Proceedings of the 5th Rodin User and Developer Workshop, 2014

June 2-3 2014, Toulouse

Michael Butler, Stefan Hallerstede, (Eds.)

UNIVERSITY OF
Southampton

AARHUS UNIVERSITY

Proceedings of the 5th Rodin User and Developer Workshop, 2014

June 2-3 2014, Toulouse

Michael Butler, Stefan Hallerstede, (Eds.),

Abstract:

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 have already been developed including ones that support animation, model checking and UML-B. While much of the development and use of Rodin takes place within EU FP7 Projects (RODIN, DEPLOY, ADVANCE), there is a growing group of users and plug-in developers outside these projects.

The purpose of the 5th Rodin User and Developer Workshop was 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.

The two day programme consisted of a half day of tutorial on theory development for Event-B followed by a day and a half of presentations on theory, tool development and tool usage. This volume contains the abstracts of the tutorials and presentations at the Rodin workshop on June 10 – 11, 2013.

The workshop was held at the ENSEEIHT Engineering School in Tolouse. The Rodin Workshop was supported by the University of Southampton, Aarhus University and the FP7 ADVANCE Project www.advance-ict.eu,

Organisers:

Michael Butler, University of Southampton
Stefan Hallerstede, Aarhus University
Thierry Lecomte, ClearSy
Michael Leuschel, University of Düsseldorf
Alexander Romanovsky, Newcastle University
Laurent Voisin, Systerel
Marina Walden, Åbo Akademi University

List of Presentations

Practical Theory Extension Tutorial session 1
Asieh Salehi, Jean-Raymond Abrial, Michael Butler

Modeling a Safe Interlocking Using the Event-B Theory Plug-in
Thang Khuu, Laurent Voisin, Fernando Mejia

Unlocking the Mysteries of a Formal Model of an Interlocking System
Michael Leuschel, Jens Bendisposto and Dominik Hansen

Run-time Management of Many-core Systems using Rodin
Asieh Salehi, Colin Snook, Michael Butler

An Experiment in Modeling Satellite Flight Formation in Event-B
Inna Pereverzeva, Anton Tarasyuk, Elena Troubitsyna

Developing and Proving a Complicated System Model with Rodin
A. V. Khoroshilov, I. V. Shchepetkov

Formalisation of Self-Organizing Multi-Agent Systems with Event-B and Design Patterns
Zeineb Graja, Frederic Migeon, Christine Maurel, Marie-Pierre Gleizes, Ahmed Hadj Kacem

Generating Tests for COTS Components with Event-B and STPA
Toby Wilkinson, Michael Butler, John Colley, Colin Snook

Towards Verified Implementation of Event-B Models in Dafny
Mohammadsadegh Dalvandi, Michael Butler

A Practical Approach for Validation with Rodin Theories
Daniel Plagge, Michael Leuschel

Toolbox for penetration testing based on Rodin and ProB
Aymerick Savary, Marc Frappier, Jean-Louis Lanet

iUML-B Statemachines
Colin Snook

EB2RC: A Rodin plug-in for visualising Event-B models and code generation
Zheng Cheng, Dominique Mery, Rosemary Monahan

Composition Operators for Event-B. CO4EB Rodin plugin
Idir Ait-Sadoune, Yamine Ait- Ameur

CODA Update: New Features for 2014
Neil Evans, Helen Marshall, James Sharp, Michael Butler, John Colley, Colin Snook

Rodin Multi-Simulation Plug-in
Vitaly Savicks, Michael Butler, John Colley, Jens Bendisposto

Code Generation – Tool Developments
Andy Edmunds

Smart Grids: Multi-Simulation, An Application
Brett Bicknell, Karim Kanso, Jose Reis

Formal Methods, Requirements and Software Engineering
Ken Robinson

Applying and Extending the Event Refinement Structure Approach to Workflow Modelling
Dana Dghaym, Michael Butler, and Asieh Salehi Fathabadi

Incorporating "operation calls" in Event-B and Rodin (by means of Guarded Events)
Jean-Raymond Abrial

Program Development in Event-B with Proof Outlines
Stefan Hallerstede

Responsiveness and Event-B
James Sharp, John Colley, Helen Marshall, Neil Evans, Michael Butler, Colin Snook

Towards Patterns for Statemachine Modelling under Timing Constraints
Gintautas Sulskus, Michael Poppleton, Abdolbaghi Rezazadeh

From Untimed Specification to Cycle-Accurate Implementation - Cyber-Physical System Model Refinement with Event-B
John Colley, Michael Butler

Event-B for Safety Analysis of Critical Systems
Matthias Gudemann and Marielle Petit-Doche

Modelling Of Systems Of Systems - An Event-B Perspective Of a VDM Project
Stefan Hallerstede , Klaus Kristensen, Peter Gorm Larsen

Modeling a Safe Interlocking Using the Event-B Theory Plug-in

Minh-Thang Khuu and Laurent Voisin*
Systerel, Aix-en-Provence, France
{minh-thang.khuu,laurent.voisin}@systerel.fr

Luis-Fernando Mejia
Alstom Transport Information Solutions, Saint-Ouen, France
luis-fernando.mejia@transport.alstom.com

June 2-3, 2014

Abstract

Interlocking (IXL) is a railway signaling sub-system. Its principal role is controlling the movement of points, the change of signal aspect and setting up traffic directions on a railway network. These controls are performed via commands on signaling system devices. The main issue of IXL concerns the safety of commands, or more precisely, properties preventing risks of train collision and derailment. This paper presents an Event-B model of a safe IXL in which IXL commands are filtered to ensure safety properties.

In the model, railway terms are expressed with the *Theory plug-in* datatypes and operators. This approach has a two-fold advantage. Firstly, the model is lighter. In fact, complex mathematical expressions are held separately in *Theory plug-in* operators. Moreover, proof rules defined in *Theories* reduce the effort of proving activities. Secondly, domain specific terms are defined and reusable.

The model is illustrated by an animation using the ProB plug-in.

References

[1] I. Maamria and M. Butler. Practical Theory Extension in Event-B. Theories of Programming and Formal Methods. 2013.

[2] I. Maamria and M. Butler. The Theory plug-in and its applications. Rodin User and Developer Workshop. Fontainebleau, 2012

[3] J.R Abrial. Modeling in Event B: System and Software Engineering. Cambridge University Press. 2010.

[4] The Rodin platform is available from `http://www.event-b.org`.

*This work was partly funded by the FP7 ADVANCE Project (ICT-287563), see http://www.advance-ict.eu.

[5] The Theory plug-in is available from http://rodin-b-sharp.sf.net/updates.

[6] ProB is available from http://www.stups.uni-duesseldorf.de/prob_updates.

Unlocking the Mysteries of a Formal Model of an Interlocking System

Michael Leuschel, Jens Bendisposto and Dominik Hansen

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

Abstract. In his book on Event-B, Abrial presents a model of a railway interlocking system. While this model is an academic model intended for teaching Event-B, it has a lot of features in common with real-life interlockings and formal models thereof. In this paper we show how we used animation, model checking and constraint-based checking to uncover various interesting aspects about this specifications. In particular, we uncover an error which allows points to be "magically" connected to unrelated track segments. We also delve upon the exhaustive model checking of the relatively small topology from the book. This turned out to be surprisingly difficult, but allows us to present various methods and techniques to get a handle on the state explosion problem. Finally, we relate this work to validation of real, industrial interlockings.

1 Summary

The most prominent industrial uses of the B-method have been within the railway domain [2, 3]. It is thus not surprising that Abrial's book [1] on the Event-B method contains a railway system case study. More precisely, chapter 17 contains a formal development of a railway interlocking system. The role of an interlocking is to safely operate signals and points within an area of the train network (usually a station). This means that the interlocking controller has to ensure that trains do not collide, that points are not moved while a train is driving over them, and that trains reach their desired destination.

In this talk we use this model, which is hopefully already familiar to some readers, to highlight a variety of related points:

- we show the usefulness of animation to better understand the model. We also describe how one can use Rodin's refinement mechanism to animate a generic model for various concrete topologies.
- we show that constraint solving has uncovered a flaw in the fourth refinement of the fully proven model,

** Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE).

- we show that, even for the simple topology in the book, exhaustive model checking is surprisingly difficult. Indeed, for the simple topology from page 524 [1] with 5 signals, 5 points, one crossing and 14 tracks segments, the first refinement of the model has has over 61 million distinct states and over 445 million transitions (aka events). Through the use of a manual partial order reduction combined with either the use of parallelisation (within ProB [5] or using our translation to TLC from [4]) we have managed to cut verification time from more than four days down to minutes.
- we provide an overview of various attempts to scale up validation of the model to larger topologies, and more realistic settings with hundreds of signals and points.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. D. Dollé, D. Essamé, and J. Falampin. B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.
3. D. Essamé and D. Dollé. B in large-scale projects: The Canarsie line CBTC experience. In J. Julliand and O. Kouchnarenko, editors, *Proceedings B'2007*, LNCS 4355, pages 252–254, Besancon, France, 2007. Springer-Verlag.
4. D. Hansen and M. Leuschel. Translating B to TLA+ for validation with TLC. Technical Report STUPS/2013, Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, 2013.
5. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

Run-time Management of Many-core Systems using Rodin

Asieh Salehi, Colin Snook, and Michael Butler

University of Southampton
{asf08r, cfs, mjb}@ecs.soton.ac.uk

As electronic fabrication techniques approach the limit of atomic dimension, increases in performance can no longer be obtained from a single core with relative ease. Even at current scales, wear-out due to physical phenomena such as electromigration has become a significant factor. Interest in recent years, therefore, has increasingly focused on many core devices. Managing the use of a large collection of cores to achieve a given computing task with adequate performance in an energy efficient manner while minimising wear-out is a challenging problem, which is being tackled by the PRiME project [1] . A *Run-Time Management* (RTM) system that is aware of application requirements and able to save energy by sacrificing performance when it will have negligible impact on user experience is required. Furthermore we require such a system for disparate operating systems and hardware platforms.

Our approach is to developed formal models using Rodin and plug-ins so that we obtain a precise specification from which we can generate variants and subsequently code for different platforms. Here we summarise our initial exploration of the problem by describing the models and modelling techniques we used to specify a temperature aware RTM for *Dynamic Voltage and Frequency Scaling* (DVFS) of a media display application. The RTM learns from the application when it can scale back voltage and frequency to save energy without missing too many frame deadlines. We also model thread scheduling and the resultant heating effects in the cores.

We use the *Event Refinement Structure* (ERS) approach [2, 3] to visualise and build the abstract level of the DVFS control as an Event-B model. ERS augments Event-B methodology with a diagrammatic notation that is capable of explicit representation of control flows. Providing such diagrams aids understanding and analysing the control flow requirements without getting involved with complexity of the mathematical formal language notation.

We use *Model Decomposition* [4, 5] to divide the DVFS control model into two sub-models: Controller and Environment. The controller sub-model consists of variables/events describing the SW layer properties whereas the environment sub-model consists of variables/events describing the properties of the user and the HW layer.

We use *iUML-B Statemachines* [6, 7] to model the thread scheduling process of the operating system under the influence of the RTM. State-machines provide excellent visualisation of mode-oriented problems and are animated for validation in synchronisation with BMotionStudio [8] visualisations of other parts of the model.

We developed a continuous model of the thermal properties of a core depending on voltage and frequency using the Modellica [9] language. The continuous model is simulated in conjunction with ProB simulation of the Event-B RTM. This is achieved via tools for mixed-simulation [10] which are under development in the Advance project [11] .

Executable code was generated using the *Code Generation* plug-in [12]. The code generation feature provides support for the generation of code from refined Event-B models. To this end a multi-tasking approach has been added to the Event-B methodology. Tasks are modelled by an extension to Event-B, called *tasking machines* which are an extension of the existing Event-B machine component. The code generation plug-in provides the ability to translate to C and Java in addition to Ada source code. We adapted the code generation plug-in and used it to generate a Java implementation of the DVFS RTM system.

References

1. PRiME: Power-efficient, Reliable, Many-core Embedded systems. `http://www.prime-project.org`
2. Butler, M.: Decomposition Structures for Event-B. In Leuschel, M., Wehrheim, H., eds.: Integrated Formal Methods. Volume 5423 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 20–38
3. Fathabadi, A.S., Butler, M., Rezazadeh, A.: A Systematic Approach to Atomicity Decomposition in Event-B. In: SEFM. (2012) 78–93
4. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for event-B. Softw., Pract. Exper. **41**(2) (2011) 199–208
5. Hoang, T.S., Iliasov, A., Silva, R., Wei, W.: A Survey on Event-B Decomposition. ECEASST **46** (2011)
6. Snook, C.: Modelling Control Process and Control Mode with Synchronising Orthogonal Statemachines. In: B2011, Limerick. (2011)
7. Savicks, V., Snook, C.: A Framework for Diagrammatic Modelling Extensions in Rodin. In: Rodin Workshop 2012, Fontainbleau. (2012)
8. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B Models with B-Motion Studio. In: FMICS'2009. Lecture Notes in Computer Science, Verlag (2009) 202–204
9. Fritzson, P., Engelson, V.: ModelicaA unified object-oriented language for system modeling and simulation. In: ECOOP98Object-Oriented Programming. Springer (1998) 67–90
10. Savicks, V., Butler, M., Bendisposto, J., Colley, J.: Co-simulation of Event-B and Continuous Models in Rodin. In: Rodin Workshop 2013, Turku. (2012)
11. ADVANCE: Advanced Design and Verification Environment for Cyber-physical System Engineering . `http://www.advance-ict.eu/`
12. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES. (2011)

An Experiment in Modeling Satellite Flight Formation in Event-B

Inna Pereverzeva, Anton Tarasyuk, Elena Troubitsyna
Åbo Akademi University, Finland

European Space Agency is launching a new technology – an autonomous satellite formation flying. It offers great benefits in acquisition of valuable scientific data. However, the autonomous aspect also significantly complicates the development and verification process. Moreover, since the representative environment is impossible to reproduce on ground, testing of the system before deployment is unfeasible. This puts a significant stress on the development and verification technologies.

We have undertaken a development of the autonomous satellite formation flying by refinement in Event-B. The main goal of our development has been to formally underpin the interplay between autonomous mode transitions, fault tolerance and unreliable communication. These aspects are all equally important for the correct system functioning. However, the requirements related to each of them, have been scattered and vaguely related to each other. Our development, has allowed us to establish a proper interrelation between them and rigorously define behavior of system in all possible combinations of conditions.

In our development, we have started from an abstract representation of mode transition scenario. We have also reserved a possibility of failure and modeled reaction on this. Our subsequent refinement steps have introduced the detailed system architecture and explicitly defined communication between components.

