMF based 'front-end' to the Rodin platform and retain the current repository and tool set. To do this we have implemented an EMF based repository for Event-B models that overrides the default serialisation to persist models via the Rodin API. There were significant challenges to provide a flexible EMF implementation. For example, some tools need to work at the project level manipulating a collection of Event-B components (machines and contexts) whereas others are scoped within a single component.

The Event-B meta-model defines the structure of Event-B projects. The model is contained within the repository plugin, *org.eventb.emf.core.* It is structured into three packages for clarity. The core package contains a structure of abstract meta-classes so that models can be treated generically as far as possible. The core package also contains mechanisms to handle extensions provided by other plug-ins and a meta-class to model entire projects. There are two sub-packages, contained with the core package, for machine and context.

The extensibility mechanism caters for extensions (i.e. new elements and/or attributes) that have been defined for the Rodin database. The extension mechanism can also be used to store temporary volatile modelling data that will not be persisted. The extensions will only be persisted if valid Rodin identifiers are provided. Two mechanisms are provided, one for simple attributes (corresponding to Rodin attribute extensions) and one for extension elements (corresponding to Rodin element extensions).

The persistence plug-in *org.eventb.emf.persistence* overrides EMF's default XMI serialisation so that models are persisted in the Rodin Database. The serialisation uses the Rodin API so that it is decoupled from the actual serialisation of the Rodin database. Our EMF Persistence API provides methods to load and unload components, save changes, and register listeners to projects and components. An extension point is provided for offering synchronisers for new element kinds (extensions). Persistence takes a rewriting approach. That is, it is more efficient to clear the contents of a model component (machine or context) and regenerate it than to calculate what has been changed and only save the changes.

Tools that will use the EMF framework for Event-B include:

Text editor Based on the EMF framework, a state-of-the-art text editor has been created that provides an extensible set of features, such as (syntactical and semantical) highlighting, code completion, quick navigation and outline view.

Compare/Merge Editor In several situations conflicts between different versions of an Event-B model can occur. A compare and merge editor for Event-B models will help users to solve these conflicts. This editor will be based on the EMF Compare sub-project. It will compare the two conflicting versions and present the differences to the user. A compare and merge editor is essential for team-based development where it can be linked to a version control system such as SVN or CVS.

Structure Editor EMF provides support to generate structured (e.g. tree, list, table based) editors for models. An adapted version of these editors will allow users to edit machine and context elements within a structure using menu-guided selections.

Project Diagram Editor A diagrammatic editor will be produced that shows the structure of an Event-B project in terms of its machines and contexts with their refines, sees and extends relationships. The project diagram editor will allow machines and contexts to be created/deleted and their relationships changed. A feature to create a 'starting point' refinement of a machine, will be included.

UML-B will be re-implemented as an extension to the Event-B meta-model. The UML-B meta-classes will extend and add to the meta-classes of Event-B. This will provide greater integration between the EMF based Event-B editors and the UML-B diagrammatic editors.

Feature Composition Tool An editor for composing two machines based on feature selection has been developed by Southampton. The tool (which is already based on EMF) will be re-implemented to utilise the Event-B EMF framework.

Using CSP Refusal Specifications to Ensure Event-B Refinement Models

James Sharp
Department of Computing,
University of Surrey,
Guildford,
United Kingdom,
GU2 7XH
j.h.sharp@surrey.ac.uk

June 16, 2009

Abstract

We are interested in determining whether Event-B models can map to CSP models and whether the refinements of Event-B models also correspond to CSP refinement. Currently we have mapped the Machine Press controller to CSP and demonstrated that the different levels of refinement can be checked using FDR. Also by using a mapping from Linear Temporal Logic to CSP, we are also able to verify the system invariants. Whilst we have looked at a simple example we need to expand this by looking at more complex Event-B models, i.e., ones which include variables that are not always used within the guards of events. Event-B refinement often requires the strengthening of guards as a result of including new invariants. Thus, another area of interest is using CSP counter examples to give some indication of which guards need to be strengthened in the corresponding Event-B model. In order to support this work automatically generating the CSP from the Event-B would be beneficial as would developing an accompanying Eclipse plug-in.

Background

Determining the correct guards in an Event-B refinement to ensure that a model preserves its invariants is not always straight forward. Stepping through a proof assists in the discovery of missing guards. Even though there many proof obligations are automatically discharged, there may be outstanding proofs which maybe non-trivial.

CSP models from Event-B models

We propose to create a mapping of abstract and refinement layers in an Event-B system to CSP, and use CSP tools to verify the Event-B models. We have already started to explore the mappings between Event-B models and CSP models using the Event-B mechanical press example. We are able to show that the refinements in Event-B can be mapped to CSP and shown to be refinements in the Stable Failures model. We are also able to convert the system invariants in the Event-B models using a mapping from LTL to CSP specifications. Then we can then discharge these CSP specifications as Refusal refinements of the CSP models in FDR. Failure to discharge the specifications highlights events which require additional guards in the Event-B model.

An Event-B to CSP Eclipse plug-in

Whilst we have been converting Event-B to CSP so that we can use alternative tools, this process has been repetitive and prone to mistakes. We believe that a tool to provide automated CSP scripts generated from Event-B systems would be beneficial. To this end the addition of an Eclipse plug-in to the RODIN platform would be a natural progression of an automated CSP generator. The automated tool should be able to perform:

- The conversion from Event-B to CSP (and if required B)
- Automatically generate, from the Event-B invariants, the Refusal Specifications needed for detecting missing guards.
- Return the event which violates a specific invariant, and suggest any additional guards that may be required.