Rodin User and Developer Workshop

27-29 February 2012
Fontainebleau, France

Workshop Organisers

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

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.

The first Rodin User and Developer Workshop was held in July 2009 at the University of Southampton while the second took place at the University of Düsseldorf in September 21-23, 2010. The 2012 workshop is part of the DEPLOY Federated Event hosted by the LACL laboratory at IUT Sénart-Fontainebleau.

While much of the development and use of Rodin takes place within the EU FP7 DEPLOY Project, there is a growing group of users and plug-in developers outside of DEPLOY. The purpose of this workshop is 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 provides an opportunity to share tool experiences and to gain an understanding of on-going tool developments. For plug-in developers the workshop provides an opportunity to showcase their tools and to achieve better coordination of tool development effort.

Deploy Federated Event

Rodin Developer Tutorial

Programme

Monday 27th February

Time	
08:30 – 09:00	***Coffee***
	Session 1: *Using the Rodin Theory Plug-in* - Issam Maamria (1h) *Using the Rodin prover API to connect external provers* *SMT Solver Integration* - Systerel (30 min) *Isabelle/HOL integration* - Matthias Schmalz (30 min)
11:00 – 11:15	***Break***
	Session 2: *Creating and using custom/parameterized proof tactics* - Nicolas Beauger, Jean-Raymond Abrial (30 min) *Integrating plug-ins with ProB* - Jens Bendisposto (30 min)
12:15 – 14:00	***Lunch***
	Master Class session 1 (1h30) in 4 Rooms: Room A : *Developping Theories* - Issam Maamria Room B : *Using custom/parameterized proof tactics* - Jean-Raymond Abrial, Nicolas Beauger, Thomas Muller Room C : *Parameterizing Isabelle with theories* - Matthias Schmalz Room D : *Developing for ProB* - Jens Bendisposto
15:30 – 16:00	***Coffee***
	Master Class session 2 (1h30) in 4 Rooms: Room A : *Developping Theories* - Issam Maamria Room B : *Using custom/parameterized proof tactics* - Jean-Raymond Abrial, Nicolas Beauger, Thomas Muller Room C : *Parameterizing Isabelle with theories* - Matthias Schmalz Room D : *Developing for ProB* - Jens Bendisposto

Deploy Federated Event

Rodin User & Developer Workshop Programme

Tuesday 28th February

Day One

Time	
08.30 – 09.00	***Coffee***
09.00 – 10.30	***Session 1:*** Welcome *SafeCap Modelling Environment* - Alexei Iliasov *Verification of a Railway Interlocking UML Model Translation to UML-B* - Gintautas Sulskus & Colin Snook *Component Reification in System Modelling* - Jens Bendisposto & Stefan Hallerstede
10.30 – 11.00	***Break:***
11.00 – 12.30	***Session 2:*** *Code Generation Update* – Andrew Edmunds C.J.Lovell, R. Silva, I.Maamria & M.Butler *Ensuring Extensibility with Code Generation* – Chris Lovell, A. Edmunds, R.Silva, I. Maamria & M. Butler *Towards a Certifying Code Generator for Rodin* – Alexei Iliasov *Generating Executable Simulations from Event-B Specifications* – Faquing Yan, Jean-Pierre Jacquot, and Jeanine Souquières
12.30 – 13.30	***Lunch:***
13.30 – 15.30	***Session 3:*** *Fault Tolerance Views* – Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky *Use of Rodin in FDIR Architechture for Autonomous Systems* – Jean-Charles Chaudemar *Pattern for Modelling Fault Tolerant in Event-B* – Michael Poppleton & Gintautas Sulskus *Extending Event-B & Rodin with Discrete Timing Properties* – Reza Sarshogh & Michael Butler
15.30 – 16.00	***Coffee:***
16.00 – 17.30	***Session 4:*** *A Framework for Diagrammatic Modelling Extensions in Rodin* – Vitaly Savicks & Colin Snook *Systematic Development for Embedded Systems Design using RRM Diagrams and UML-B* – Manoranjan Satpathy, Colin Snook, Silky Arora *CODA: A formal Event-B based Refinement Framework for High Integrity Embedded System Development* – John

	Colley, Michael Butler, Colin Snook, Neil Evans, Neil Grant & Helen Marshall *ADVANCE: Advanced Design & Verification Environment for Cyber-Physical Systems Engineering – The Multi-Simulation Framework* – John Colley & Michael Butler

Deploy Federated Event

Rodin User & Developer Workshop Programme

Wednesday 29th February
Day Two

Time	
08.30 – 09.00	***Coffee***
09.00 – 10.30	***Session 5:*** *Proving Consensus* - Jeremy Bryans & Alexei Iliasov *Verification and validation of BPEL processes - A proof and animation based approach* - Idir Ait-Sadoune, Yamin Ait-Ameur & Mickael Baron *Formal Specification of a Mobile Diabetes Management Application Using the Rodin Platform and Event-B* - Daniel Brown, Ian Bayley, Rachel Harrison, Clare Martin *Synthesis of Processor Instruction Sets from High-Level ISA Specifications* - Andrey Mokhov, Alexei Iliasov Danil Sokolov, Maxim Rykunov, Alex Yakovlev, Alexander Romanovsky
10.30 – 11.00	***Break:***
11.00 – 12.30	***Session 6:*** *An Event-B Plug-in for Creating Deadlock-Freeness Theorems* - Faquing Yang & Jean-Pierre Jacquot *The Theory plug-in and its Applications* - Issam Maamria & Michael Butler *Can rippling discover the missing lemmas for invariant proofs?* - Gudmund Grov, Yuhui Lin & Alan Bundy *Proof Hints for Event-B Models - Extended Abstract* - Thai Son Hoang
12.30 – 13.30	***Lunch:***
13.30 – 15.30	***Session 7:*** *Requirements Traceability between Textual Requirements and Event-B Using ProR* - Michael Jastram, Lukas Ladenberger & Michael Leuschel *Towards Relating Sub-Problems of a Control System to Sub-Models in Event-B* - Sanaz Yeganefard & Michael Butler *Lessons from Deployment* - Manuel Mazzara, Cliff Jones & Alexei Iliasov *Assessment of the Evolution of the RODIN Open Source platform* - Christophe Ponsard, Jean Christophe Deprez, Jacques Flamand
15.30 – 16.00	***Coffee:***

16.00 – 17.30	***Session 8:*** *A Rodin Plugin for automata learning and test generation for Event-B* – Ionut Dinca, Florentin Iplate, Laurentiu Mierla & Alin Stefanescu *VTG - Vulnerability Test cases Generator, a Plug-in for Rodin* - Aymerick Savary, Jean-Louis Lanet Marc Frappier & Tiana Razafindralambo *Visualisation of LTL Counterexamples with ProB* – Andriy Tolstoy, Daniel Plagge & Michael Leuschel.

SafeCap Modelling Environment

Alexei Iliasov

UK research project SafeCap (Overcoming railway capacity challenges without undermining rail network safety) [2] is investigating modelling techniques and tools for improving railway capacity whilst ensuring that safety standards are maintained. One of the goals of the SafeCap is the development of a user friendly modelling environment that would enable railway engineers to benefit from the verification and capacity assesment technology developed within the project without having to learn a formal notation. The SafeCap academic partners, Swansea University and Newcastle University, are currently working on complementary approaches to formalising safety and capacity aspects of railways in CSP (Swansea) and Event-B (Newcastle).