Modelling and verification of autonomous behavior was a particularly interesting task. The absence of centralized control requires sophisticated behavioural patterns that allow the satellites independently of each other perform the required actions in a coordinated manner. Our modeling has ensured that even when the communication link between the satellites is broken, the satellites preserve safety of flight formation and can regain coordination and control over the formation flying.

As a result of our development, we created a detailed model of the distributed architecture of the formation flying, explicitly defined the inter-satellite communication and verified correctness of system behavior in the autonomous and controlled stages.

The development was carried in the Rodin platform. We have extensively relied on Pro-B to validate complex mode transitions and on modularization extension to perform decomposition.

Developing and Proving a Complicated System Model with Rodin

A. V. Khoroshilov, I. V. Shchepetkov
Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
{khoroshilov,shchepetkov}@ispras.ru

We used the Rodin toolset for modeling a system with a large number of dependences between its objects, and also with many restrictions imposed on them. Although a resulting model size of this system is not very large (approximately 1700 lines, see Table 1 for details), we encountered with some challenges at both modeling and proving stages. We would like to present these challenges and to discuss possible ways of their solution.

At the modeling stage we used several Rodin plugins, namely the Camille text editor, animator and model checker ProB, Atelier B Provers, and SMT Solvers. Camille provides a user friendly environment for model development. However with increasing a model size Camille started to consume gigabytes of RAM and a computer stalled. ProB helped us to perform quick checks of a model consistency until the model size reached ~400 lines (~20 state variables). So we left with Atelier B Provers and SMT Solvers that we used to prove correctness of our model and these plugins work well except for some obstacles discussed below.

	Number	Lines of code
Contexts	1	57
Sets	9	10
Constants	22	7
Axioms	18	39
Machines	1	1669
Variables	37	1
Sets	7	1
Functions	30	1
Invariants	100	288
Type invariants	37	45
State invariants	63	243
Events	35	1375
The largest event	-	241
The smallest event	-	8
An average event	-	40
Proof obligations	1226	-

Table 1. Model's statistic.

There are many complicated logical expressions duplicated in various parts of the model. It would be natural to define them once and to use everywhere else by reference. Event-B allows to define a constant of a functional type with a lambda expression. But such a functional constant can not be used in this situation as far as Event-B does not support $(\lambda p \cdot P \mid E)$ lambda expressions where E is a predicate (our massive and complicated logical expression) and lambda(p)=E(p) (the output of this lambda expression is a BOOL value).

At the proving stage we found that Rodin adds a large number of axioms to the proving perspective. In most cases it greatly complicates both automatic and manual proofs. We suggest to add an ability to choose a way for adding axioms: manually by the user, automatically add a minimal set of axioms, and the approach used currently. For that approach we would like to discuss ways to sample required axioms more intelligently.

There are proof obligations in our model that require up to 2 days for their proving. Considering the total number of proof obligations (~1200), it makes this task challenging. Unfortunately Rodin and its plugins are not friendly for team verification. A file with proofs for our model occupies more than 200 megabytes that makes difficult to use traditional collaboration tools such as version control systems. It would be interesting to experiment with splitting of files with proofs into several small files – one event per a file, or even one proof obligation per a file.

Despite all these issues Rodin helped us to reveal a number of inaccuracies in the initial system description and to prove correctness of quite a complicated model for this system. We hope that discussion of our experience with Rodin community helps to make the toolset more usable for such kind of tasks out of box.

Formalisation of Self-Organizing Multi-Agent Systems with Event-B and Design Patterns (Tool usage)

Zeineb Graja[1 2], Frédéric Migeon[2], Christine Maurel[2], Marie-Pierre Gleizes[2], and Ahmed Hadj Kacem[1]

[1] ReDCAD laboratory, University of Sfax, Tunisia
zeineb.graja@redcad.org, ahmed.hadjkacem@fsegs.rnu.tn
[2] IRIT, Paul Sabatier University, Toulouse, France
{zeineb.graja,frederic.migeon,christine.maurel,marie-pierre.gleizes}@irit.fr

With the growing complexity of today's applications, Self-Organizing Multi-Agent Systems (SOMAS) are becoming widely used as a paradigm for software design. A SOMAS is defined as a set of autonomous entities called agents, having a local knowledge about their environment and interacting together in order to achieve a given task. The global behavior of the overall system emerges from the interactions between the entities and their interaction with the environment. According to the framework proposed by Serugendo in [1], three types of properties need to be questioned when developping a SOMAS: invariants, robustness and resilience. Invariants are properties that must hold at any time during the system execution. Robustness focuses on the ability of the system to converge to its goal (convergence) and to maintain it (stability) in the absence of perturbations. While resilience evaluates the ability of the self-organization mechanism to adapt the system to perturbations and changes. In order to verify these properties, SOMAS designers usually make use of simulation as well as stochastic model checking. In this presentation, we focus on formal assurances about robustness and resilience. For the moment, we suppose that these properties can be observed at the macro level after simulation or runtime execution. Our goal is then to formalise the SOMAS by means of the Event-B language in order to prove them by theorem provers under the Rodin tool.

Our modelling framework is guided by the use of patterns which enable us to take advantage from reuse of both the refinement and proofs which are defined in the patterns. More precisely, we define three patterns described as follows.
- The Agent Pattern (AgP): It describes a stepwise refinement strategy allowing to design a correct local behavior of the agents which guarantee the deadlock freeness in each step of their execution cycle. This refinement strategy begins with a very abstract model depicting a set of agents executing according to a cycle of three steps: perceive, decide and act. The refinement steps allows to model gradually the actions, then the decisions and finally the necessary operations for updating the agents perceptions.
- The Global Behavior Pattern (GBP): This proof pattern guides the designer in order to prove the convergence of the system. GBP represents an interpretation in Event-B for the following temporal formula: $\Diamond \Box$ *taskAchieved*, where

taskAchieved describes the state where the system converges to its goal. This interpretation is given based on the framework proposed in [2].
- The Self-Organization Pattern (SOP): This proof pattern guides the designer in order to prove the ability of the self-organization mechanism to adapt the system when perturbations occur. It represents an interpretation of the following temporal formula: $\Box(perturb \Rightarrow \Diamond SuccessSO)$, where *perturb* describes the system state after a perturbation or a change and $SuccessSO$ describes a state where the system has succeeded to recover from the perturbation. The pattern SOP is also defined based on the framework proposed in [2].

The proposed patterns were applied for the foraging ants case study by means of the *Pattern* plug-in described in [3]. This plug-in allows to instantiate the generic models without an explicit enumeration of the concrete sets' elements. The case study describes the behavior of a foraging ants colony. The properties to be proved are the following:
-For robustness: The property RB_Ants given below indicates that the ants are able to bring all the food to the nest.

$$RB\_Ants \mathrel{\widehat{=}} \Diamond(\Box(QuantityFood(Nest) = TotalFood(InitDistFood) \wedge \forall\, loc.loc \in Locations \backslash \{Nest\} \Rightarrow QuantityFood(loc) = 0)).$$

-For resilience: We define two properties for the resilience. The first one ($SO1\_Ants$) indicates that when a source of food is detected, the ants are able to focus on its exploitation.

$$SO1\_Ants \mathrel{\widehat{=}} \Box(\forall\, loc.loc \in Locations \backslash \{Nest\} \wedge Detected(loc) \Rightarrow \Diamond(QuantityFood(loc) = 0)).$$

The second ($SO2\_Ants$) indicates that the ants continue the environment exploration and detect new food when the source of food they are exploiting disappears.

$$SO2\_Ants \mathrel{\widehat{=}} \Box((\exists l.l \in Locations \wedge QuantityFood(l) > 0) \wedge (\forall\, loc1.loc1 \in Locations \backslash \{Nest\} \wedge EntirelyExploited(loc1)) \Rightarrow \Diamond(\exists loc2.\ loc2 \neq loc1 \wedge QuantityFood(loc2) \neq 0 \wedge Detected(loc2)).$$

Our experience with the use of these patterns shows an interesting help for novice designers especially about the manner to do proofs with Event-B. We plan now to reduce the effort which remains for the designer to instantiate the patterns and to do the proofs.

References

1. G. Di Marzo Serugendo, Robustness and dependability of self-organizing systems - a safety engineering perspective, in *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 254–268.
2. T. S. Hoang and J.-R. Abrial, Reasoning about liveness properties in Event-B, in *ICFEM*, 2011, pp. 456–471.
3. T.S. Hoang, A. Frst, J.-R. Abrial, Event-B patterns and their tool support, in *Hung, D.V., Krishnan, P. (eds.) SEFM, pp. 210219. IEEE Computer Society*, USA (2009)

Generating Tests for COTS Components with Event-B and STPA

Toby Wilkinson, Michael Butler, John Colley, and Colin Snook

University of Southampton, UK
{stw08r, mjb, J.L.Colley, cfs}@ecs.soton.ac.uk

Abstract. We describe ongoing work combining Event-B with System Theoretic Process Analysis (STPA), a hazard analysis technique developed by Leveson at MIT, to the generation of test suites for COTS components.

Increasingly in industry bespoke systems are incorporating large numbers of third party, off the shelf, components. The reason is simple, with the increasing complexity and sophistication of modern systems, it is not economically possible to re-implement bespoke versions of these components. However, such third party components may come with little or no formal guarantee of their correctness, and indeed, sometimes the only indicators of quality might be anecdotal evidence from other users. In many applications this is not acceptable.

The solution is to develop an independent test suite for a third party component, that can be used to verify the suitability of the component, in the specific context into which it is to be deployed as part of a larger system. Such a test suite does not aim to test all the functionality of such a component, but only that functionality that is required in the context into which the component will be deployed. It is the minimum standard to which the component must be measured. In some safety-critical contexts this may not be sufficient, and it may be required that it be shown that the component has no additional functionality beyond that covered by the test suite, or that in the given deployment context, that any additional functionality remains disabled.

The technique we have chosen to employ, is to build from the requirements of the target system, a model of the third party component in the specific context of the target system. Then we apply System Theoretic Process Analysis (STPA) [2] to discover the different hazards, and their possible causes, that could result from the interaction of the component and the target context. The resulting closed-system model of the component, target context, and the safety constraints that the hazard analysis identified, is formalised in Event-B [3]. The construction of this model is typically an iterative process, with the formal modelling in Event-B feeding back greater understanding of the components interaction with the target context through invariant violations identified by the ProB model checker, and STPA identifying new hazards that must be mitigated or eliminated.

Through careful use of refinement, an initially highly abstract model of the component in the context of the system can be refined to a concrete model that

accurately describes the interface between the component and the target system, the assumptions the component can make about its deployment context, and the guarantees that are required of the component in that context. Decomposition of the resulting Event-B machine along the interface between the component and its context, yields two Event-B machines, one that represents an abstract model of what we require of the component, and another that embodies all that the component may assume about its context.

A case study derived from a synthetic model of the FADEC (Full Authority Digital Engine Control) system of a hypothetical helicopter has been developed, and an Event-B model constructed for a subset of the functionality, an Anti-Ice Bleed Valve (AIBV). STPA hazard analysis has been applied to the AIBV, and safety constraints discovered and modelled in Event-B. The resulting closed-system model has been decomposed into a model that embodies the actual AIBV valve, actuators, sensors, and the rest of the environment, and a model that describes the functionality required of the AIBV controller for the safe operation of the AIBV.

The next step in our work is to use the formal model of the AIBV controller to generate test cases that could be used to verify that any potential implementation of the AIBV controller was safe. Indeed, for the purposes of this work we have used the code generation abilities of the Rodin toolset to generate an implementation of the AIBV controller from the Event-B model. We intend to use this implementation to assess the effectiveness of the test cases that we generate from the same model.

Time allowing, we hope to expand the model to include a larger subset of the FADEC functionality, and explore the scalability of our approach, and also, again time allowing, we hope to explore how our models can be validated using the co-simulation techniques developed in the ADVANCE project [1].

References

1. ADVANCE: An FP7 project. Website `http://www.advance-ict.eu`.
2. Leveson, N. G.: Engineering a Safer World. MIT Press. Free download available from `http://mitpress.mit.edu/books/engineering-safer-world` (2012)
3. Rodin: The Event-B toolset. Available from `http://www.event-b.org`.

Towards Verified Implementation of Event-B Models in Dafny

Mohammadsadegh Dalvandi, Michael Butler
University of Southampton

{md5g11,mjb}@ecs.soton.ac.uk

Ideally the development process of software systems with Event-B and Rodin platform should lead to implementation and executables. The Event-B language and Rodin in their basic form do not have any facility to support this. To bridge this gap, some research has been carried out and several plugins have been developed to provide facilities for the Rodin platform to make code generation possible. While the specification and modelling phase of the system can be proved by automatic and interactive provers in Event-B and Rodin, the generated code by most of the existing code generators is not verified. One way to tackle this issue is to verify the generated code with a static program verifier. A static program verifier proves the correctness of a code with regards to a well-defined formal specification. Using this approach to generate verifiable code from Event-B models requires the generation of not only the code itself but also verification assertions from the Event-B.

The focus of this research is on linking Event-B models and their implementation in Dafny [1]. Dafny is a programming language and program verifier based on the Z3 SMT-solver. Dafny has proved to be a powerful program verifier by verifying a number of challenging problems [2]. By providing formal specifications for a Dafny program, the verifier proves the correctness (or incorrectness) of the code with regards to its specifications. The specifications includes pre- and post-conditions, loop invariants, and variable framing. The language also supports specification-only updateable ghost variables and functions which can be used recursively. Ghost entities (ghost variables, functions and etc.) are only used for verification purposes and do not form part of the compiled code.

To identify the differences and similarities between Event-B models and Dafny specifications and programs, implementation of a map abstract data type was taken from [2] for a case study and modelled in Event-B. The Map ADT is implemented in Dafny by a linked-list and specified by two sequences: one for storing keys and the other for storing associated values. In Event-B, the map is modelled by an abstract model following by two successive refinements. In the abstract level the map is modelled simply by the use of a partial function where keys are in domain and values are in range. Sequences are introduced in first refinement and the linked-list is added in the second level.

To decrease the distance between Event-B and Dafny syntax, the standard Event-B may be extended by Theory Plug-in [3] and new data-types and operators may be defined. For modelling maps in Event-B, several new operators, inference rules and rewrite rules have been added to the existing theory of sequences. Without using the theory plugin there will be a huge syntactic gap between Event-B and Dafny.

Although Dafny does not have any special object invariant construct, this can be easily done by defining a validity function and using it as both pre- and post-conditions of every method. This can be seen equivalent to Event-B invariants. Therefore all invariants of the Event-B model can be placed within the body of the validity function and vice versa. The precondition of methods in the map implementations is only the validity function. Apart from the validity function, which must be placed in the post-condition of all methods, other post-conditions are needed to specify the desirable and exact behaviour of a method. This is essential because of Dafnys modular verification approach. The Dafny verifier only looks at the specification (pre- and post-conditions) of the other methods to understand their behaviour when it is verifying a method. Modelling the map data structure in Event-B shows that the required post-conditions can be inferred from guards and actions of each event. One or more events in an Event-B model may

be equivalent to a method in a Dafny program. Related events can be translated to a single method in Dafny and each event will translate to a conditional statement within that method. Guards of each event will be conditions and actions will be consequence of that conditional statement so each event regarding to its guards will perform a possible action by the method.

The following code snippets show an Event-B event and its equivalent implementation in Dafny. The event models the way in which a new key and its value is added to the map. The Dafny code contains both implementation and specification which describes the desirable behaviour of the method:

```
Add1 ≙
        any k,v
        where
                @grd1: k∉ran(keys)
        then
                @act1: keys := seqPrepend(keys, k)
                @act2: values := seqPrepend(values, v)
        end
```

```
method Add(k: KEY, v: VALUE)
        ...
        ensures k !in old(keys) ==> keys == [k] + old(keys)
                && values == [v] + old(values);
  {
        ...
        if(k !in keys){
                keys := [k] + keys;
                values := [v] + values;
        }
        ...
  }
```

There are some important issues about linking Event-B models with Dafny implementations which should be addressed and are subject of this ongoing research. There is a considerable syntactic gap between Event-B and Dafny. Event-B language with the help of the Theory Plug-in can be extended. This very interesting feature can help to decrease the gap. Although this gives rise to the problem of how translating newly defined operators to Dafny implementations where they dont have an equivalent in Dafny. Another issue to be investigated is refinement. Answering the question that how and which refinement levels should be used for generating specification and code is important. Is there a specific guideline that should be followed during modelling phase in order to be able to generate specification and code from the Event-B models is another valid question. After answering these questions and proposing a comprehensive approach to generate Dafny specification and code from the models, developing a tool for automation of the code generation process should be planned.

References

[1] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, Conference Proceedings, pp. 348–370.

[2] K. R. M. Leino and R. Monahan, "Dafny meets the verification benchmarks challenge," pp. 112–126, 2010.

[3] M. Butler and I. Maamria, "Practical theory extension in event-b," in *Theories of Programming and Formal Methods*. Springer, 2013, pp. 67–81.

A Practical Approach for Validation with Rodin Theories

Daniel Plagge and Michael Leuschel

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

1 Introduction

The Theory plug-in [2] provides the ability to append new data types and operators to Event-B's mathematical toolkit and extend Rodin's proof rules.

E.g. the railway sector case study [1] makes use of the Theory plug-in to define new mathematical operators and to facilitate the proofs.

We give a brief overview about how operators can be defined in a theory and how ProB can deal with that kind of definition. In the next paragraphs, we illustrate which ways the Theory plug-in offers to define new data types and operators and how ProB can handle them automatically. We also show the limitations of the approach and how ProB will be able to handle problematic operators by adding annotations manually.

2 Datatypes

The Theory plug-in allows to define recursive data types similar to algebraic data types known from functional programming languages. We take the theory of inductive lists as an example. A list is either empty or has a first element connected to the rest of the list which again is a list. We have two constructors:

- nil returns the empty list.
- $cons(head, tail)$ constructs the list which has $head$ as first element followed by the list $tail$.

Both constructors constitute a new operator in Event-B. The data type $List$ itself is also an operator which returns the set of all lists over a given set.

Lists makes also use of parametric polymorphism, another feature of theories. E.g. for the list example, we have a generic type parameter T to allow lists for arbitrary sets. The argument $head$ of the operator $cons$ is of type T, the argument $tail$ of type $List(T)$.

The datatypes of the Theory plug-in are similar to the free types of the Z notation. In a previous project [3] to support the Z notation by ProB we

** Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE).

implemented an internal representation of free types. To re-use this effort we adapted the implementation in a way that it now also supports type parameters. For animation the constructors and destructors are directly replaced by the internal syntax constructs for free type constructors and destructors.

We can also encounter the situation that the type argument of a list is not only a type. E.g. the expression $List(1..9)$ specifies the set of lists whose elements can be integers between 1 und 9. Internally we use comprehension sets like

$$List(1\,..\,9) = \{\ l \cdot l \in List(\mathbb{Z}) \land \\ (l = cons(h,t) \Rightarrow h \in 1\,..\,9 \land t \in List(1\,..\,9))\ \}$$

Note that this definition is recursive, in the comprehension set, we refer again to $List(1\,..\,9)$. We explain the consequences of this in more detail below in section 4.

3 Directly defined operators

Directly defined operators can directly expressed by another predicate or expression. Let's take the theory of sequences as functions as an example. The set of sequences over a set S is defined by the operator $Seq(S)$. $Seq(S)$ is directly defined by the expression $\{n \mapsto f \mid n \in \mathbb{N} \land f \in 1\,..\,n \rightarrow S\}$. Thus, a sequence is a pair whose first element is the size of the sequence, and its second element is a total function which defines the elements at each position. Another example for a directly defined operator is the operator $seqIsEmpty(s)$ to check whether a sequence s is empty. It is defined by the predicate $\mathrm{prj}_1(s) = 0$.

We can animate the behaviour of such an operator by just replacing the operator with the given definition.

4 Recursively defined operators

Recursively defined operators are defined by a distinction of cases for an operator argument. E.g. the size $listSize(l)$ of an inductive list can be defined recursively by:

$$listSize(nil) = 0$$
$$listSize(cons(head, tail)) = 1 + listSize(tail)$$

We implemented recursive operators by translating an application of the operator to an B/Event-B function application with the particularity that the function is recursively defined:

$$listSize(l) = \{\ a, r \cdot a = nil \Rightarrow r = 0 \land \\ a = cons(head, tail) \Rightarrow r = 1 + listSize(tail)\ \}(l)$$

Please note that we omitted existential quantifiers in the formula above to make it more readable. If an operator defines a predicate instead of an expression, we replace the application of the operator by a membership test.

ProB's support for recursive functions was limited to global functions that depend not on a current state. But the theory plug-in's datatypes can be used in a more general style. E.g. the expression $List(X)$ specifies a the set of lists whose elements in the set X. But X can be a variable of the machine or even a parameter of an operation. Thus we needed a more flexible way to specify recursive functions. We introduced a new element $recursive(I, S)$ in the internal abstract syntax tree of ProB that allows us to define an identifier I that refers to the specified comprehension set S. We had to adapt the internal datastructure that represents symbolic sets such that the recursive definition is respected when evaluating the set.

The translation of $e \in List(X)$ with X beeing a set of integers is now

$$e \in \text{recursive}(L, \{\ l \cdot l \in List(\mathbb{Z}) \wedge \\ (l = cons(h, t) \Rightarrow h \in X \wedge t \in L)\ \}).$$

5 Axiomatic defined operators

The most flexible approach to define the behaviour of an operator is to specify a set of axioms. We currently do not see any feasible way to provide generic support for expressions that use these operators.

An example for an axiomatic definitions is the summation operator, whose behaviour is defined by the following three axioms:

$$\begin{array}{l} \text{axm1 } SUM(\varnothing) = 0 \\ \text{axm2 } \forall t, x \cdot t \in T \wedge x \in \mathbb{Z} \Rightarrow SUM(\{t \mapsto x\}) = x \\ \text{axm3 } \forall s, t \cdot s \in T \nrightarrow \mathbb{Z} \wedge t \in T \nrightarrow \mathbb{Z} \wedge s \cap t = \varnothing \\ \qquad\qquad \Rightarrow SUM(s \cup t) = SUM(s) + SUM(t) \end{array}$$

6 Annotations for ProB

In the previous sections, we explained how operators can be animated by analysing their definition. For practical purposes it can be more effective to instruct ProB directly how an operator should be handled. We give two examples where an alternative to the standard behaviour described above is preferred.

6.1 Transitive closure

The *closure* operator which returns the transitive closure of a relation r is defined by using a direct definition:

$$closure(r) = fix(\lambda s \cdot s \in S \leftrightarrow S \mid r \cup (s; r))$$

The fixpoint operator fix is also defined by a direct definition

$$fix(f) = inter(\{s | f(s) \subseteq s\}).$$

In summary, the operator could theoretically be handled automatically by ProB by replacing $closure(r)$ with the expression

$$inter(\{s|r \cup (s;r) \subseteq s\}).$$

In practice, the number of possibilities for the quantified variable s in the expression becomes large very fast. If s is of type $T \leftrightarrow T$, s has $2^{|T|^2}$ possible values. For $|T| = 4$, ProB must check 65536 sets, for $|T| = 5$ already more than 33 million sets. Thus for most models, ProB is not capable of handling this direct definition effectively.

On the other hand, ProB has already built-in support for classical B's `closure1` operator. To handle the closure operator effectively, we have added an annotation to the operator definition that instructs ProB to use its built-in closure support (rather than the direct definition). The built-in closure support has no problem dealing with large relations, as the following transcript from ProB's REPL (Read-Eval-Print-Loop) shows:

```
>>> f=closure1(%x.(x:1..5000|x*x)) & f[{2}] = r
Existentially Quantified Predicate over f,r is TRUE
Solution:
       f = #5077:{(1|->1),(2|->4),...,(4999|->24990001),(5000|->25000000)} &
       r = {4,16,256,65536}

>>> g=closure1(%x.(x:1..5000000|x*x)) & g[{2}] = r
Existentially Quantified Predicate over f,r is TRUE
Solution:
       g = closure1(%x.(x : (1 .. 5000000)|x * x)) &
       r = {4,16,256,65536,4294967296}

>>> h=closure1(%x.(x:NATURAL|x/2)) & h[{2**40}] = r
Existentially Quantified Predicate over h,r is TRUE
Solution:
       h = closure1(%x.(x : NATURAL|x / 2)) &
       r = {0,1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
       65536,131072,262144,524288,1048576,2097152,4194304,8388608,16777216,
       33554432,67108864,134217728,268435456,536870912,1073741824,
       2147483648,4294967296,8589934592,17179869184,34359738368,
       68719476736,137438953472,274877906944,549755813888}
```

The first expression shows that the transitive closure `f` of a 5000 element relation can be computed quickly (in about 50 ms). The last two expressions show that, for large or infinite relations, ProB reverts to computing the closure lazily on demand. The computation is instantaneous (10 ms or less).

6.2 Sum and Product

The sum operator as described above cannot be animated by ProB without additional information because axiomatic definitions are not supported. By explicitly

instructing ProB we can compute the sum operator by using the classical B sum operator Σ:

$$SUM(s) = (\Sigma t, x \cdot t \mapsto x \in s | x)$$

Currently, ProB supports a theory with the sum operator together with a product operator (Fig. 1). We have added an annotation that instructs ProB to use its built-in support for sum and product.

6.3 Implementation of the Annotation Mechanism

ProB checks if there exists a file with the name $\ll theory\_name \gg .ptm$ in the same directory of the theory and reads its content. A first version only allows tags that show ProB that this is an operator with an alternative implementation. E.g. "SUM is the summation operator".

We examine upcoming theories to check whether a more flexible approach is beneficial. E.g. the file could contain instructions how the value of an operator can be computed effectively. We currently do not see a need to provide this feature for the theories known to us.

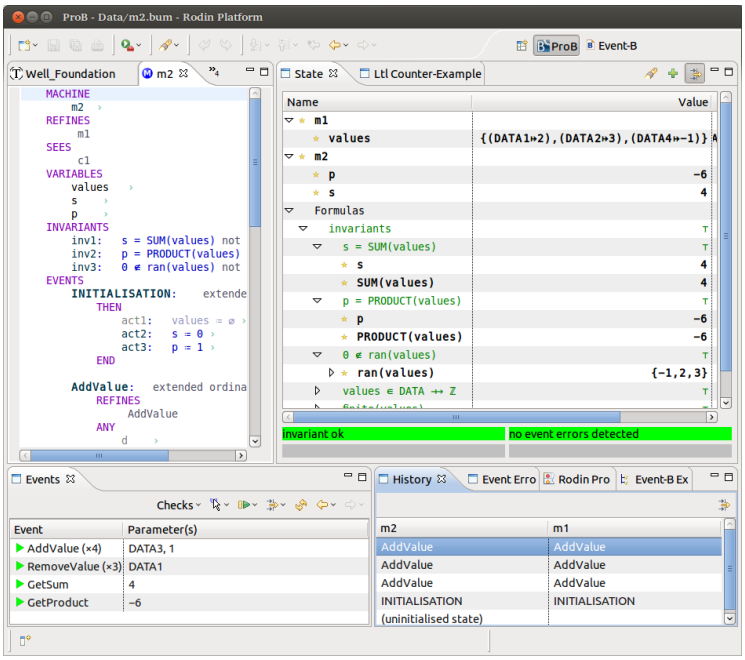

Fig. 1. Screenshot of an animation using the theory of sum and product

7 Theorems in Theories

A theory might also contain arbitrary theorems. They are usually provided to facilitate proofs. ProB completely ignores the theorems because it is currently not our goal to check the correctness of the theory but to animate the models that use it.

In future, it could be interesting to provide support for checking theories, too. This would be especially helpful for users who define their own theories.

8 Pretty printing

Currently we support the operators above by replacing an occurence of an operator directly in the syntax tree by its definition.

One drawback of this approach is that it is not obvious anymore what the original expression in the syntax tree was. This aspect is usually not so important as long the expressions are evaluated in ProB, but it might irritate the user when we present parts of the specification to him. To make the pretty-printed expressions more readable, we add annotations for the pretty-printer. Instead of printing the e.g. comprehension set defining a recursive function, the original expression will be shown.