Early in the project, it was recognised that a succesful cooperation of the two teams depends on a common understanding of what constitues a railway schema and what are the metrics of capacity. To this end, we have defined the SafeCap domain specific language, strongly influenced by previous work done in Invensys and the work by Dines Bjørner [1]. The domain language defines the foundational concepts for the modelling of railway capacity. It attempts to capture, at a suitable level of abstraction, track topology, route and path definitions and signalling rules. Many concepts relevant in some way to capacity were intentionally left put this stage.

Eclipse is used as the basis for the modelling environment. The SafeCap DSL [2] is placed at the heart of the tool architecture and serves as a common exchange format for a variety of verification and editing tools. The editing tool is a graphical GMF-based editor that follows conventional approach to depicting railway schema. The environment is meant to be extensible and is itself realised as plug in to the core Eclipse; as such it may be easily integrated with any Eclipse-based tool such as the Rodin Platform.

One of the first extension tools available for the environment is an SMT solver-based verification plug in [3]. The plug in transforms a railways schema represented in the SafeCap DSL into an SMT theory file. The schema itself is used to assign values to constants so if any model is found by the solver it would have to be exactly the model defined in the DSL. In addition to these constants, there is a list of constraints. A solver would determine if the constant values satisfy all the constraints. If no model is found, the plug in tries to identify any one constraint that is not satisfied. The current procedure is based on a binary search within the set of constraints: a solver is given a varying set of constraints until the failed constraint is identified. The plug in offers push buttom verification - it returns a result within few seconds for valid schemas and within 20-30 second for schemas with an error. In principle, it may fail to provide any result due to resource constraints (time, memory) but we have not yet seen this in practice.

The number of constraints is quite large - more than fifty - and it is important that they are adequate (relate to the problem we are interested in) and mutually consistent. To derive such a set of constraints we have built a detailed Event-B model of trains travelling over tracks. The model defines concepts of line, route, segment, train, signal, point, crossover and control rules for points and signals. A number of key properties like the absence of train collisions and train derailments (points are never moved under a train) are formally proven. The construction of the model has resulted in a set of axioms charactersing what is understood to be a valid railway schemas. These axioms, essentially formalised environmental assumptions, are exactly the set of constraints used in the verification plug in.

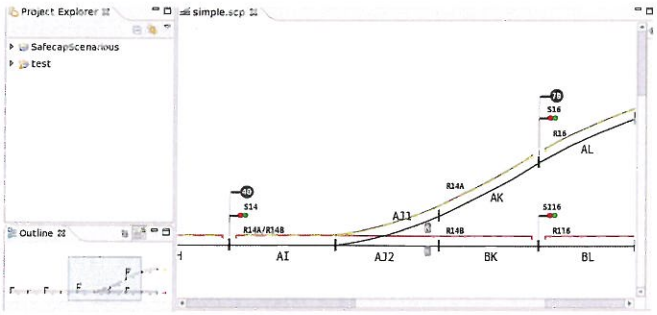


Fig. 1. Screenshot of the SafeCap modelling enviroment showing the schema editor.

References

[1] D. Bjørner. *Domain Engineering: Technology, Management, Research and Engineering*. JAIST Press, 2009.
[2] safecap.cs.ncl.ac.uk. SafecCap project web site. 2012.
[3] sf.net/projects/safecap. SafecCap platform web site. 2012.

Verification of a Railway Interlocking UML Model by Translation to UML-B

Gintautas Sulskus and Colin Snook
University of Southampton

The UML-B tool is a UML-like, graphical front-end for the Event-B formal notation and provides Class diagram and state-machine notations. Invariants may be added to the UML-B model to represent safety conditions. An automatic translator converts UML-B models into Event-B for verification. In the INESS project, UML-B is being used to verify a UML model of railway interlocking requirements.
The planned strategy is shown in Fig. 1. The UML model is transformed into UML-B to create an equivalent model. Safety requirements were to be provided by the railways and formalized as invariants in the UML-B model. The Rodin tools would then be used to verify that the safety invariants were obeyed.

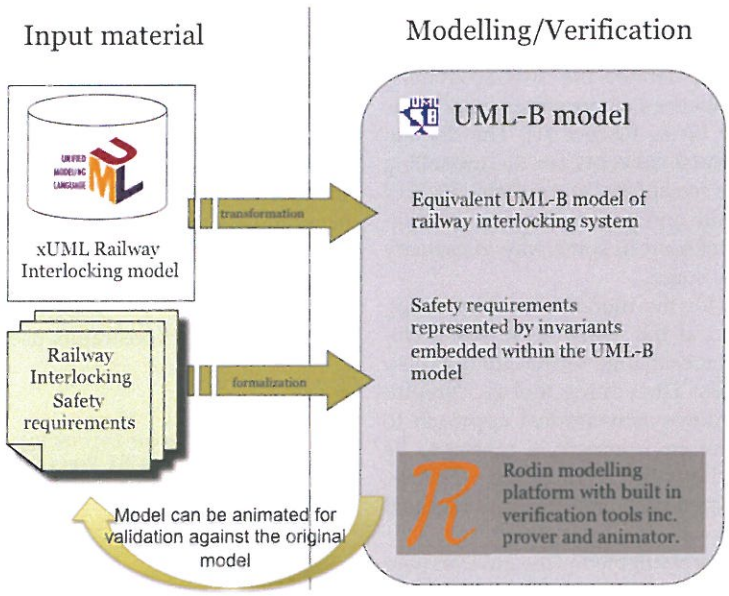


Figure 1. Strategy for Verification Using UML-B

The strategy has been followed as closely as possible but several problems were encountered.

1) The UML model was provided in iterations starting with low-level details of the system such as TVP sections rather than high-level concepts such as Routes. This is contrary to the refinement method where modelling starts with the most abstract, high-level concepts first and then proceeds to more detailed concepts.
2) Previous work on developing techniques for translating UML to UML-B was based on examples from the preceding project on the assumption that the final INESS UML model would use similar modelling techniques.

However, this was not the case. In particular, inheritance, which has been found useful when translating with refinement, was not used in these INESS UML models.

3) The safety requirements were provided in the form of statements about a specific example layout. An attempt was made to generalize the safety requirements but this was not satisfactory as the resulting requirements were contradictory. Consequently, invariants were derived from the INESS textual requirements document directly.

As has been noted, the iterations were not helpful to the development of the work since Iteration 4 is required for the first stage of work but is available last. However, it was discovered that even the earlier iterations contained the main structural elements of the higher level concepts expected in Iteration 4. The missing parts were the detailed behavioural aspects such as conditions for taking transitions. Hence, the UML-B models were constructed based partly on the UML models and partly on the INESS requirements document. The sequence of activities performed in relation to the Iterations received is as follows.

Iteration 1. Iteration 1 was investigated and a model of TVP sections and Diamond crossings was attempted. It became obvious that it would be difficult to refine this model to incorporate high-level concepts and this work was abandoned.

Iteration 2. Iteration 2 was used for the remainder of the work to date. The UML-B models of Routes, Signals and Track elements were based on this iteration. Behavioural details (UML-B guards on transitions) were invented and added to enable the invariants to be proved.

Iteration 3. Iteration 3 contains further details of behaviour that could now be added to the UML-B model to verify it.

Iteration 4. Iteration 4 is not yet available but is expected to contain further details of behaviour that could be added to the UML-B model to verify it.