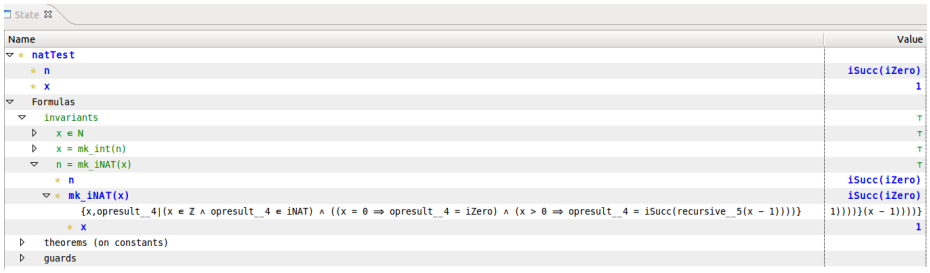

Fig. 2. Screenshot of a pretty-print showing the internal representation

The current implementation of this approach still has some rough edges. Figure 2 shows a screeshot of ProB where the invariant of an animated model shown. We can see an expression $mk\_iNAT(x)$ where mk_iNAT is an operator defined in a theory of natural numbers. Internally, the operator call is a function application with x as argument and a recursive comprehension set as function. The code that allows the user to analyse expressions currently ignores the annotations for the pretty-printer and shows both the recursive comprehension set and x as sub-expressions. Thus we can see the internal representation:

$$\begin{aligned}\{x, opresult\_\_4 \cdot\ & (x \in \mathbb{Z} \wedge opresult\_\_4 \in iNAT) \\ & ((x = 0 \Rightarrow opresult\_\_4 = iZero) \wedge \\ & (x > 0 \Rightarrow opresult\_\_4 = iSucc(recursive\_\_5(x-1)))) \ \}\end{aligned}$$

This expression is not a valid Event-B expression because the definition of the recursive identifier $recursive\_\_5$ is not visible to the user. Internally it is denotes the comprehension set.

9 Currently supported standard theories

The developers of the Theory plug-in contributed a project with a set of theories. These are candidates for standard theories when the next version of the plug-in will be released.

All operators defined in these standard theories **are now supported** by ProB:

- "Sum and Product" defines operators to compute the sum or product of integer sets. It is fully supported by specific tagged operators as explained above (6.2).
- "Binary Tree" defines a new polymorphic data type to represent binary trees and operators on these. The structure is very similar to the "List" theory. It uses recursively defined operators, all operators are supported.
- "Bool Ops" defines operators on Boolean values (AND, OR, NOT). It uses direct operator definitions, all operators are supported.
- "Fix Point" uses a direct operator definition, is theoretically supported but usually to complex to animate.
- "List" defines a polymorphic datatype and operators on lists. The operators are all recursively defined, all operators are supported.
- "Main" contains no operators, just theorems and proof rules which is not relevant for animation.
- "Seq" is a theory over sequences. All operators are defined by direct definitions and are supported (see 3).
- "closure" is a theory which defines an operator that yields the transitive closure of a relation. It supported by a specific tagged operator (see 6.1).
- "Natural" is a theory of natural numbers where

There is some demand within the Advance project to support a theory of real numbers. (This would allow to express certain models in Event-B rather than requiring co-simulation with continuous models.) This would, however, require considerable implementation effort to extend the ProB kernel for real numbers and the associated operators.

References

1. J. Colley. ADVANCE Deliverable D1.2, Proof of Concept Application in Railway Domain. Technical report, 2012.
2. I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.
3. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer-Verlag, 2007.

Toolbox for penetration testing based on Rodin and ProB

Aymerick Savary[1,2], Marc Frappier[1], and Jean-Louis Lanet[2]

[1] University of Sherbrooke
[2] University of Limoges

Penetration testing is used to find security weaknesses. These tests can be obtained using specification mutation and model-based testing. In our research, we propose to use Rodin and ProB to generate these penetration tests. The Vulnerability Tests Generator (VTG) [2] is the implementation of our methodology. Its input is a Rodin project representing the accepted behaviour of a SUT (System Under Test). Abstract penetration tests are produced and they verify the rejection by SUT of rejectable behaviours (w.r.t. to the specification). This is achieved by negating some parts of a guard or axioms in the original specification.

The first version of VTG [1] permitted to discover security failures on some smart cards. However, its architecture made it difficult to adapt to methodology improvements. The new version 2.0 is based on components architecture. It makes it easier to add, remove or improve components.

The VTG is a generic program that be used to test arbitrary systems. In addition to the VTG, a program for generating initial models and another one for generating concrete tests from abstract tests are required. These programs are specific to a given system. Figure 1 shows the major steps of our methodology and they will be described in the next sections.

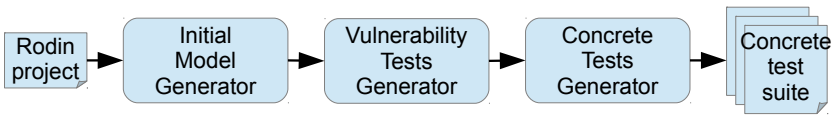

Fig. 1. Toolbox groups

We have used this methodology to generate vulnerability tests for the Java Card byte code verifier (JCBCV). JCBCV is responsible for verifying an applet wrt to the static constraints described in the Java Card virtual machine specification. Verification is split in two processes: structure verification and type verification. The structure verification will be represented by Event-B contexts and type verification by Event-B machines.

1 Initial Model Generator

Multiple transformations are made on the input Rodin project. It is necessary to keep a copy before processing. The *CloneRodinProject* plugin duplicates a Rodin project.

An event has a guard, itself composed of parts. We distinguish two types of them, those used for penetration testing and those not used. The plug-in *GuardConcatenator* concatenates used parts into a single one, and not tested parts into another one. Thus, after concatenation, each event has a guard consisting of only two parts: one used for penetration testing and one not used.

For the structure verification process, we use a model composed of three contexts: structure, constraints and values. The *CAP2Rodin* plugin provides the value context by compiling a CAP file (*Converted APplet*), which is an optimized version of a Java Card Class file. All these steps are illustrated in Figure 2.

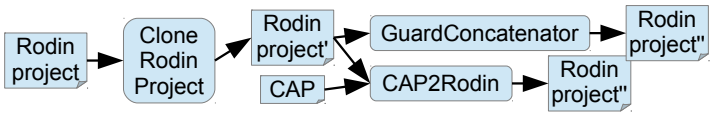

Fig. 2. Initial Model Generation plugins

2 Vulnerability Tests Generator

VTG is split into two main parts, *ModelDerivator* and *AbstractTestsGenerator*, which correspond respectively to specification mutation and model-based testing. The model derivation is also divided into two steps: *FormulaNegator* and *ModelNegator*.

For the *FormulaNegator* plugin, we have reused the grammar provided by Rodin. This grammar is based on the API TOM and can easily be integrated into a Java program. The formulas to negate are axioms and guards. To apply our rules of negation to a formula, we have implemented a TOM analyzer and a Java rewriter class. To save mutations, we have extended the Rodin database and its viewers.

The mutant contexts are obtained by replacing each axiom by one of its negations. The number of mutated axioms per mutant context is a parameter set by the user, according to his/her requirement. In the mutant machines, the guard of the Event Under Test (EUT) is replaced by its mutation. Conservation of the original event, the EUT action or the number of possible executions of the EUT, are also parameters. These transformations are performed by *ModelNegator* plugin.

For the generation of test suites, the *AbstractTestsGenerator* plugin uses the model checker ProB. For contexts, we generate a set of values to satisfy the constraints. For the machines, we generate traces that reach the EUT, *i.e.* preambles of the tests. The implementation of our tests generator is based on the command line interface of ProB. It is therefore difficult to precisely define our test criteria and the information we want to extract. To better control this process, it would be interesting to use the ProB API. All these steps are illustrated in Figure 3.

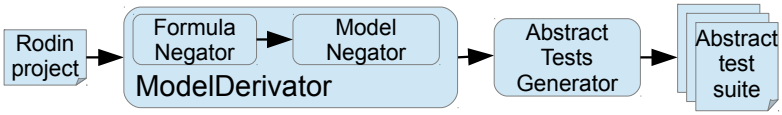

Fig. 3. Vulnerability Tests Generator plugins

3 Concrete Tests Generator

The concretization of abstract tests is currently based on a prototype. The *Rodin2CAP* plugin will be helpful to transform abstract tests into CAP file format. These tests will be easily sent to a smart card in order to verify it's resistance against attacks.

4 Conclusion

The Rodin platform allows us to implement our methodology and apply it to real cases. The new architecture makes our tools more generic and therefore applicable to various problems.

References

1. A. Savary, J.-L Lanet, M. Frappier, T. Razafindralambo, J.D.: VTG - Vulnerability Test Generator, a Plug-in for Rodin. Workshop Deploy 2012 (2012)
2. Savary, A., Frappier, M., Lanet, J.: Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. Integrated Formal Methods (2013)

iUML-B Statemachines

New Features and Usage Examples

Colin Snook

University of Southampton
cfs@ecs.soton.ac.uk

iUML-B is an integrated form of the classical UML-B graphical front-end for Event-B[1] . iUML-B consists of a collection of diagrammatic modelling notations and tools, which can be used to augment Event-B models. Currently a state-machine modelling plug-in has been released and a class diagram plug-in is under development. iUML-B is based on the generic Diagram Extensions framework for Event-B, which is in turn based on the Event-B EMF Extensions and Event-B EMF frameworks[2]. iUML-B was initially developed within the Deploy project[1] and is now being continued under the Advance project[2].

A new version of the iUML-B State-machines plug-in has recently been released. The new version introduces additional diagrammatic notations in response to user-driven modelling experiences. It provides significant other enhancements including more flexible options for generating Event-B. In this talk, we will illustrate both the previous and the new enhanced features of the iUML-B State-machine tool via a series of small abstract examples that use a variety of modelling styles. The material developed for this talk will form a new user manual for the iUML-B State-machines which will thereafter be available on the Event-B wiki and via the Rodin handbook. The new features that will be covered are briefly summarised below:

Transitions may own event features

Transitions may now have the same kinds of children as events (i.e. parameters, witnesses, guards and actions). Hence it is no longer necessary to edit the events of a machine directly. This reduces the amount of switching between editors. If several events are elaborated by a transition, any parameters, witnesses, guards or actions of the transition will automatically be replicated into each elaborated event.

Junctions for compound transitions

Junctions allow compound transitions to be formed that have a disjunctive source (i.e. the transition is enabled when in any one of several source states). To avoid conditional actions, only guards may be attached to transition segments that target a junction. Several transitions may also exit a junction so that the same guard disjunction is contributed to each outgoing transition. (c.f. UML: http://www.omg.org/spec/UML/2.4.1/)

---

[1] DEPLOY: EU Project IP-214158, www.deploy-project.eu
[2] ADVANCE: EU Project IP-287563, www.advance-ict.eu

Forks and Joins for entering/leaving parallel nested state-machines

When entering or leaving a state with several parallel nested state-machines, it is necessary to indicate the target or source states, respectively, in the nested state-machines. A fork-join pseudo-state is now available to help specify this. (c.f. UML: http:// www.omg.org/ spec/UML/2.4.1/).

Multiple instances of State-machines

In classical UML-B, classes could own state machines so that there were many instances which could be in different states at any time. The new version now provides a way to *lift* state-machines to a set of instances by specifying an *INSTANCES* set for the state-machine. When the new iUML-B class diagram plug-in is released, placing a state-machine inside a class will cause this property to be configured to the class instances. However, the *INSTANCES* property can also be used independently of class diagrams in order to provide state-machine lifting.

State-machine elaboration of data representation

The previous version always generated a new enumeration for every state-machine. Sometimes it is more convenient to have several state-machines with the same type or to utilise an existing enumeration for type. A special case of this is a 2-state state-machine that is most naturally represented as a single Boolean value. The new version allows a state-machine to be linked to existing variables instead of generating a new one. (The variables type must correspond to an enumeration provided by the state-machine's states). This approach also allows a state-machine to be split into several overlaid diagrams, for example, to segregate some kinds of transitions.

Improvements to State-machine Animation

The state-machine animation has been updated to support these new features as well as having an important enhancement of its own. When an state-machine is animated, any other state-machines in the same model that are currently opened for editing are also linked to the same animation. If a BMotionStudio visualisation is open for the same machine, it too is animated in the same animation. This allows BMotionStudio visualisations and multiple state-machines to be simultaneously animated for validation.

References

1. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008. (February 2008)
2. Savicks, V., Snook, C.: A Framework for Diagrammatic Modelling Extensions in Rodin. In: Rodin Workshop 2012, Fontainbleau. (2012)
3. Snook, C.: Modelling Control Process and Control Mode with Synchronising Orthogonal Statemachines. In: B2011, Limerick. (2011)

EB2RC: A Rodin plug-in for visualising Event-B models and code generation

A tool development presentation

Zheng Cheng[2], Dominique Méry[1], and Rosemary Monahan[2]

[1] LORIA, Université de Lorraine, Campus Scientifique, BP 70239, 54506 Vandœuvre-lès-Nancy, France
[2] Computer Science Department, National University of Ireland Maynooth, Ireland.

We present EB2RC, a plug-in for the RODIN platform, that reads in an EVENT-B [1] project and uses the control framework introduced during its refinement to generate both a graphical representation of the executable algorithm, and a recursive algorithm[3]. The transformations involved in generating the executable code and the proof of their correctness are presented in [3]. These are (a) the transformation from an EVENT-B specification into a concrete recursive algorithm and (b) the transformation from the recursive algorithm into its equivalent iterative version.

Our plug-in is written in Java and targets the Rodin platform (v2.7), interacting with the Rodin API to generate the recursive algorithm corresponding to the input. This recursive algorithm is used to generate LaTeX, text and graph representations which help the user understand the algorithm (see Figure 1).

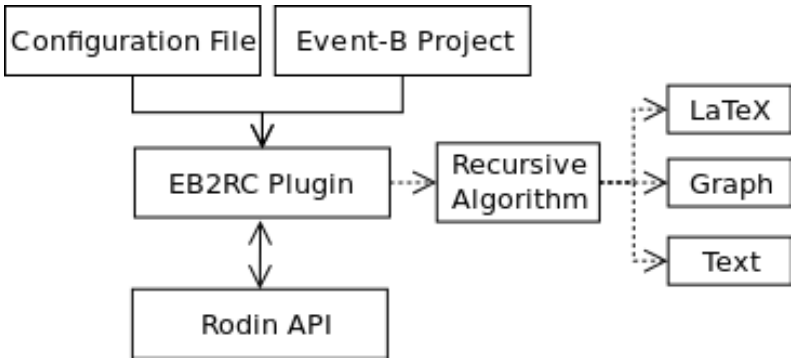

Fig. 1: Overview of EB2RC Plugin

The first step in using EB2RC to translate an EVENT-B model into its graphical representation and a corresponding recursive algorithm, is to read in the EVENT-B model and a user-defined configuration file. For reasons of efficiency, EB2RC stores this information in a data structure which we refer to as *bEventObject*. To assist the generation of a recursive implementation from the Event B model, a textual representation of the algorithm is constructed via a pretty-print-procedure which we implemented in Java. The generated algorithm has a C# like syntax that can easily be translated to the language for the target platform langauge.

[3] We acknowledge the Irish Research Council and Campus France for the joint funding of this research collaboration via the Ulysses scheme 2013.

To produce the graphical representation, the EB2RC tool automatically formats the information in the *bEventObject* data structure into input for the Dot graph visualisation tool of *GraphViz* [4], drawing a circle to present each label, and a directed edge between two circular nodes to indicate that an event occurs. The guards of each event label the arrows, and the actions of the event are indicated in the text of the rectangular node belonging to each arrow. This graphical representation of the algorithm, as in Fig 2, supports a clearer understanding of the algorithm, and is a prerequisite for modularizing complex algorithms.

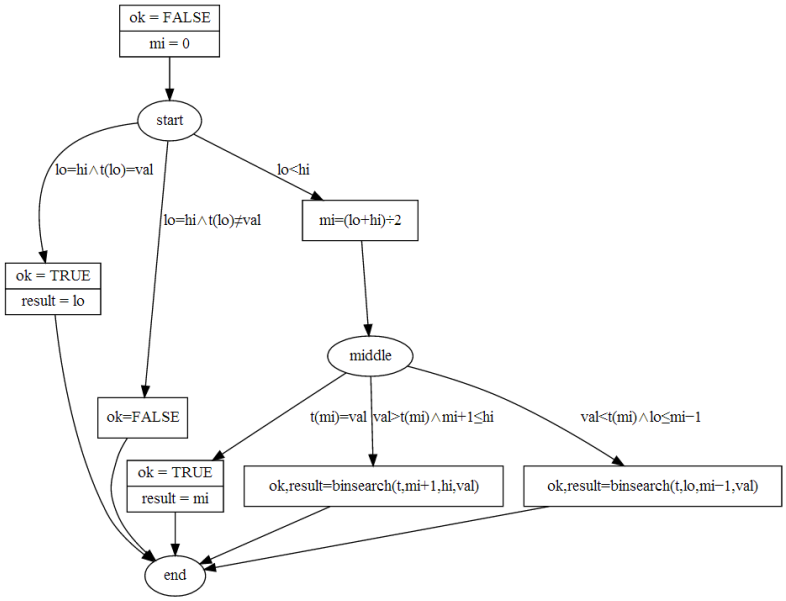

Fig. 2: Visualized Representation of the Binary Search Algorithm

Our work builds on a method for code generation that is detailed in [2, ?]. In this tool develoment presentation we will decribe how our tool generates its outputs: a graphical representation of the models algorithm and a concrete recursive implementation in C# like code. We will also illustrate the effect of our technique through case studies and their analysis.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Dominique Méry. A simple refinement-based method for constructing algorithms. *ACM SIGCSE Bulletin*, 41(2):51–59, 2009-06.
3. Dominique Méry and Rosemary Monahan. Transforming event b models into verified c# implementations. In Alexei Lisitsa and Andrei P. Nemytykh, editors, *VPT@CAV*, volume 16 of *EPiC Series*, pages 57–73. EasyChair, 2013.

[4] http://www.graphviz.org/

Composition Operators for Event-B. CO4EB Rodin plugin.

Idir Ait-Sadoune* and Yamine Ait-Ameur**

SUPELEC, Gif-Sur-Yvette, France*
idir.aitsadoune@supelec.fr
IRIT - ENSEEIHT, Toulouse, France**
yamine@enseeiht.fr

1 The proposed approach

Our approach of modelling composition operations of a process algebra in Event-B follows the formal modelling rules formally defined in [1]. We propose to encode these operators in Event-B, using an explicit variant to encode the events order and successive refinements. Each process algebra expression defined by the rule $A_0 ::= A_1\ OP\ A_2$ is modelled by two Event-B models. The first one is associated with the left hand side of the rule and contains only one event evt_{A0} associated with the action A_0. The second model is a refinement of the first one and corresponds to the right hand side of this rule. Two new events evt_{A1} and evt_{A2} associated with the actions A_1 and A_2 are added in the refinement. These events carry the semantics of the OP operator and of the right hand side of the expression. The firing order of the events is determined by introducing an explicit decreasing variant. The new events are fired and when they are completed, the refined event evt_{A0} is fired.

2 The Event-B formalisation

To illustrate our approach, we show the Event-B templates associated to the expression $A_0 ::= A_1 \mid\mid A_2$ (concurrency operator) with A_2 defined by the expression $A_2 ::= A_{21} >> A_{22}$ (sequence operator) ($A_0 ::= A_1 \mid\mid (A_{21} >> A_{22})$). The associated semantics is interleaving, imposes to describe all the possible behaviours (all the possible traces). It uses the interleaving of Event-B events.

Three events evt_{A2}, evt_{A21} and evt_{A22} formalizing the three actions A_2, A_{21} and A_{22} are defined in the Event-B Machine formalizing the sequence operator (figure 1). This Machine uses a variant expressed by the $varSeq$ variable initialized to the value 2. The evt_{A21} and evt_{A22} events are declared "*convergent*" and once the guard of A_{21} is evaluated to "*true*" ($varSeq = 2$), the event is fired and the variant is decreased. The evt_{A22} event can be fired after evt_{A21} event, when its guard is evaluated to true ($varSeq = 1$), and the value of $varSeq$ is set to 0. The evt_{A2} event ends the sequence operation of A_{21} and A_{22} actions ($varSeq = 0$).

The same Machine contains two others events evt_{A0} and evt_{A1} corresponding to the two actions A_0 and A_1. This part of this Event-B machine formalises the concurrency operator and uses a variant expressed by the sum of $varPar_1$ and $varPar_2$ variables that are both initialized to the value 1. evt_{A1} and evt_{A2} events are declared "*convergent*". Thank to the variant and if the three events evt_{A1}, evt_{A21} and evt_{A22} have their guard evaluated to "*true*", they are fired

in parallel in an interleaving manner. The animation of this example with ProB animator [2] gives the following traces $evt_{A1} >> evt_{A21} >> evt_{A22}$, $evt_{A21} >> evt_{A1} >> evt_{A22}$ and $evt_{A21} >> evt_{A22} >> evt_{A1}$. As the evt_{A2} event ends the sequence of the events evt_{A21} and evt_{A22}, the evt_{A0} event ends the parallel operation of evt_{A1} and evt_{A2} events.

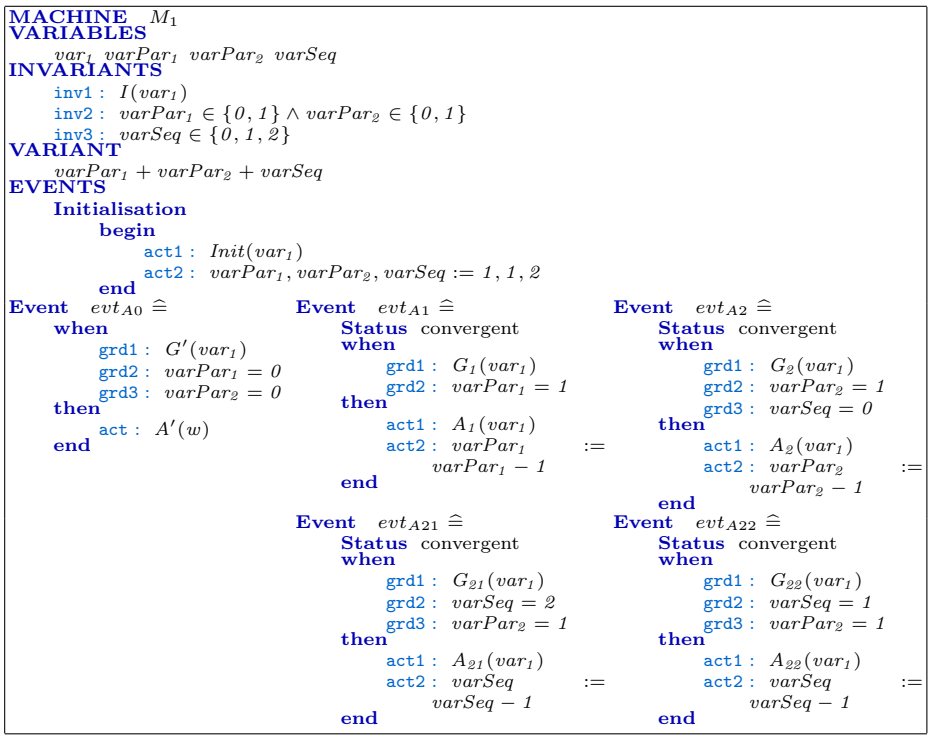

Fig. 1. Encoding concurrency and sequence operators in Event-B

3 The CO4EB plugin

The Event-B based approach, proposed for formal modelling and verification systems [1], defines different transformation rules from a composition operator definition to an Event-B model. We have automated this transformation process in the CO4EB plugin[1]. This plugin builds a RODIN project from process algebra expression of the form $A_0 ::= A_1\ OP\ A_2$ following the approach defined in the previous section. A video showing how the CO4EB plugin can be used is available in this link[2].

References

1. Ait-Ameur, Y., Baron, M., Kamel, N., Mota, J.M.: Encoding a process algebra using the Event B method. International Journal on Software Tools for Technology Transfer (STTT) 11(Number 3), 239–253 (2009)
2. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Formal Methods, International Symposium of Formal Methods Europe (FME'03). LNCS, vol. 2805, pp. 855–874 (2003)

[1] CO4EB update site: http://idir.aitsadoune.free.fr/tools/updatesite
[2] CO4EB video: http://www.youtube.com/watch?v=oQwsu8CQFOw

CODA Update: New Features for 2014

Neil Evans[1], Helen Marshall[1], James Sharp[1],
Michael Butler[2], John Colley[2], Colin Snook[2]
[1] AWE plc, Aldermaston, RG7 4PR, UK
[2] The University of Southampton, SO17 1BJ, UK

CODA is a framework for designing embedded systems that comprise software and digital electronic hardware that has been developed through a collaboration between AWE plc and the University of Southampton. A CODA model comprises components that interact via communication over connectors, and is provided through a graphical environment plug-in (Component Diagrams) within the Rodin tool. The CODA meta-model is the foundation on which the graphical environment plug-in is built: it represents the various CODA constructs in the tool-set and facilitates the translation of CODA models to Event-B. The CODA tooling and methodology are currently being exercised on an example at AWE and this work is highlighting further possible enhancements to the framework.

Extensive use is made of the underlying Rodin engine, and several of the existing Rodin plug-ins support the (also evolving) CODA methodology – in particular:

- iUML-B state machines are used to model behaviour of individual CODA components, and state machine animation is used to validate models;
- ProB is used to animate and model check the translated Event-B to give confidence that the system will not deadlock and that specified properties will not be violated;
- the Proof Obligation Generator (POG) and Rodin's theorem proving capabilities are used to demonstrate correctness of translated Event-B models.

At the previous Rodin Workshop, in 2013, several updates and features of CODA were unveiled; these included the following:

- CODA operations (*PortWake, SelfWake, External, Method,* and *Transition)*. Operations represent the different methods of communication between components within a system and can be introduced through the Component Diagrams palette, and morphed (elaborated) into another operation through the Component Diagrams properties pane. Rules have been implemented in the plug-in to translate these different CODA operations *to Event-B.*
- A validation check prior to translation to Event-B. This ensures that a valid design is provided within the Component Diagrams view, e.g., ensuring that the data type of information passed over a connector adheres to the type specified within the connector definition.
- CODA Simulator. This is essentially an additional ProB view which is "aware" of the components in the model and hence enables the user to *simulate* the design (*cf*. a hardware simulator). It allocates and displays the events performed to the relevant components and provides features such as defining the number of time increments with which to advance the simulation. It also enables the external recording (to file) of the results (trace) of a simulation run.

Since the last Rodin Workshop, CODA has been exercised and further developed through its application at AWE. These further developments are summarised below:

- **SelfWake queues and refinement** – Through exercising the CODA tool-set it was realised that the implementation of the *SelfWake* queue was too restrictive with respect to the underpinning Event-B concept of refinement, specifying a discrete point in time when the *wake* action should occur. It was deemed appropriate to alter this definition of the *SelfWake* queue to enable event refinement to take place through the introduction of a time window instead of a discrete point in time. By specifying a time window (through a minimum and maximum value) in which the component could respond at the abstract level, it is now possible to refine *Skip* or *wake* events and hence introduce a finer granularity of timed behaviour within the model (as long as the time window is only ever reduced).

- **Colour selection for CODA components** – A recent piece of work evaluating the use of CODA to assess the safety theme of a system has demonstrated the need to visually distinguish different collections of CODA components. Moreover, it has given a suitable basis on which ideas relating to nested components can be formed; a feature that is currently under review and may become realised in the near future.

- **Co-Design Intermediary Notation (CODIN)** – At the previous Rodin Workshop it was highlighted that a move towards a Hardware Description Language (HDL) output, more specifically the Very High Speed Integrated Circuit (VHSIC) HDL (VHDL), was the intended next step for CODA. In order to follow refinement through its natural course, from the '*bird's eye view'* of a system, through to a concrete model, and down to the implementation level, the semantics of the target language must be introduced, and the model verified with regards to these additional semantics. Only once the system behaviour and language semantics are verified within the Rodin environment can the target language representation be generated and the resultant HDL description be guaranteed valid. The integration of CODIN within CODA provides this crucial link, allowing the transition from a concrete to an implementation ready design. This is possible within the Component Diagrams view through the morphing of either an Asynchronous or Synchronous state machine to a Process state machine, which results in the addition of HDL semantics, through code generation to Event-B, to a concrete CODA design. These HDL semantics currently support a direct translation to either VHDL or SystemC.

Current Application – In addition to these developments to the CODA tool-set, both Event-B refinement and the CODA methodology have been used in the assessment of a sizeable industrial problem, providing refinement from a 'true' abstraction through to a CODA, and then a CODIN level, to provide a cycle-accurate HDL design. It is felt that this investigation has pushed Rodin's capabilities, exploiting the newly introduced SMT solvers to aid in the automated verification of some 1,127 proofs of a total 1,140 proofs over twelve refinement layers. This percentage (98%) of automatically discharged proofs (through some setting adjustments and re-styling of invariants) is considered an impressive result.

CODA's development is set to continue for the next two years with a focus on simulation and validation of CODA based models. These formal models will be used to add further rigour to the current industry development processes. These enhancements have been identified as follows:

- **CODIN to VHDL** – the final stage necessary to enable a translation from an Event-B CODIN model to a VHDL representation. Such a VHDL model may then be used within a verification domain, providing a formally correct definition of the functional behaviour of a hardware component(s) upon which verification may be made with a manually produced VHDL implementation of the same component(s).
- **Refinement Refactoring** – removing the onus on the developer of tracing property changes within the Component Diagrams view of CODA, and thus removing human induced errors.
- **Transition to ProB2 for the CODA Simulator** – enabling CODA to leverage the new model checking features within ProB2, and benefit from the flexibility of the Groovy scripting for the verification of a CODA model's *responsiveness*.
- **Generation of suitable VHDL test bench stimuli and PSL assertions** – to drive the verification of manually implemented VHDL designs against the automatically generated, formally verified, VHDL model within EDA tools such as Mentor Graphics' Questa.