UML class diagrams and state-machines were translated into equivalent UML-B diagrams preserving their pattern as far as possible so that the correspondence is clear. However, the UML-B model was arranged in refinement levels starting from a basic definition of routes and gradually introducing detail. Requirements relating to the modelled concepts were derived from the INESS requirements document and translated into invariants expressed in terms of the model elements. Appropriate invariants were placed at each refinement level. The Rodin tools were then used to generate proof obligations. The proof obligations could not be discharged because the UML model (Iteration 2) lacks necessary detail that determines when transitions are taken. Therefore transition guards (i.e. conditions deduced from the invariant) were added to the UML-B model to allow the proof obligation to be proved. Of course, adding guards to ensure the proof succeeds means that the value of the proof as a verification of the source UML model is diminished. However, it still has value in showing that the model is safe if appropriate conditions are added and expressing precisely and with surety what those appropriate conditions should be.

As a final stage, the corresponding conditions will be sought in the UML models (Iteration 3 and 4) and checked. It may also be possible to convert the conditions from the UML model into UML-B and re-prove to ensure that they are indeed equivalent.

Component Reification in Systems Modelling

Jens Bendisposto and Stefan Hallerstede

January 16, 2012

1 Introduction

When modelling concurrent or distributed systems in Event-B, we often obtain models where the structure of the connected components is specified by constants. Their behaviour is specified by the non-deterministic choice of event parameters for events that operate on shared variables. From a certain point on in a development we would rather develop components separately. The other –behaviourally identical– components should not concern us when we are focusing on one of them. We have to ask the following question: How can we systematically extract "real" components from such a model ? These components may still refer to shared variables. Events of these components should not refer to the constants specifying the structure. The non-deterministic choice between these components should not be via parameters. We say the components are *reified*. We need to address how the reified components get *reflected* into the original model. This reflection should indicate the constraints on how to connect the components.

2 A simple model

We base our discussion on the model of the leader election protocol of [1, Chapter 10] that chooses the maximum among a set of nodes arranged in a ring-shaped network. We have two constants N and n. They model the set of nodes and a successor relation for those nodes stated in context $ACTX$:

$$N \subseteq \mathbb{N}$$
$$n \in N \rightarrowtail\!\!\!\!\!\rightarrow N$$
$$\forall S \cdot n^{-1}[S] \subseteq S \land S \neq \varnothing \Rightarrow N \subseteq S$$

That is, the successor relation n is a bijection that does not map any proper subsets of the set of nodes N to itself.

In the implementation a variable $w \in N$ models the elected node; and a variable $a \in N \pfun N$ models a buffer of nodes. The conflation of the "real" nodes and their representation is part of the abstraction of the model. Reification resolves it. The protocol works as follows: initially each node stores its predecessor in its buffer. From then on, each node passes a node stored in its buffer on to its successor if the stored node is larger than itself and it drops the stored node if its smaller than itself. The maximum has been found when a node finds itself in its buffer. That node is the leader. The model has the following events:

$$\begin{array}{ll}
\mathsf{init} & \widehat{=}\ w :\in N \parallel a := n \\
\mathsf{event}\ \mathit{elect} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in \mathrm{dom}(a) \land x = a(x)\ \mathsf{then}\ w := x\ \mathsf{end} \\
\mathsf{event}\ \mathit{accept} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in \mathrm{dom}(a) \land a(x) < x\ \mathsf{then}\ a(x) := n(a(x))\ \mathsf{end} \\
\mathsf{event}\ \mathit{reject} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in \mathrm{dom}(a) \land x < a(x)\ \mathsf{then}\ a := \{x\} \lhd a\ \mathsf{end}
\end{array}$$

One result of the reification should be the removal of the references to n in init and event *accept*. Unfortunately, the non-deterministic choices of x in the events concern the contents of the buffers. The corresponding component is chosen implicitly by having a non-empty buffer: $x \in \mathrm{dom}(a)$. We

have to change the shape of the events so that the parameters specify the involved components. We refine the variable a by a variable $b \in N \leftrightarrow N$ such that $b = a^{-1}$. Our new model has the shape:

$$
\begin{array}{ll}
\mathsf{init} & \widehat{=}\ w :\in N \parallel b := n^{-1} \\
\mathsf{event}\ \mathit{elect} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in b[\{c\}] \wedge x = c\ \mathsf{then}\ w := x\ \mathsf{end} \\
\mathsf{event}\ \mathit{accept} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in b[\{c\}] \wedge c < x\ \mathsf{then}\ b := (b \setminus \{c \mapsto x\}) \cup \{n(c) \mapsto x\}\ \mathsf{end} \\
\mathsf{event}\ \mathit{reject} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in b[\{c\}] \wedge x < c\ \mathsf{then}\ b := b \setminus \{c \mapsto x\}\ \mathsf{end}
\end{array}
$$

Note that the guards now refer to the buffer of each component c and no longer to the entire buffer as in $x \in \mathrm{dom}(b^{-1})$. We have changed variable b to improve the readability of $x \in b[\{c\}]$ in particular.

We could now introduce a constant for each component, e.g. , $N = \{C1, C2, C2, C4, C5\}$ and expand each event into five events – one for each component, for instance,

$$
\begin{array}{l}
\mathsf{event}\ \mathit{acceptC1}\ \widehat{=} \\
\quad \mathsf{any}\ x\ \mathsf{where}\ x \in b[\{C1\}] \wedge C1 < x\ \mathsf{then}\ b := (b \setminus \{C1 \mapsto x\}) \cup \{nC1 \mapsto x\}\ \mathsf{end}
\end{array}
$$

Here we would define $nC1 = n(C1)$ in the context. The components are reified by replicating the events of the model. Next, we could introduce variables $b1$ to $b5$ for the buffers and begin to refine the components separately. Note that using this method we could also realize a system with only one component. We do not find this method satisfactory because of the replication involved. We prefer a method that would be inductive, splitting off one component a time.

3 A more inductive approach

Instead of introducing all components at once, we split one component $C1$ off. We can then refine the same component in different ways, arriving at the same number of components as above. We introduce two variables $b1$, the buffer of $C1$, and bN, the buffers of the remaining components such that

$$
\begin{array}{l}
bN = \{C1\} \lhd b \\
b = (\{C1\} \times b1) \cup bN
\end{array}
$$

The events have now the shape:

$$
\begin{array}{ll}
\mathsf{init} & \widehat{=}\ w :\in N \parallel b1 := P1 \parallel bN := \{C1\} \lhd (n^{-1}) \\
\mathsf{event}\ \mathit{elect1} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in b1 \wedge x = C1\ \mathsf{then}\ w := x\ \mathsf{end} \\
\mathsf{event}\ \mathit{accept1} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in b1 \wedge C1 < x\ \mathsf{then}\ b1 := b1 \setminus \{x\} \parallel bN \cup \{N1 \mapsto x\}\ \mathsf{end} \\
\mathsf{event}\ \mathit{reject1} & \widehat{=}\ \mathsf{any}\ x\ \mathsf{where}\ x \in b1 \wedge x < C1\ \mathsf{then}\ b1 := b1 \setminus \{x\}\ \mathsf{end} \\
\mathsf{event}\ \mathit{electN} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in bN[\{c\}] \wedge x = c\ \mathsf{then}\ w := x\ \mathsf{end} \\
\mathsf{event}\ \mathit{acceptN} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in bN[\{c\}] \wedge c < x\ \mathsf{then} \\
\multicolumn{2}{r}{b1 := b1 \cup \{n(c) \mapsto x\}[\{C1\}] \parallel bN := (bN \setminus \{c \mapsto x\}) \cup \{C1\} \lhd \{n(c) \mapsto x\}\ \mathsf{end}} \\
\mathsf{event}\ \mathit{rejectN} & \widehat{=}\ \mathsf{any}\ c\ x\ \mathsf{where}\ x \in bN[\{c\}] \wedge x < c\ \mathsf{then}\ bN := bN \setminus \{c \mapsto x\}\ \mathsf{end}
\end{array}
$$

Using this model we can continue the development as if we had only two components $C1$ and the "rest". We can refine *accept*1 and *acceptN* to allow the model to be decomposed as in [2]. (The initialization is already disentangled.) In order to obtain this model we have declared the following axioms in the corresponding context $CCTX$:

$C1 \in N$	Component $C1$ is a node in the network.
$P1 = n^{-1}(C1) \wedge N1 = n(C1)$	The information how to connect $C1$
$N \setminus \{C1\} \neq \varnothing$	There is at least one other component.