As alluded to, the application of CODA to AWE projects is becoming more prominent, and the CODA methodology will continue to be developed as the tool-set is applied to the technical challenges provided within these AWE projects, and the enhancements (both present and planned) exploited. The Southampton group continue to provide great insights into what is possible in modelling functional and safety requirements. These insights (based on System Theoretic Process Analysis) and the experiences of the users of the tool-set continue to drive the development of both the CODA methodology and its associated tool-set.

Rodin Multi-Simulation Plug-in

Vitaly Savicks, Michael Butler, John Colley
University of Southampton

Jens Bendisposto
Heinrich-Heine-Universität Düsseldorf

In this work we introduce a plug-in extension to Rodin that enables co-simulation of Event-B models and continuous-time models in the Functional Mock-up Unit (FMU) format. The plug-in aims at overcoming the lack of continuous time modelling capabilities in the current version of Event-B by leveraging the Functional Mock-up Interface (FMI) standard for tool-independent physical model exchange and co-simulation [1]. That makes it possible to carry out a standard formal development of discrete-event subsystems in Event-B and compose them for the simulation-based analysis with continuous-time models of environment, which can be designed in any FMI-compliant modelling tool[1].

The simulation semantics is loosely based on the concept of a simulation master algorithm from the FMI standard that splits the simulation time interval $[t_{start}, t_{stop}]$ into discrete communication steps $[tc_i, tc_{i+1}]$, where $0 < i \leq N,\ tc_i \leq tc_{i+1},\ tc_0 = t_{start},\ tc_N = t_{stop}$, at which the data exchange between the co-simulated interacting components is performed. The simulation of individual component is performed in steps of the size $hc_i = tc_{i+1} - tc_i$. Our implementation of the master algorithm is generic, i.e. allows any number and composition configuration of the simulated components, and uses a fixed size communication step hc_i. The simulation of FMU components is performed by the master via the FMI interface, while the Event-B components are executed using an experimental version of the ProB animator [2].

The Rodin Multi-Simulation plug-in provides a component diagram editor (see Figure 1) that allows to import and configure Event-B machines and

[1] According to `https://www.fmi-standard.org`, over 35 tools have some support of the FMI standard.

FMUs as components with input and output ports, which can be composed on the diagram via connectors. The simulation is performed on the diagram model, and the values of variables can be either plotted in real-time using a special Display component, or analysed later from a generated .csv file.

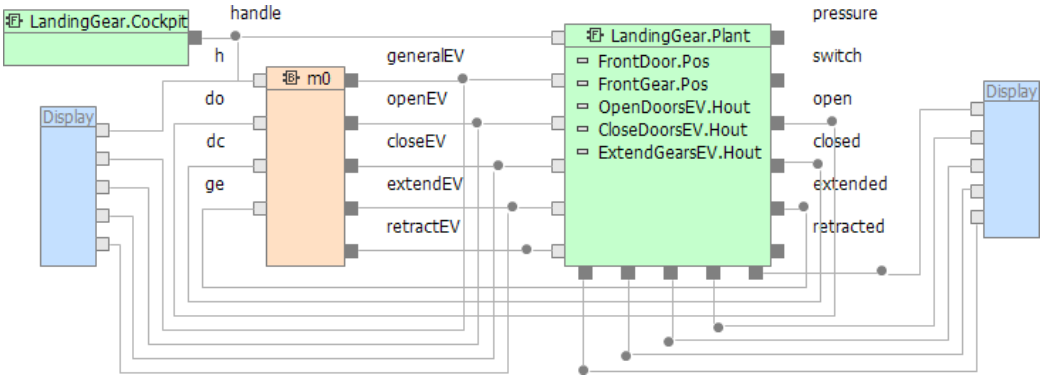

Figure 1: RMS component diagram in Rodin (co-simulates an Event-B machine *m0* and FMUs *LandingGear.Cockpit* and *LandingGear.Plant*)

As a work in progress we consider implementing an adaptive (variable step size) master algorithm, optimising tool's performance and validating it against the existing simulation-based approaches on a real-scale case study.

Acknowledgement: This work is funded by the FP7 ADVANCE Project (`http://www.advance-ict.eu`).

References

[1] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The Functional Mockup Interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

[2] Michael Leuschel and Michael Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

Code Generation – Tool Developments:

Andy Edmunds
University of Southampton
ae2@ecs.soton.ac.uk

This work has been funded by the FP7 ADVANCE Project (http://www.advance-ict.eu).

[1] Industrial Experience.

We collaborated on an industry-led assessment of Rodin/Event-B, and Tasking Event-B for code generation, with Thales Deutschland. The feasibility study involved modelling controller software, and the operating environment, of a simple fan controller. The aim of the study was to evaluate the complete Event-B methodology, from abstract specification to Java implementation. During the project various tool enhancements were completed. A list of 'significant' (Code Generation) feature requests was submitted, from suggestions made by Thales' engineers. They consider these features necessary for the tool to be useful in a production environment. The list could be used to guide future research and development.

[2] Java Translation Improvements.

The tool was updated (Sept 2013) largely driven by the activities of [1]. Improvements were made to the translators for generating Java code, including bug-fixes, and enhancements aimed at improving usability. Such as setting up the project with a Java Nature, and generating code in the current project. We added automatic flattening of invariants, and events; and automatic inference of typing, and parameter direction annotations. We also added environment interfaces, which simplifies the process of specifying interactions with driver software in the environment. The enhancements mean that a developer has to perform fewer steps, to generate code from an appropriately constructed model. The generated code can often be run in the same project as a Java application, with very little further configuration.

[3] C Translation: for Co-simulation with FMI.

The Functional Mock-up Interface (FMI) is a framework supporting co-simulation of distinct Functional Mock-up Units (FMUs). Recent work allows Event- B models to be co-simulated with FMUs, using ProB2. Existing code generation for Event-B supports generation of embedded controller implementations. We adapted the existing Event-B code generators, to produce code for use in FMUs. Controller code, generated from Tasking Event-B, can be compiled and packaged in an FMU. This allows generated controller code to be tested with a continuous model of the environment (in ProB2). Alternatively, it can be imported into a 3rd party simulator.

[4] Templates for Configuration and re-use.

This work arose from a feature request in [1]. The template-driven approach is a partial solution for tailoring code to be deployed on a specific target platform. The templates contain ‘boilerplate’ code, which would otherwise need to be hard-coded in the translator. We developed a tool to merge the code templates with code generated from the formal model.

We developed a lightweight approach, where tags (i.e. tagged mark-up) can be placed in the source templates. A 'generator' interface can be implemented to generate code, or additional information, when a tag is encountered. The template-processors may be of use to other plug-in developers, when wishing to merge an annotated text file with some automatically generated output.

[5] Theories for implementable Sets and Functions.

Code generation involving Event-B sets and functions has largely been avoided, in our work, until now. We have recently been looking at this issue, and have created Theories for implementable sets and functions, using polymorphic type parameters to describe implementations involving generic (parametrised) sets and maps.

For sets, we introduce operators for typing, and initialisation. We support a number of set operations (such as union/subtraction/intersection) and introduce new operators to facilitate code generation. A translation, to Java, of the new *implementable* type and operators is also defined in the theory. This is supported by a Java implementation of a set, which is intended to be a refinement of the model.

Functions are similar in that we introduce operators for typing, and initialisation, and a Java translation. We also add update and lookup operators. The translation to Java is backed by an implementation using a HashMap to store the domain and range values, as key-value pairs.

Smart Grids: Multi-Simulation, An Application

Brett Bicknell, Karim Kanso, Jose Reis
Critical Software Technologies, UK

Abstract:

Smart grids are an emergent technology that have the potential to provide substantial benefits to both energy producers and consumers, and have a typical system-of-systems architecture. In recent years, with the addition of micro-generation, electric vehicles and second life batteries, the traditional top-down power management infrastructure within power distribution networks has proven to be ineffectual in responding to these new technologies, resulting in the inability to keep supplied voltages within required limits. With the increased uptake of new technology in the future, this will only become worse.

Within the FP7 ADVANCE project, Critical Software Technologies have been exploring and applying the Rodin toolset within the smart grid domain, with particular focus on the stable control of the low voltage network. Verifying the control systems for the low voltage network is an interesting case study for the Rodin toolset as it exercises both its formal capabilities as well as its simulation capabilities (including multi-simulation). The formal framework of Rodin and Event-B is used to develop and verify models of the control devices, as well as the communications network and sensor units. The simulation used is twofold, first, animation is used to validate the formal models, and secondly, multi-simulation is applied to validate the composed system within a realistic continuous environment. The formal models are defined using Event-B and state machines through the UML-B tool, and the continuous models are defined using the modelling language Modelica. This talk discusses the modelling strategy including both the Event-B and Modelica models, with a special focus on our experiences and results from using the multi-simulation environment for complex systems engineering.

This work was funded by the FP7 ADVANCE Project (ICT-287563), www.advance-ict.eu.

Reflections on Formal Methods, Requirements and Software Engineering
Does *Software Engineering* exist?

Ken Robinson
School of Computer Science & Engineering
UNSW Australia

ABZ2014 Workshop

This talk probably differs from other workshop talks in that it is concerned with the use of Event-B in the development of software engineers and raises the question "what is software engineering, especially in contrast to computer science?". As an academic in the School of Computer Science and Engineering at UNSW I gained experience in the teaching of formal methods commencing with the work of Dijkstra and continuing with Z, Morgan's refinement calculus and Abrial and Sorensen's B Method using the B-Toolkit. The B-Toolkit was used for quite elaborate software projects(commencing around 2000) and Thai Son Huang developed a framework that provided a means of adding an interface to the execution of the models. On the way I also became interested in the discipline of *Software Engineering*. It is quite alarming that the concept and name of software engineering, identified at the 1968 NATO conference has progressed so little from that time.

Event-B was a significant game-changer and an important contribution to software engineering, a contribution that seems to be frequently unappreciated within computer science. I often hear the statement that formal methods should be used for software development, and while not disagreeing with the intention, it is not enough. An important part of any engineering design and implementation exercise is requirements analysis. If the implementation does not satisfy the requirements then the project has failed. Something like 80% of large software projects fail due to requirements failures. With many formal methods, requirements verification is very difficult and here the contrast of Event-B with the B-method and many other formal methods is dramatic. Unfortunately requirements verification cannot be achieved completely formally. At some stage the verification of whether a formal action satisfies an informal requirement is an informal judgement. The simple structure of an Event-B event: parameters, guards, actions and invariants makes such a determination much easier and more reliable.

In the Software Engineering program at UNSW, software engineering students take a series of workshops in which they progress from a study of requirements in which they devise a set of requirements for a given project to the next workshop in which they concurrently undertake a course in Event-B and and map their project requirements into Event-B. They spend considerable

time informally verifying their formal statements of the requirements and formally verifying the Event-B model. In this part of the exercise they also use animation to informally verify scenarios. In the third and final workshop the students manually translate the Event-B model into an appropriate programming language, for example Scala. This has been done informally but it has been my intention to develop decompositions to formally model the structure of the implementation.

The above workshops and Event-B lecture were managed by myself and a colleague, Peter Ho. In 2012 Peter and I ceased teaching and subsequently the workshops have dropped Event-B and the current formal content consists of what I would describe as computer science formal methods for programming.

The reason for raising the above at this workshop is to hear about the experiences and achievements of others in the distinction between what I would call *software engineering* formal methods and *computer science* formal methods.

I am also interested in getting an idea of how many universities are using Event-B in undergraduate software engineering programs. I get the impression that the answer might be *not many*. I believe that UNSW was the only university in Australia to be doing so and now it also has dropped it.

The following issues concerned with the understanding of software engineering will be raised at the workshop.

- Programming models versus engineering models.
- Is *Software Engineering* engineering?
- Are there differences between "conventional" engineering and software engineering, apart from, or perhaps because of, the implementation using software?

Applying and Extending the Event Refinement Structure Approach to Workflow Modelling

Dana Dghaym[1], Michael Butler[2], and Asieh Salehi Fathabadi[3]

University of Southampton, UK
dd4g12[1],mjb[2],asf08r[3]@ecs.soton.ac.uk

The Event Refinement Structure approach (ERS) is a diagrammatic notation that aims at structuring the Event-B refinement. Its tree like structure, inspired by Jackson Structure Diagrams (JSD), explicitly represents the relationship between the event at the abstract level and the corresponding concrete events, which decompose its atomicity during refinement. ERS supports the complex refinement in Event-B by defining different refinement patterns. These patterns are divided into four categories:

1. Sequencing Pattern
2. Logical Constructor Patterns: and, or, xor
3. Loop Pattern
4. Replicator Pattern: all, some, one

In addition to structuring the Event-B refinement, ERS explicitly describes the ordering of events, addressing another Event-B weakness where control flow is implicitly modelled via variables and event guards.

We focus on applying the ERS approach in modelling workflows, using the fire dispatch case study. While modelling the fire dispatch workflow, we encountered some restrictions with the current ERS patterns. We try to overcome these limitations by suggesting some extensions to the existing ERS constructs and defining a new construct to support unbounded replication.

We are planning to integrate our suggested ERS extenstions into the atomicity decomposition plug-in, a tool supporting the ERS approach in Rodin. The atomicity decomposition plug-in provides an automatic generation of part of the Event-B model related to the ordering of events and their relationships at different refinement levels. The ERS language is defined using the Eclipse Modelling Framework (EMF) meta-model, and then transformed into an Event-B EMF meta-model. The transformation is done using the Epsilon Transformation Language (ETL), which is a rule based model to model transformation language.

Introducing Pre-conditioned Operations in Event-B (by means of Guarded Events)

Jean-Raymond Abrial

Event-B is based on events restricted by some guards. In this presentation, we explain how we can transform some guarded events into pre-conditioned operations.

We remind the reader that in a refinement, guards and pre-conditions behave in different ways: guards are strengthen while pre-conditions are weakened. Their operational behaviors are also different. A guarded event *possibly occurs* (can be observed) when its guard holds, whereas a pre-conditioned operation is executed when it is *called* and when its *pre-condition holds*. A guarded event with a false guard just *wait* (no execution), whereas a pre-conditioned operation called with a false pre-condition results in a *crash*.

The usage of pre-conditioned operations is important when we are asked to develop and prove some programs dealing with procedures and procedure calls (operations and operation calls).

An operation P is specified by means of a pre-condition (defined on the formal parameters of P) and an action (its post-condition), not by means of a guard and an action like events are. Moreover, in case of a call to an operation P, the *actual pre-condition* has to be proved. The actual pre-condition is the predicate obtained by substituting the actual parameters of the call to occurrences of formal parameters that are present in the operation pre-condition.

In conclusion, and in view of these differences between the two, it seems completely impossible to define operations by means of events. This is the challenge we solve in this presentation. Notice that we do not want to modify Event-B in any way. We want to define a design pattern to obtain our desired goal.

The outcome of this allows one to establish a connection between Event-B and Classical-B. After explaining the theoretical aspect of this question, some interesting examples illustrates this approach.

Program Development in Event-B with Proof Outlines

Stefan Hallerstede

8 April 2014

Abstract

Proof outlines are a notation that combines programs with proof information. They consist of assertions that specify properties that hold at different program locations during the execution of a program. We discuss how Event-B can be used for program development using proof outlines to shape the program as it is constructed by refinement. The method is based on the program development approach of Event-B using anticipated events. Commands have the shape "ant • evt" where evt is the "current transition" and ant are the "anticipated transitions" that may occur beforehand. The control structures for the program notation are unconventional. In particular, loop and conditional statement are mixed "**LOOP** body **OR** branch **END**". This is a consequence of the using the refinement method of Event-B. However, the common control structures like loops and conditional can be recovered by simple transformations.

An additional proof rule is needed to introduce fresh intermediate states. This is based on the sequential composition rule of Morgan's refinement calculus.

Responsiveness and Event-B

James Sharp[1], John Colley[2], Helen Marshall[1],
Neil Evans[1], Michael Butler[2], Colin Snook[2]
[1] AWE plc, Aldermaston, RG7 4PR, UK
[2] The University of Southampton, SO17 1BJ, UK

Within the digital hardware domain it is becoming apparent that constrained random testing is no longer considered adequate for high-consequence systems. In order to provide the required level of confidence there is now a need to evaluate total path coverage (the exploration of every path through the system). However, for this total path coverage, the question should be asked: "*Which of the total paths within a design actually pertain to the expected behaviour; and which are unwanted; and more specifically, does the design always respond in the way it should?*" Event-B provides a partial solution to this question, providing reasoning about safety and functional requirements, but to our knowledge it does not provide a clear solution on assessing a system's ability to always respond in the correct and expected way to external stimuli. We call this property *responsiveness*. We therefore suggest an initial approach towards developing a suitable methodology for determining and assessing the responsiveness of a formal design by utilising the tools already available within Event-B, and by reasoning about a system with respect to its expected total *possible* and *impossible* paths.

During an application of Event-B to a high-integrity component by AWE and the University of Southampton, we discovered a need to reason about the responsiveness of a system. In our initial work we adopted a somewhat laborious approach to verifying total path coverage and, in the process, developed a method with which to reason about responsiveness. Our approach requires a specification upon which a composite technique, both manual and automatic, can be performed to determine the *possible* and *impossible* paths through the system, and provide confidence in the identified paths.

Alongside the development of the formal model within Event-B, two diagrammatic specifications captured two key refinement levels: the initial abstract level and the concrete level. In our case, the abstract diagrammatic specification identified a total of 12 paths through the system all of which were *possible*, and the concrete diagrammatic specification identified a total of 9067 paths (both *possible* and *impossible*). These specifications of the system behaviour formed the basis of our approach to responsiveness verification.

By using the abstract specification to partition the concrete specification into more manageable sections, the *possible* paths of the system can be determined through manual identification; that is, each sequence of events considered to be valid was walked through and recorded. From this manual identification, rules regarding the system became apparent, succinctly describing the bad behaviour that should not occur as a result of particular events, and thus capturing the groupings of the *impossible* paths. These manually identified *possible* paths, along with the rules regarding *impossible* paths formed the first element of our composite approach.

The second element of our composite approach utilised the concrete level specification and the MALvern Program Analysis Suite (MALPAS) to identify the total paths of the system, both the *possible* and *impossible*. The rules identified within the manual approach were then applied to the MALPAS output (using common Linux tools within a bash script) to remove the *impossible* paths. From the resultant *possible* paths of both approaches, a comparison was made between the results. As a consequence of this comparison stage, further rules identifying *impossible* paths for the automated approach were introduced until a point was reached where: the set of *possible* paths derived automatically was almost equivalent to the set of *possible* paths derived manually. Those paths available within the automatically derived set that were not amongst those manually identified, were evaluated and determined to be those omitted (through human error) in the manually derived possible paths. Thus the automatically deduced *possible* paths could be considered valid and used for assessing the responsiveness of the system.

This composite approach was deemed necessary because the automated technique relied on the identification of rules to reduce the number of paths to those that were *possible*, whilst the manual approach provided only verified *possible* paths and therefore was likely to identify a subset of the *possible* paths (human error meant that *possible* paths were missed). Indeed, through our experience it was discovered that from a total 9067 paths within the system, 304 *possible* paths were manually identified, and after several iterations of comparison between the automated and manually determined *possible* paths, a resultant 320 *possible* paths were identified as providing complete path coverage.

From these paths, model checking (using ProB) can be performed to verify the responsiveness of the formal model. Whilst laborious, walking through these *possible* paths within ProB provides the essential arguments about responsiveness that are needed to validate the high-integrity systems required in our industrial setting. It is suggested that a suitable plug-in, utilising ProB2 through a Groovy script, could be developed to automate the verification of the existence of these *possible* paths to reduce the manual effort in the future.

These *possible* paths not only inform and validate the formal model, but in our work will form the basis of the stimuli needed to drive VHDL test benches to ensure complete coverage of the *possible* paths is achieved in the verification of the implemented design.

The rules used to discover the *possible* paths of the system in the automated approach lend themselves to LTL and PSL definitions that can be used as assertions, against the formal model and the implemented design respectively.

Whilst people may argue that the derivation of the rules required to remove the *impossible* paths, and that therefore result in the complete set of *possible* paths, could have been performed without the need to determine the *possible* paths of the system, our experience has been that their manual identification provided the means to reason about the output from the automated approach. Without this manual step, there would be no means with which to guide or sanity check the number and style of the automatically produced *possible* paths; in essence each approach for determining the *possible* paths taken validated the other.

In other formal techniques, e.g. Communicating Sequential Processes (CSP), there exist approaches that evaluate a formal model and ensure not only that '*good behaviour cannot be refused*' (*Failures* refinement*)*, but also verify properties of *responsiveness* (the CSP *Revivals* refinement model). However, even within these other formal techniques, it is not always clear whether the definition used to validate and reason about the *responsiveness* of a formal model is the correct one, or if it allows additional unwanted behaviour. Hence, it is essential, even in approaches such as CSP's *Revivals* model, that an independent, manual evaluation is performed to validate the specification used in the automatic verification of a design.

From this recent work we have concluded that in developing a high-integrity system safety is only half the battle. When developing high-integrity systems responsiveness is just as important; the system must be verified to perform as expected, '*to always do something good'*, as well as ensure '*nothing bad happens'*. More significantly, we believe that the verification of responsiveness within a formal model must always include some manual effort in determining the valid paths, since without completely exploring the specification and understanding the full behaviour of the system invalid paths can easily be identified as *possible*.

In summary, it is clear to us that there is a definite need to formally evaluate a high-consequence system's responsiveness to its environment. We have proposed an initial approach to capturing and assessing the expected responsiveness of a system, but it is by no means a polished solution. A more efficient, tool focused solution in Rodin would enhance Event-B to become a '*de-facto'* language for providing the confidence that is essential for qualifying high-integrity systems.

Towards Patterns for Statemachine Modelling under Timing Constraints

Gintautas Sulskus [*], Michael Poppleton [†] and Abdolbaghi Rezazadeh [‡]

University of Southampton

This paper is inspired by our work on the case study of a dual chamber cardiac pacemaker. The pacemaker is a demanding real-time embedded application that interacts with a non-deterministic environment (the heart) via sensors and actuators.

Our implementation involves state based sequencing and timing aspects. We use the Rodin tool to formally model the pacemaker in Event-B. State based sequencing with concurrency elements is generated with iUML [2] – a UML-aided visual formal modelling tool. The part of pacemaker's core functionality – a number of interdependent cyclic timing constraints – is implemented following Sarshogh's patterns for discrete timing [1]. We aim to improve modelling techniques, identify the applicability and constraints of used approaches, provide workarounds for identified limitations and suggest new prototype patterns.

Sarshogh defines three types of timing constraints: deadline (Figure 1) – response event must occur within time t of trigger event occurring; delay (Figure 2) – response event cannot occur within time t of trigger event occurring; expiry (Figure 3) – response event cannot occur after time t of trigger event occurring.

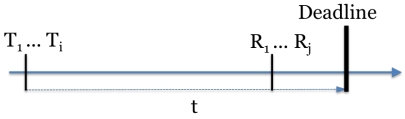

Figure 1: Deadline

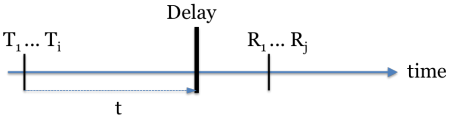

Figure 2: Delay

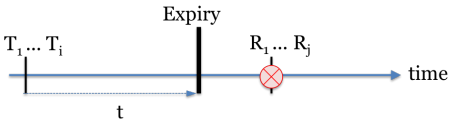

Figure 3: Expiry

In this paper we provide a prototype pattern solution, inspired by limitations found in the pacemaker case study. We extend Sarshogh's work by introducing a redefined timing notation (1), a concept of interval and interrupt event.

We define *interval* as timing entity that has timing constraints as its properties and can be manipulated by events.

$$Interval(\{T_1 ... T_i\}, \{R_1 ... R_j\}, \{I_1 ... I_k\}, \{TC_1(t_1)[, TC_2(t_2)]\}) \quad (1)$$

The interval is manipulated by three kinds of events: one of a set of trigger events $T \in \{T_i ... T_i\}$ initiates the interval; a response event $R \in \{R_i ... R_j\}$ terminates the interval; an interrupt event $I \in \{I_i ... I_i\}$ interrupts the interval.

[*] gs6g10@ecs.soton.ac.uk
[†] mrp@ecs.soton.ac.uk
[‡] ra3@ecs.soton.ac.uk

The interval must have either one or two timing constraints $TC_x(t_x)$, where t is the duration of the associated timing constraint and "[ ]" denotes optional TC.

As a simple example, consider a Lower Rate Interval (2). LRI is the longest time interval that is allowed between consecutive heart contractions. The interval is triggered by either intrinsic or artificial electrical stimulus $\{sense, pace\}$. It must be responded by the pacemaker stimulus $\{pace\}$ within $Deadline(t_2)$ time, but no earlier than $Delay(t_1)$. If intrinsic heart activity $\{sense\}$ occurs, interval is interrupted.

$$LRI(\{sense, pace\}, \{pace\}, \{sense\}, \{Delay(t_1), Deadline(t_2)\}) \quad (2)$$

In contrast to the original Sarshogh's patterns, our prototype is independent of event sequencing, therefore can be applied on any Event-B model. The prototype pattern has a modular design: a standard template code of T, R and I events is instantiated and injected into the target Event-B model as invariants, guards and actions to implement the timing constraints. The template is unaffected by target Event-B model contents. Supported event overloading enables event to serve as T and R or I for multiple intervals at the same time. Proof obligations are automatically discharged with the help of external provers.

Our future plan is to verify that our prototype supports Sarshogh's refinement and decomposition patterns; consider variable timing constraint duration t; perform more case studies; develop a plugin for Event-B code generation (Figure 4); add visualisation support for iUML plug-in and investigate – Event-B timing to code generation.

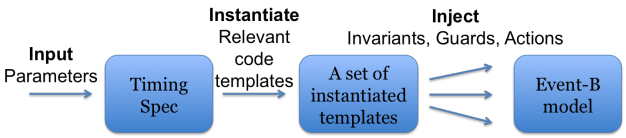

Figure 4: The workflow of the plug-in to generate Event-B timing code

Bibliography

[1] Mohammad Reza Sarshogh. *Extending Event-B with Discrete Timing Properties*. PhD thesis, 2013.

[2] Colin Snook and Michael Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, January 2006.

From Untimed Specification to Cycle-Accurate Implementation - Cyber-Physical System Model Refinement with Event-B

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

April 9, 2014

Abstract

Cyber-physical systems (CPS) are integrations of computing and physical mechanisms engineered to provide physical services including transportation, energy distribution, manufacturing, medical care and management of critical infrastructure. Such hybrid systems, although amenable to Event-B formal proof, must also be verified in a mixed continuous/discrete simulation environment such as that defined in the Functional Mock-up Interface (FMI) Standard. A major challenge that CPS present to systems modelling is that a well-developed notion of time needs to be introduced [Lee and Seshia, 2011], and often this is necessary quite early in the model refinement process. On the other hand, introducing a too detailed representation of time at the abstract level can complicate and constrain the model unnecessarily.

In this work, we present a methodical approach to timing in CPS modelling with Event-B which allows the notions of *synchronisation* and *communication* to be introduced in a lightweight manner at the higher levels of abstraction to target specific timing issues and then subsequently refined to the point that a concrete, synchronous, cycle-accurate model of the system controller can be developed.

The approach presented is based on the Synchronous Calculus of Communicating Systems (SCCS), of which asynchronous systems can be considered a sub-class [Milner, 1983]. Although CPS can often be modelled synchronously, this approach does not preclude the modelling of asynchronous behaviour.

In the early stages of CPS specification refinement with Event-B, a simple, untimed model of the system is usually appropriate. As refinement proceeds and a component-based view of the system emerges, it can be necessary to introduce the notion of *synchronisation*, especially when modelling the safety aspects of the system. System Theoretic Process Analysis (STPA) [Leveson, 2012] is a technique for Hazard Analysis which can be applied at an early stage of CPS development. In an STPA-based development of a CPS controller, the controller itself has a process

model of the system that it is controlling. Typically, the controller issues commands on its outputs, *waits* and then checks the values on its inputs against the internal process model to verify that the state of the process that it is controlling is consistent with the state that the controller expects. If it is not consistent, the controller has detected a hazard and can issue safe commands to mitigate that hazard. To describe this behaviour correctly, it is necessary to model the synchronisation between the controller and its environment at the point that the controller *waits*. In our approach, we only introduce sufficient synchronisation to effect the *rendezvous* between controller and environment.