Hier sollten wir wahrscheinlich etwas ausf¨hrlicher sein.

Introducing $C1$ has forced us to spell out all constraints that it must satisfy. In particular, it requires knowledge of it successor and its predecessor. This was not so obvious in the first model.

The constraint $N \setminus \{C1\} \neq \varnothing$ is necessary because our method yields at least two components, $C1$ and another one that is represented by the events $electN$, $acceptN$ and $rejectN$. We can no longer produce a system with only one component. To produce such a system, we would have to let $N = \{C1\}$ and proceed as in the first approach briefly outlined at the end of Section 2.

4 Questions

We have presented a technique that permits reification of components in Event-B models in a simple way. The method makes constraints concerning each component explicit. This makes it easier to produce the necessary documentation: The user has to be informed how to set up the different components. We have avoided complicated solutions to our problem. For instance, we could have developed a method for instantiating "array variables" of the form $v \in A \to B$. But by following such approaches we would not be able to reason about the emerging components as we have done above. We have not formulated a theory of array instantiation. So we do not know whether it would work at all. But the latter argument convinces us that we should not try.

It would be interesting to know whether one could use the approach "properly inductive". Having proved a couple of components $D1$ to $D5$ correct implementation of $C1$, can we conclude that their composition is also correct directly from our inductive model? We think the solution will be related to how to deal with context $CCTX$, removing more and more references to n but proving that the axioms of $ACTX$ are satisfied.

References

[1] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[2] T. S. Hoang and J.-R. Abrial. Event-B decomposition for parallel programs. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2010.

Tooling: Code Generation Update

A. Edmunds, C. J. Lovell, R. Silva, I. Maamria and M. Butler

The Event-B method, and supporting tools have been developed during the DEPLOY project. A number of the industrial partners, are interested in the formal development of multi-tasking, embedded control systems. During the project, work has been undertaken to investigate automatic generation, from Event-B models, for these type of systems. Initially, we chose to translate to the Ada programming language, and use it as a basis for the abstractions used in our approach. The first version of the code generator supported translation to Ada, and the current version also has limited support for C.

We released a new version of the code generator on 14-12-2011. The presentation will give details of the the enhancements that have been made. We have made changes to the methodology, user interface, and tooling. The code generators have been completely re-written. The translators are now implemented using Java only. In our previous work we attempted to make use of the latest model-to-model transformation technology, available in the Epsilon tool set, but we decided to revert to Java since Epsilon lacked the debugging and productivity features of the Eclipse Java editor. We have also updated the documentation, including the Tasking Event-B Overview, and Tutorial materials.

We described our previous code generation feature as a demonstrator tool; chiefly a tool designed as a proof of concept, used by us to validate the approach. In this sense, the tool as it stands now, is the first prototype intended to be used by developers. However, we can use the demonstrator as a baseline, and describe the new features as follows:

- Tasking Event-B is now integrated with the Event-B explorer. It uses the extensibility mechanism of Event-B EMF (In the previous version it was a separate model).
- We have the ability to translate to C and Ada source code, and the source code is placed in appropriate files within the project.
- We use theories to define translations of the Event-B mathematical language (Theories for Ada and C are supplied).
- We use the theory plug-in as a mechanism for defining new data types , and the translations to target data types.
- The Tasking Event-B to Event-B translator is fully integrated. The previous tool generated a copy of the project, but this is no longer the case.
- The translator is extensible.
- The Rose Editor is used for editing the Tasking Event-B. A text-based editor is provided, using the Rose extension, for editing the TaskBody.
- The composed machine component is used to store event 'synchronizations'.
- Minimal use is made of the EMF tree editor in Rose.

A text-based task body editor was added, to minimize the amount of editing required with the EMF tree editor. The task body editor is associated with a parser-builder; after the text is entered in the editor the EMF representation is generated (by clicking a button) that is, assuming parsing is successful. If the parser detects an error, information about the parse error is displayed in an adjoining text box. When specifying events in the task body, there is no longer a need to specify two events involved in a

synchronization. The code generator automatically finds the corresponding event of a synchronization, based on the event name, and using the composed machine component. Composed machines are used to store event 'synchronizations', and are generated automatically during the decomposition process. This reduces the amount of typing in the TaskBody editor, since we no longer need to specify both local and remote (synchronizing) events. The new feature also overcomes the 'problem' that we previously experienced, with duplicate event names in a development, and event selection, when specifying the task body. The EMF tree editor in Rose is now only used minimally; to add annotations for Tasking, Shared and Environ Machines; typing annotations, and parameter direction information; and sensing/actuating annotations, where necessary. Further work is under way to integrate the code generation feature with the new Rodin editor.

The code generation approach is now extensible; in that, new target language constructs can be added using the Eclipse extension mechanism. The translation of target's mathematical language is now specified in the theory plug-in. This improves clarity since the the translation from source to target is achieved by specifying pattern matching rules. The theory plug-in is used to specify new data-types, and how they are implemented. Translated code is deposited in a directory in the appropriate files. An Ada project file is generated for use with AdaCore's GPS workbench. Eventually this could be enabled/disabled in a preferences dialog box. The Tasking Event-B to Event-B translator is now properly integrated. Control variable updates to the Event-B model are made in a similar way to the equivalent updates in the state-machine plug-in. The additional elements are added to the Event-B model and marked as 'generated'. This prevents users from manually modifying them, and allows them to be removed through a menu choice.

The DEPLOY project is funded by the European Commission ICT DEPLOY contract / grant number: 214158.

Ensuring Extensibility within Code Generation

C. J. Lovell, A. Edmunds, R. Silva, I. Maamria and M. Butler

School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK
cjl3@ecs.soton.ac.uk

Making the step from Event-B to code is a process that can be aided through automatic code generation. The code generation plug-in for Rodin is a new tool for translating Event-B models to concurrent programmes. However users of such a tool will likely require a diverse range of target languages and target platforms, for which we do not currently provide translations. Some of these languages may be subtly different to existing languages and only have modest differences between the translation rules, for example C and C++, whilst others may have more fundamental differences. As the translation from Event-B to executable code is non-trivial and to reduce the likelihood of error, we want to generalise as much of the translation as possible so that existing translation rules are re-used. Therefore significant effort is needed to ensure that such a translation tool is extensible to allow additional languages to be included with relative ease. Here we concentrate on translation from a previously defined intermediary language, called IL1, which Event-B translates to directly.

The intermediary language IL1 is an EMF metamodel representation of generic properties and functionality found in many programming languages. It has representations for key structural concepts such as variables, subroutines, function calls and parameters. The translation of predicates and expressions contained within the code are handled by a new extension to the theory plug-in, which allows translation rules to be developed for specific target languages within the Rodin environment. The generic nature of the intermediary language is designed to allow for a wide range of different target languages. Developers of new target languages are required to write translators in Java for the conversion from the EMF representation to the code of their target language. To do this we provide a central translation manager, that takes an IL1 model and automatically calls the appropriate translators for each element of the model, whilst also providing the link to the predicate and expression translators provided by the new theory plug-in. The developer registers their translators for the target language through an extension point, where currently there are 15 light-weight translators required for a new target language. To aid the developer, we provide abstract translators for each required element in the IL1 model that has to be translated. These translators perform the majority of the translation automatically, meaning that in most cases all the developer is required to do is format strings into the appropriate structure for their target language. For example in an branch statement, the developer would be required to write a method stating how a branch is defined and structured in their language, using a set of previously translated guard conditions and actions. Importantly, the flexibility remains for the developer to re-write any of the translations if the ones provided are not suitable. To test our approach, we have built translators for C, Ada and Java using the same underlying abstract translators.

Additionally we consider the case where a new language may be required that has only modest differences to an existing language. A good example of this is to consider the case where a different library may want to be used from one used in an existing translation. For instance in C, concurrency can be achieved through different mechanisms such as OpenMP or Pthreads. In this case it may be that all but the mechanism for handling a subroutine call are the same, meaning that the majority of the translation can occur using common translators, with separate translators for the different methods of handling a subroutine call. To allow for this we allow the developer to assign a core and specialisation language to each translator they build. In cases where a translator for the specialisation language does not exist, the translator will automatically defer to the default core language translator, if one exists. This means that default translators for a particular core langauge can be written for the majority of the translation, with specialisations being provided where differences occur. The core and specialisation of the language is also reflected in the theory translator, meaning that language theories are only required for the core languages, rather than for each individual specialisation.

Part of this research was carried out within the European Commission ICT DEPLOY contract / grant number: 214158.

Towards a certifying code generator for Rodin

Alexei Iliasov

A development process based on the application of formal notations and verifications techniques potentially delivers a system that is free from engineering defects. A typical formal development starts with a comprehensive requirements document, proceeds with a modelling stage where the requirements are formalised and transformed into implementable steps, and completes with the construction of a final product, e.g., a program or a hardware description. An automated code generator transforms models into runnable software quickly, consistently, reproducibly and with a lower rate of errors than is if coded manually. Most code generators, however, do not offer the guarantee of an error-free result. Commonly, a code generator is a fairly large program constructed informally and producing an output that is not (at least formally) traced to an input. This undermines the value of using formal methods in safety-critical domains. Industrial standards to the development of safety-critical systems, such as IEC 61508, require that for any tool used in a development there is a sufficiently strong justification. Such a justification could be an extensive prior experience with the tool or a formal certification by a relevant certification authority. As there are not many formal modelling toolsets that have been around for decades, the prior use is rarely an option for formal method adopters. These leaves just two opportunities: certify a code generator for the use in a given domain, or completely ignore the code generating activity in the safety case and verify the generated software as if it were constructed informally. The latter case leaves no reason for using formal modelling in the first place. In the former case, a code generator itself must be constructed accordingly to the relevant industrial standards for safety-critical softwarewhich means a higher cost, a longer development cycle and, possibly, tips the balance away from the use of formal modelling. In addition, certification requirements vary considerably between the certification bodies of differing nations and industries.

We propose an approach where instead of attempting to justify the use of a code generator a user places no trust whatsoever in the code generation stage but, through a code generator, obtains software that is certifiable without any further effort. The essence of the approach is in the transformation of a formal model into runnable software that is demonstratably correct in respect to a given set of verification criteria, coming from a requirements document. In this approach, all the correctness guarantees are embedded in the resultant program; intermediate formal models are disregarded for the purpose of product certification and the design and functioning of a code generator are deemed irrelevant. In our understanding, code generation is not an end of a development but rather the start of a new phase. It is unrealistic to attempt to formally specify every implementation detail such as OS level calls or user interface. Emitting code with desing by contract annotations [1] delivers a fair degree of rigour with the flexibility of code added by hand.

To evaluate the idea, we have implemented to a proof of concept tool for code generation from Event-B models. The tool, called B2H5, is available for evaluation together with a set of sample problems [5]. B2H5 works in the context of a single Event-B machine and, optionally, a Flow diagram. The former describes the possibles computation steps, variables and their types, and global invariant properties. The latter defines an algorithmic structure of a target program and may be used to express additional verification constraints. The output is a code annotated with a Hoare logic proof scheme. To this end, we have defined a custom Hoare logic with most of the inspiration from [4]. Proof annotations are automatically derived from event definitions and Flow diagram assertions. Event-B and Flow proof obligations are recorded as evidence of the satisfaction of side conditions of the inference rules of the logic. For instance, the Event-B invariant satisfaction proof obligation supplies a proof for a side condition in the global correctness rule of the Hoare logic. The tool is able to emit JML [1] code where non-deterministic statements (derived from non-deterministic actions or event parameters) are replaced with method calls. Such methods are annotated with pre- and post-conditions but have empty bodies. It is an obligation for a programmer to fill in the missing code; verifications tools would ensure that the method conditions are satisfied by the added code.

The related work may be loosely structured into approaches to verifying compilers and certifying compilers. A verifying compiler, for instance [2], implements a provably correct translation procedure for transforming a high-level programming language into machine code. The correctness is defined in terms of the observable behaviour of a program and, possibly, additional annotations. A certifying compiler generates a program together with a proof of its correctness. A notable example is the proof-carrying code technique [3]. Our approach belongs to this group.

References

[1] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7:212–232, June 2005.

[2] Thilo Gaul, Andreas Heberle, Wolf Zimmermann, and Wolfgang Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. 1999.

[3] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Proceedings of the Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997. ACM Press.

[4] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[5] Rodin platform plug-in. B2H5. 2011. Available at http://iliasov.org/b2h5/.

Generating Executable Simulations from Event-B Specifications

Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières

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

Abstract. This short paper presents a work-in-progress about the generation of executable programs from Event-B formal specifications to simulate the behavior of the model. Execution techniques such as animation of the formal text allow users or domain experts to be involved in the early stage of the development process. Unfortunately, they fail on some specifications which contains a high level of non-determinism, too abstract domains, or domains which lead to combinatorial explosions. Our strategy is to involve developers and users at three levels by providing annotations for the implementation of state variables, hand-coded functions for selected quantified expressions, and argument values during the execution.

The formal foundation underlying Event-B [1] can be easily thought out in an operational way. Several animators have been written which allow users to see how the state evolves with the firing of the events. Those tools have strong limitations on the class of specifications they can execute. In [3], we see how the class of executable specifications can be extended with the help of a few safe transformations. Yet, there are still texts on which animators fail. For those texts, we propose to use *simulation* rather than animation. The specification is translated into a program.

Animators fail because they "pick" values by enumerating their domain rather than compute them. By providing hints to the translator and hand-coded functions to the run-time, we can deal with three causes of blockage:

- too abstract domains can be replaced by concrete data-structures,
- uncomputable nested quantifications can sometimes be replaced by functions,
- free variables in events are classified into parameters, which are provided either interactively or by ad hoc algorithms, and local variables which can be computed.