As the CPS model is further refined and a more concrete component view of the system emerges, it is necessary to ensure that the synchronisation and communication between multiple components is more precisely modelled to prevent *race conditions*. The Event-B events associated with a particular process *Pn* are encapsulated between an entry point event *PnEvaluate* and a suspending event *PnWait*. At each synchronous tick of the clock, this set of events is evaluated in turn for each process before a *tick* event, *Update*, advances time. The inter-process *communication* of data values is modelled in such a way that the new value is only available at the next *tick*, thereby avoiding *race*. At this stage the *tick* is still just an abstraction of time which can subsequently be refined to represent the clock period of the controller. The behaviour of each process *Pn* between *PnEvaluate* and *PnWait* can also be abstract and non-deterministic. However, with this synchronisation and communication mechanism in place it is feasible to execute the abstract component models in a hybrid, continuous/discrete co-simulation at an early stage of CPS development.

Further refinement of the CPS controller component process introduces the ordering and determinism necessary for implementation, while the components that model the controller environment can remain non-deterministic to model potential hazard scenarios. At each stage of the refinement, formal proof, model checking and simulation-based testing can be used to verify the development. The final, concrete controller model has a synchronous, cycle-accurate representation which can be mapped directly to an implementation in a language such as VHDL, C, Bluespec or Esterel.

This work is funded by the FP7 project ADVANCE(287563), Advanced Design and Verification Environment for Cyber-Physical System Engineering (*http://www.advance-ict.eu/*).

References

Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach.* Lee & Seshia, 2011.

N.G. Leveson. *Engineering a safer world: Systems thinking applied to safety.* MIT Press (MA), 2012.

Robin Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

Event-B for Safety Analysis of Critical Systems

Matthias Güdemann and Marielle Petit-Doche*

{matthias.gudemann, marielle.petit-doche}@systerel.fr
Systerel — Les portes de l'Arbois, bâtiment A — 1090, rue Descartes
13857 Aix-en-Provence CEDEX 3, France

1 Introduction

In safety-critical domains, it is of very high importance to (i) increase the safety of a developed system and to (ii) provide convincing evidence to certification authorities that adequate means to construct a safe system have been taken.

Empirical research showed that many, if not most, of the major errors already originate from the system design phase and their early elimination therefore would be much more cost-effective [4]. The Event-B language [1] and its associated modeling approach aim at achieving this by providing means to formally prove abstract system-level models wrt. specifications.

For the moment however, despite some first approaches [3,2], formal methods are rarely involved on safety activities in the railway industry. We propose to extend the use of Event-B to support such activities; we show how Event-B supports safety analysis early in the development of critical systems, using formally verified and validated models of formalized safety requirements. The formal model provides strong feedback to the safety analysis about how safety is ensured. We apply the proposed approach to the European Train Control System (ETCS) within the context of the openETCS project.

2 Formal System Analysis with Event-B

Within safety activities, we propose first to construct a formal model of the system which correctly implements the high-level functional specification, e.g., its intended observable behavior. This model is augmented with formalized safety requirements, which are implemented in the model, traced in a requirements management tool and their correctness is proven. Afterwards, we use model simulation in order to validate the *intended* behavior of the system and of the risk mitigation measures. The advantages of such a formal model is (i) a mathematical reasoning associated to the safety analysis, (ii) an unambiguous model, which can be simulated, (iii) a model that is reusable and easy to update according to some maintenance process, (iv) a model that focuses on the analysis of the core functionality and (v) a model which provides the formal properties to validate on the vital software (via formal proof on the model or via functional test).

* This work was funded by the "Direction Générale de la compétitivité, de l'industrie et des services" (DGCIS) (Grant No. 112930309) within the ITEA2 project openETCS.

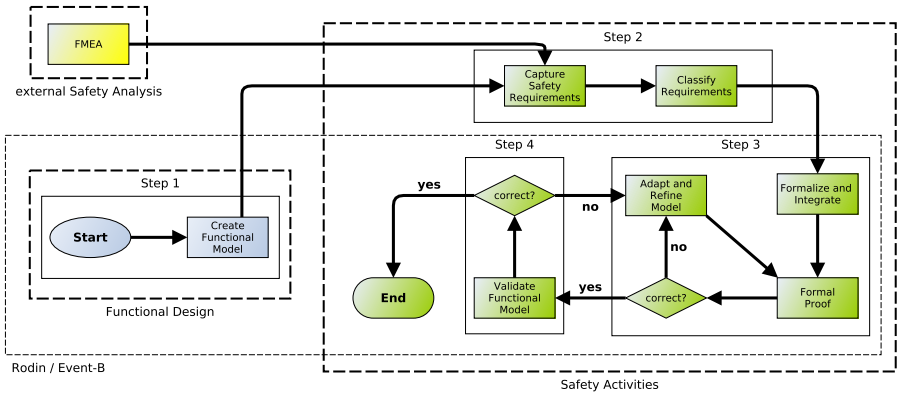

Fig. 1: Event-B in Process for Safety analysis

Fig. 1 details the proposed process to support safety activities. The first step on the left side describes preparatory work which is common to all design activities. It represents the initial construction of a functional model whose correctness is shown wrt. the functional requirements of the specification. The right hand side details the necessary steps for safety activities.

The second step consists of taking the results from a preceding external, informal safety analysis, in this case the failure mode and effects analysis (FMEA), and to make these results available to formal analysis. In general, these safety requirements will be of different levels of abstraction. Low level requirements often describe implementation details, e.g., redundant calculation chains or programming diversity, which are not captured at the system level. In an analysis based on Event-B, only system / high level requirements are applicable.

The third step consists of an iterative process. First, one formalizes the safety requirements (often as invariants and guards, buts other methods are possible). Then one tries to formally verify the safety properties on the system. If this is successful, the functional system model already verifies the safety properties and one can proceed. If the safety requirements do not hold, the model is adapted in order to correctly integrate them. This is done by creating a refinement from the functional system model and by using the feedback from the unsuccessful proof attempts as insight. These insights represent potential non-safe system behavior of the system and are important test cases for the later implementation phase.

The fourth step finally validates that the correct functioning of the system is preserved even after the integration of the safety requirements. This consists of animating the system using a model animation plug-in, and allows to analyze the effects of sequences of events. This facilitates the validation of the possibility to execute the intended use cases of the model and the observation that safety is ensured in the intended way. It is also possible to detect problems within the safety requirements using this process. In this case, there is feedback from the fourth step to the external safety analysis (not represented in Fig. 1).

3 Conclusion

Using the Event-B approach and the Rodin tool provides several benefits for supporting safety activities:

- it constructs a strong link between the formalized specification and the formal system model. The process of formalizing a specification helps to detect ambiguously formulated requirements, and proving the specification wrt. a functional system model provides insight into the completeness and coverage of the specification. These aspects allow for important constructive feedback which enhances the specification and eliminates errors already in early phases of system development. Writing a formal model associated with a safety analysis, allows to highlight the key elements of the system related to safety and to elicit formal properties to enhance the safety analysis. These properties can be directly reused during the design to validate the software, e.g., in the case where the software is formally designed with the B method, by proving them on the correct-by-construction model.
- Secondly, the possibility to simulate the formal model, and to observe its state at each time-step allows for validation of correct implementation of the risk mitigation of the safety requirements. This increases the confidence in the arguments of the safety case which explain the identified hazard causes and the intended risk mitigation techniques. As this is an important document for the certification of safety-critical system, augmenting and strengthening its arguments is therefore an important task in the development cycle of critical systems.
- Thirdly, the integration of Rodin into the Eclipse platform provides collaboration possibilities with third party plug-ins. One example is the close integration of the ProR requirements management tool, which allows to link Event-B artifacts requirement documents in the standardized ReqIf format, supporting transparent requirements tracing and interoperability with other tools. Thus fully open-source tool support for the early stage of the system safety analysis is available.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. 1st edn. Cambridge University Press, New York, NY, USA (2010)
2. Prokhorova, Y., Troubitsyna, E.: Linking modelling in event-b with safety cases. In: Proceedings of the 4th International Conference on Software Engineering for Resilient Systems. SERENE'12, Berlin, Heidelberg, Springer-Verlag (2012) 47–62
3. Prokhorova, Y., Troubitsyna, E., Laibinis, L.: Supporting Formal Modelling in Event-B with Safety Cases. In: Proceedings of the 4^{th} Rodin User and Developer Workshop. TUCS Lecture Notes (2013)
4. Westland, J.C.: The cost of errors in software development: Evidence from industry. Journal of Systems and Software **62**(1) (2002)

MODELLING OF SYSTEMS OF SYSTEMS
AN EVENT-B PERSPECTIVE OF A VDM PROJECT

STEFAN HALLERSTEDE AND KLAUS KRISTENSEN AND PETER GORM LARSEN

1. Introduction

A *system of systems* (SoS) is a system that is it self composed of systems, called *constituent systems*. Modelling of systems of systems poses some notational and methodological challenges on the modelling approach taken. In general, SoS models need to take into account:

- stakeholders and confidentiality,
- distribution and communication,
- continual evolution,
- dynamicity and adaptation.

The difficulty of modelling SoS stems from having to address all of these together. In the COMPASS project a combination of VDM [1] and CSP [3] called CML [4] has been used as SoS modelling notation and an accompanying SoS modelling methodology devised. A surprising result of the work is that in oder to deal with the complexity introduced by the modelling challenges only a reduced subset of VDM and CSP is used. This reduced subset appears to be easily transferable to Event-B using a simple CSP-based communication mechanism.

The purpose of this paper is to show how the results we have attained can be interpreted in Event-B and suggest modelling styles and extensions that would make Event-B an SoS modelling notation.

Some of the results are rather suprising in that we had expected that most problems would be amenable to "syntactic solutions": with the right syntax and expressive enough modelling notation all challenges can be met. However, this turns out to be wrong. And the answer to why this is the case lies in the place such models take in the life-cycle of large scale-engineering projects and the kinds of collaborations involved stakeholders may engage in.

2. The Challenges and Our Solutions

Stakeholders and Confidentiality. At first it appears that different parts of models could be supplied with information concerning stakeholder ownership and confidentiality. As a result of such an annotated model different stakeholders would be given different "views" of the model in some cases only getting very

Date: 8 April 2014.

abstract views. This approach that the stakeholders are willingly collaborating to produce an SoS with a set of emerging behaviours reaching beyond behaviours attainable by the constituent systems. However, in practice *commercial interestes* get in the way of producing and sharing such models and there is no syntactical cure for that. Each stakeholder will usually have their own model and maintain information about stakeholders classifying them in categories such as "collaborative" or "hostile". Such information can be used to determine which emergent behaviours can be achieved and which architecture can facilitate this. As a consequence, the model will contain results of a market analysis and future projections. It is unlikely that commercial enterprises will share such information even with contractors. The models are useful in a very different way than what we expected.

Distribution and Communication. In order to model a distributed system it would appear reasonable to modularise a model and let the modules communicate by way of CSP channels. In practice, a decision has to be made either for a CSP model containing a little VDM or a VDM model containing a little CSP. The second alternative bears advantages, in particular, with respect to the two points below dealing with evolution and dynamicity. The state-based model using VDM and a little CSP is more flexible.

We also found that using CSP meant introducing many channels and having to cope with a multitude of channels names and renamings. Even in rather small-sized SoS models with only a few constituent systems this turned out to be a problem to the level that understanding the model became difficult.

Finally, an important insight is not to modularise and to introduce a dedicated networking constituent system by means of which all other constituent systems communicate. This has reduced the number of CSP channels to two. All constituent systems are composed by interleaving and the result is synchronised with the network over the two channels. Using many channels it can get quite intricate to find the right combination of interleaving and synchronisation. Having only one module that contains all constituent systems as abstractions of their state also permits to represent the architecture of the SoS model with mathematical means: it can be analysed and it can be modified from within the model itself. A side effect of the very limited use of CSP is that it can be used for a different purpose: we can now introduce dedicated channels for model testing where CSP expressions are used to specify expected behaviour that can be verified against the SoS model.

Continual Evolution. SoS evolve continually. Their development can never be considered finished. Much of this takes place on the level of constituent systems. As a consequence many basic assumptions that can be made in a closed system about its components do not apply. On a low level constituent systems have some differences in the data-types and communication protocols they use. And we also

have to say which unexpected and unwanted behaviour can arise by connecting them. This is sometimes referred to as *negative emergent behaviour*. On a higher level one has to deal with varying services being offered and requested. This makes predicting or even verifying the actual behaviour of SoS difficult. What one can do is to analyse certain configurations of constituent systems that fulfil known specifications and then derive information about on which level the SoS perform on a scale such as: none, degraded, full. In general this scale will be finer and more tree shaped. And more than one model needs to be maintained to deal with new and changing constituent systems and their behaviour.

Dynamicity and Adaptation. When connecting many constituent systems using CSP channels it would appear "obvious" to declare CSP channels for all the different constituent systems and protocol stages. However, in an SoS constituent systems enter and leave the SoS continually. The approach relying heavily on CSP channels deals well with static architectures but gets very complicated when dealing with dynamic architectures. This is one of the reasons why the core model of the SoS only contains two channels: one to send a message to a network and one to receive from the network. Constituent have a number, their ID, in the SoS that are maintained in a data structure representing the architecture. Numbers can be easily generated and the data structure easily manipulated. Because there is only one module this is not difficult to realise.

3. Discussion

During the modelling of SoS there was a tendency to start with powerful constructs and subsequently to reduce them to a set of quite basic ones. A lot of the modelling power that we thought was needed is not used in this way. It turns out that for writing an SoS model we can live with VDM-SL, a subset of VDM, very few CSP concepts and only two CSP channels. Only for model testing we use some more CSP channels but still the same subset of CSP constructs. In the VDM-SL parts preconditions are used in a pattern that corresponds just to guards in Event-B. This is why we think that most of our work and results apply to Event-B (with those concepts of CSP added).

We belive that the kind of model that we produced will also make the modelling of faults easier. Problems such as message loss are easily incorporated into the network constituent system and apply to communications in the SoS. Compared to modifying many CSP expressions in many constituent systems the effort is negligible. Had we a sized-down concept of inheritance for VDM-SL we could also use the well-established VDM approach to fault modelling [2].

Acknowledgement. This work is supported by EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829).

For more information see `http://www.compass-research.eu`.

References

[1] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[2] Ken Pierce, John Fitzgerald, and Carl Gamble. Modelling faults and fault tolerance mechanisms in a paper pinch co- model. In *Proceedings of the ERCIM/EWICS/Cyber-physical Systems Workshop at SafeComp 2011, Naples, Italy (to appear)*. ERCIM, September 2011.

[3] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[4] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*. IEEE, July 2012.