The general translation strategy is to involve humans in the generation and the execution of the simulator. The involvement takes three forms: annotating the formal text to drive the translation, providing hand-coded functions for some expressions leading to combinatorial explosion, and providing values during execution.

Annotations are used to specify the type and nature of Event-B elements which are either too abstract (carrier sets for instance) or ambiguous (function vs array for instance).

Several works address the translation of Event-B models into executable programs, notably B2C [5] and EB2ALL [4]. The aim of these translators is to generate a correct

program, i.e., a program which preserve the invariant of the most abstract model. They are meant to be applied on a refined model where the state has been reified down to implementable data structures and the events are deterministic. So, they ignore non-deterministic constructs and quantification. Our aim is different. We are less interested in formal correctness than in the visualization of the behavior of the model. On the one hand, the generated simulator is not guaranteed to be an exact implementation of the model, but, on the other hand, we can translate a much broader class of models. This is consistent with the intended use of simulation in validation.

We follow a pragmatic approach to develop the translator. We use essentially two specifications of a platooning algorithm[2,6]. The first model, in 1 dimension, can be animated; hence, we use it as a reference. The second model, in 2 dimensions, cannot be animated but can be simulated; hence, we use it as a test bed. We proceed in development cycles along the sequence of refinements. A cycle has four steps. First, we program by hand a simulator to infer translation rules. Second, we formalize the rules. Third, we implement the rules as part of a Rodin plug-in. Last, we compare the output of the plug-in with the hand-coded simulator. This allows us to experiment with different translation strategies. At present, simulators for all refinements of the 1D model can be generated. The first refinements of the 2D model can also be generated.

Current efforts are focused on the non-deterministic substitutions and the management of parameters. Those issues are connected with the management of user's input during a simulation. The computation of the events' guards or the checking of the validity of user's provided values depends on when the input is performed during an execution cycle. Different strategies must be experimented to find a good trade-off between the volume of input requested from the user and the efficiency of the computation.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008). p. 8 pages. France (2008-06), http://hal.archives-ouvertes.fr/hal-00260577/en/
3. Mashkoor, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert'09). York, UK (2009)
4. Méry, D., Singh, N.: Automatic Code Generation from Event-B Models. In: Proc. 2011 Symposium on Information and Communication Technology, SoICT2011. ACM International Conference Proceeding Series, ACM, Hanoi (2011)
5. Wright, S.: Automatic Generation of C from Event-B. In: Workshop on Integration of Model-based Formal Methods and Tools (2009)
6. Yang, F., Jacquot, J.P.: Scaling Up with Event-B: A Case Study. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 438–452. Springer Berlin / Heidelberg (2011)

FAULT TOLERANCE VIEWS

I. LOPATKIN, A. ILIASOV, A. ROMANOVSKY
NEWCASTLE UNIVERSITY

The Fault Tolerance (FT) Views is a modelling environment for constructing fault tolerance features in a concise manner and formally linking them to Event-B models. It provides fault tolerance modelling facilities explicitly supporting the traceability of the FT and dependability requirements. The FT View is a special case of the Mode View, developed in our previous work on modelling modal systems [2]. It is essentially an application of the Mode View approach to fault tolerance. The FT View approach extends the Mode View with additional fault tolerance semantics, structural checks, and helps the modeller by offering reusable refinement templates. The details of the FT View approach can be found in [4].

A mode/FT view is a graph diagram developed alongside an Event-B model which contains modes and transitions along with additional information necessary for establishing a formal connection with the model. The two basic concepts of the Mode View are *mode* and *transition*. Mode is a general characterisation of a system behaviour. It describes the functionality of a system and the operating conditions under which the system provides this functionality. A system switches from one mode to another through a mode *transition*.

The FT View adds two types of transition specialisation: an *error* and a *recovery* transitions. Relative to the transition and its type, we differentiate the FT types of modes: we say that an error originates in a *normal mode* and leads to switching to a *degraded mode* or a *recovery mode*. The recovery transition leads from the recovery mode back to normal. The distinction between the degraded and recovery modes is that the recovery mode is obliged to terminate and pass control back to the mode from which the initiating error originated. Safe-stop is regarded as a special case of a degraded mode.

Diagrams are built in a step-wise manner, starting from the most primitive case and introducing details using our *detalisation* process. The FT Views development process is a chain of documents similar to Event-B models. FT diagrams are built by incrementally adding new modes, errors and recoveries using the provided templates, and proving the refinement relationship between each two consequent views. Description of the *detalisation* process and templates can be found in [4].

There is a formal relationship between a diagram and an Event-B model establishing that a model agrees with a diagram. The formalisation approach is based on a more general notion of formal modal systems [3]. There is a study on linking mode diagrams and Event-B [2] and some of the results are reused for the FT Views.

During the presentation, we will show the usage of the FT Views plug-in for Rodin. We will demonstrate on a case study the benefits that developers can gain for their modelling, and limitations of the approach.

References

[1] Mode/FT Views wiki page. http://wiki.event-b.org/index.php/Mode/FT_Views.

[2] F. L. Dotti, A. Iliasov, L. Ribeiro, and A. Romanovsky. Modal systems: Specification, refinement and realisation. In *ICFEM*, pages 601–619, 2009.

[3] A. Iliasov, A. Romanovsky, and F. L. Dotti. Structuring specifications with modes. *Latin-American Symposium on Dependable Computing*, pages 81–88, 2009.

[4] I. Lopatkin, A. Iliasov, and A. Romanovsky. On fault tolerance reuse during refinement. In *2nd International Workshop on Software Engineering for Resilient Systems, SERENE'10*, London, UK, April 2010. available as CS-TR-1188 at Newcastle University, UK.

Use of Rodin in FDIR Architectures for Autonomous Systems

Jean-Charles Chaudemar[1], Eric Bensana[2], and Christel Seguin[2]

[1] ISAE-DMIA, Toulouse, France
[2] ONERA-DCSD, Toulouse, France

Abstract. The contribution of this paper consists of a modeling approach based on Event-B for a layered architecture. This modeling approach aims at providing a framework of formal description for autonomous control systems. The proposed Event-B modeling is original because it takes into account exchanges and relations between architecture layers by means of refinement. Safety requirements are first specified with axioms, invariants and theorems, then validated by proof tools of the Rodin platform. The functional properties and the properties relating to fault tolerant mechanisms improve the relevance of the adopted Event-B modeling for safety analysis.

Keywords: formal methods, refinement, dependability, fault tolerant architectures.

1 Introduction

Complex critical systems are often structured in hierarchical layered architectures integrating dependable mechanisms [1]. The stringent design of such fault tolerant architectures requires the use of best suited methods. Therefore, we choose in this article a formal method called Event-B, which has been developed to correctly and gradually specify complex systems through a refinement mechanism [2]. Refinement is one incremental technique aiming at transforming an abstract model into a concrete model i.e. a model containing more details in its specification. The application of the Event-B method in our study relates mainly to the specification of nominal and abnormal behaviors in each layer of the architecture. Moreover, by means of a stringent specification of layer interactions we ensure the consistency of system behavior with a safety analysis.

2 Architecture modeling

The system abstract modeling constitutes a suitable means to master the complexity while being focused on some fundamental aspects of its architecture and its behavior.

A generic description of the architecture emphasizes component families. The families of components are closely related to the definition of the layers and are named *EQUIPMENT*, *FUNCTION* and *OPERATION*. They are characterized by the following attributes: *the operating status* enables to identify the current operating condition for a given component; *the health status* characterizes a monitoring of the operation quality; *the observation resources* enable to identify the produced or consumed data by some components. Some architecture characteristics evolve after execution of activities or services of the various layers. More concretely, it is considered that the execution of a service could be composed of the 4 kinds of following events: *Activation* enables to select a functionality associated with a layer component; *Acquisition* reinforces the links between various components of architecture by waiting for resources, above all useful data for the next step; *Execution* deals with activities relating to function. It could produce data to be exchanged.

The faulty behavior occurs when health status is modified for the various architecture components. The state variable which allows this monitoring is a relation *e_status* for the equipment layer, or *f_status* for the function layer. The value evolution of health status is described by failure events associated with the fault detection mechanism: *detect_err*[3] concerns the switch to erroneous status due to fault detection leading to erroneous output; *detect_lost*[3] expresses the loss of a component due to a failure. Another mechanism related to reconfiguration due to a fault (FDIR) is represented by an event *recover* which triggers the activation of redundant equipment component in the case of equipment and switches off lost component.

This layered architecture that integrates fault tolerant mechanisms is modeled by using Event-B formalism. Event-B enables to formally express some intrinsic properties relating to the components interaction.

3 Conclusion

The Event-B method supported by the Rodin platform is a framework of modeling adapted to our need for formal specification. Our modeling could also be applied to other control systems in mobile robotics or in space fields, for instance in the study of satellite FDIR. The interest of our modeling is to provide a generic common framework of formal specification of these systems for the safety analysis.

References

1. Alami, R., Chatila, R., Fleury, S., Ghallab, M., Ingrand, F.: An Architecture for Autonomy. International Journal of Robotics Research **17** (1998) 315–337
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press. (2010)

[3] A prefix *f_* is added to distinguish detection events related to function layer

Pattern for modelling fault tolerant systems in Event-B

Michael Poppleton, Gintautas Sulskus

School of Electronics and Computer Science

University of Southampton, Southampton, SO17 1BJ, UK

{mrp, gs6g10}@ecs.soton.ac.uk

Introduction

Formal methods are used for specification and verification of software and hardware systems. One class of systems interacts with the outside world through sensors and actuators and may include non determinism – hardware faults or external influence – making it difficult to model. This can compound the modelling of requirements, which can be in itself a major challenge.

Previous work [1] by Hayes, Jackson and Jones tries to tackle these issues by primarily focusing on deriving from the systems environment.

The aim of this work is methodological - we seek patterns in our case study development, which is based on [1], that may be applicable within this class of systems.

Development pattern

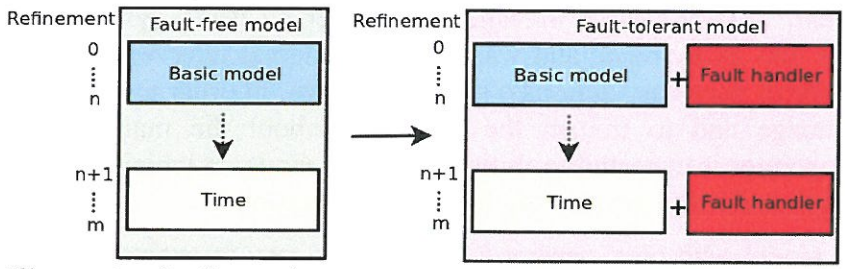


Illustration 1: General pattern structure

Our pattern is concerned with 3 distinct areas – basic modelling, time and fault handling (Illustration 1). *Basic model* deals with non-time requirements and is based on Butler's Cookbook [2] for control systems modelling. *Time module* introduces time infrastructure layer (clock, event delays and deadlines) and time related requirements via further model refinements. The *fault handler* is an extension of already modelled *fault-free* and *time* modules, meaning that this is not a refinement but rather additional code in already existing refinements. This fault handling can be corresponded as an requirements feature, being composed onto the basic development. This extension inevitably alters fault-free model due to necessity to modify/remove invariants which rely on monitored variables and to add FH mode's variable checks to fault-free event guards.

Basic model

A model of a real world system that may fail, should always include a Fault Management System [3]. Introducing FMS "hooks" in a fault-free model does not cause much overhead, however, in later stages this simplifies model extension with fault tolerance by reducing the extent

of possible perturbations (Illustration 2). FMS never detects any errors in the fault-free model and lacks fault handling events. However, its presence is recommended since the model should reflect the actual system configuration in both hardware and software levels.

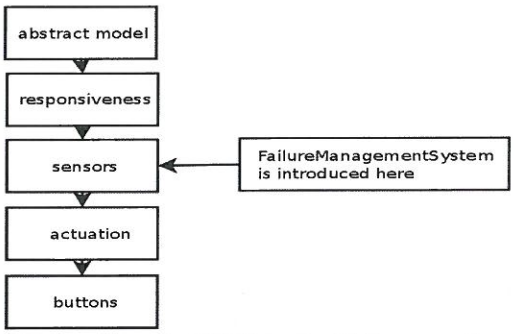


Illustration 2: FMS in fault-free mode

It is assumed, that fault-free model does not have any faults, and so it is allowed to make an assumption that environment behaves as expected. To fulfil such assumption, environment events and directly related (e.g. sensing) events are constrained by specifying strict guards (in this case, direct monitored variable usage is allowed) – $G_u \wedge G_t$, where:

- G_u - guards for fault-free execution, excluding time requirements – constrained event occurs only in a specific system state;
- G_t – time requirements – some systems may expect environmental changes in a limited time.

Fault handling

Fault handling is based on 4 different system modes, controlled by FMS (Illustration 3). Under normal conditions, when no faults are detected system operates in *normal mode* and performs intended operations – this is the only mode present in *fault*-free model. In case of a fault, system enters into *suspicious mode*. This mode allows system to tolerate faults (or particular fault) to a certain extent. In case system manages to recover by itself, system re-enters *normal mode*, otherwise it falls into *faulty mode* and waits for the external intervention. The latter mode's main purpose is to perform necessary actions e.g. to shut down, in order to prevent (further) system and environment damage and to inform the operator about the malfunction. After faults were eliminated, the operator is expected to initiate *recovery mode*, in which system tries to return into a state from which it could start performing its normal operation.

Fault-free model with only *normal* FMS mode is easy to extend with fault tolerance simply introducing FMS and related controller events for the rest 3 modes.

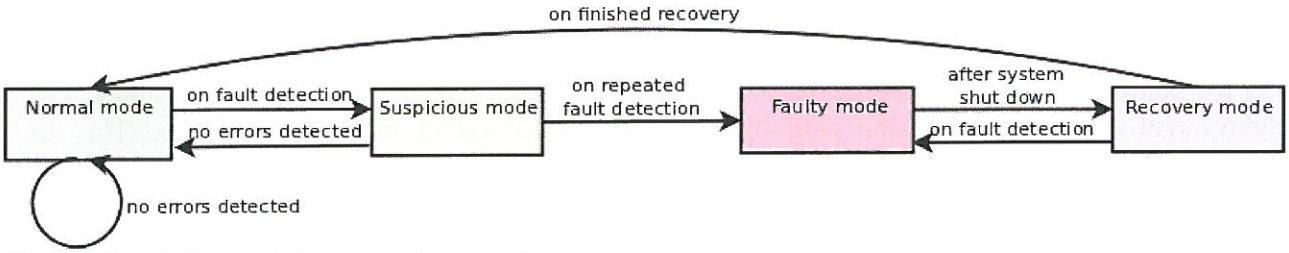


Illustration 3: Fault tolerant system modes

In the fault tolerant model, environment events become non-deterministic, thus additional events are introduced for each environment event and directly related (sensing) events from fault-tolerant model. These newly introduced events are expected to cause system fault and a deviation from a normal operation. To ensure this, events have guard block as $\neg G_u \vee \neg G_t$.

For each relevant events $(G_u \wedge G_t)_{\text{fault-free}} \wedge (\neg G_u \vee \neg G_t)_{\text{fault-tolerant}}$ results in a complete event guard space, meaning that event in fault tolerant model may be without guards.

References

[1] I. J. Hayes, M. A. Jackson and C. B. Jones, "Determining the Specification of a Control System Fromthat of Its Environment", FME 2003: Formal Methods. International Symposium of Formal Methods Europe, 2003
[2] M. Butler, "Towards a Cookbook for Modelling and Refinement of Control Problems", 2009
[3] D. Ilic, E. Troubitsyna, L. Laibinis, C. Snook, "Formal Development of Mechanisms for Tolerating Transient Faults", 2006

Extending Event-B and Rodin with Discrete Timing Properties

Reza Sarshogh, Michael Butler, University of Southampton

Timed Event-B

Our talk will outline some recent extensions to Event-B for expressing and verifying timing properties [1]. Modelling time-critical systems by using Event-B has been investigated in several studies. What distinguish our work, is categorizing timing constraints in three groups, introducing a systematic way of encoding each of them in an Event-B model, introducing patterns for refining timing constraints and proving satisfaction of abstract timing constraint by their concrete ones. In this way, the consistency of the timing properties in the system specification can be proved by using refinement feature of the language.

Time Properties

In order to explicitly represent timing properties we extend the Event-B syntax with constructs for deadlines, delays and expiries. Each of these timing properties places a discrete timing constraint between trigger events and their response events. A typical pattern is a trigger followed by one of several possible responses, thus each of our timing constructs specifies a constraint between a trigger event ***A*** and a set of response events $B_1..B_n$.

The syntax for each of these constructs is as follows:

- Deadline (A, {B_1,..,B_n}, t),

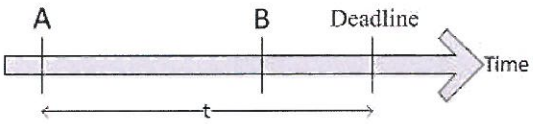


Figure 1: Timing diagram of Deadline constraint.

- Delay(A, {B_1,..,B_n}, t),

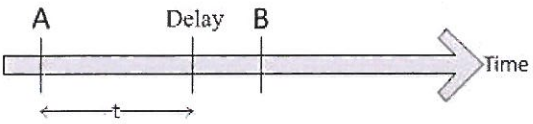


Figure 2: Timing diagram of Delay constraint.

- Expiry(A, {B_1,..,B_n}, t).

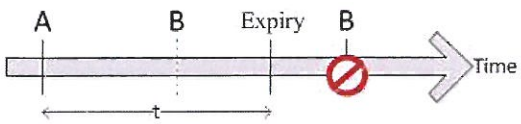


Figure 3:Timing diagram of Expiry constraint.

Deadline (A, {B_1..,B_n}, t) means that one of the response events (***B_1..B_n***) must occur within time ***t*** of trigger event (***A***) occurring. Figure 1 provides a diagrammatic illustration of the relationship between events ***A*** and ***B*** and the deadline. In the case of delay (Figure 2), all of the response events can only happen if delay period has been passed following an occurrence of the trigger event. Finally the expiry means that none of the response events can happen after the expiry period has been passed following an occurrence of the trigger event (Figure 3).
We give a semantics to our timing constructs by translating them into Event-B variables, invariants, guards and actions. In particular, these timed-Event-B elements constrain the order between trigger events, responses events and the time-progression event (***Tick_Tock***); the Event-B elements that a deadline constraint adds to a machine, will prevent the ***Tick_Tock*** event from occurring more than ***t*** times in between an occurrence of the trigger event and one of its response events' occurrence.

Rodin Plug-in

We have developed a plug-in to allow the modeller to specify the timing properties of a model by using the introduced annotations and generate the required guards, actions and invariants to encode them in the corresponding Event-B model automatically. So by having the tool, modeller need not be concerned with the semantic of timing properties.

We have developed some refinement patterns for mapping abstract timing constraints through Event-B refinements and in the future we plan to automate the application of these refinement patterns including automatic generation of the required gluing invariants to prove the refinements.

[1] Sarshogh, M. R. and Butler, M. (2011) *Specification and refinement of discrete timing properties in Event-B*. In: AVoCS 2011, September 2011, Newcastle. (http://eprints.ecs.soton.ac.uk/22480/)

This research was carried out within the European Commission ICT DEPLOY contract / grant number: 214158.

A Framework for Diagrammatic Modelling Extensions in Rodin

Vitaly Savicks and Colin Snook

University of Southampton

The Event-B language is purposely designed to have minimal features to support automatic proof. In this sense the language can be seen as low-level in some problem domains. It is attractive therefore to enhance the language with higher-level modelling constructs (often using diagrammatic notations) and automatically generate Event-B for verification. This was the motivation behind the State machines plug-in. The Eclipse Modelling Framework (EMF) provides supporting infrastructure for building modelling tools. An extensible EMF framework for the Event-B language is already available. Further frameworks are now available to support language extensions and diagrammatic editors.

These frameworks have been developed to support higher-level diagrammatic models which contribute to an Event-B model. For verification purposes, model extensions are converted into pure Event-B. This is done within EMF, prior to persisting into the Rodin database. An alternative approach would be perform the translation in Rodin, or even to extend the verification tools to work directly upon the higher-level modelling constructs. However, the disadvantage of these approaches is that the model extensions have to be replicated within the Rodin DB. By translating to Event-B within EMF we avoid the need for Rodin to comprehend the higher-level model and persist it as a single string attribute of a generic Rodin element defined for this purpose. A generic facility provides load/save operations of the higher-level model extension. All that is required to invoke this facility is a declaration of the meta-class of the root element of the model extension.

Diagram extension to Event-B meta-model introduces two important abstract classes – Diagram and DiagramOwner – to be used for any diagram tool built on top of Event-B. These are high-level abstractions for any particular diagram meta-model that a developer decides to implement for his specific tool. Diagram class is an abstraction of a diagram element that would usually map to a canvas in a concrete diagram editor. DiagramOwner, in turn, is an element of a diagram that can contain other diagrams, as it has diagrams containment attribute. These two simple abstractions enable any concrete diagram to have a common type and to contain other diagrams that use this same extension.

The Rodin tool provides an Event-B navigator view that shows the contents of Event-B models in a tree structure. The problem with this view that an EMF developer of Event-B extension would face is that it only displays Rodin elements and doesn't know how to treat EMF elements. This problem is generic, as is generic a solution we've provided. The developer is merely asked to provide an adapter factory extension for his specific domain meta-model. The implementation of the required adapter factory is available from generated EMF.Edit

plug-in. Thus, navigator view will be aware of new extensions to Event-B and how to display them with appropriate labels and icons.

As long as a diagram extension is an extension of Event-B EMF, the navigator support mentioned above works for free, provided a developer has specified a correct adapter factory. However, diagrams are more than just extensions to Event-B when used in a diagram editor, combining both domain model and graphical notation, which are kept in separate resources. The navigator support for diagrams provides an extension point to diagram editor developers with an interface of a diagram provider that, when implemented, enables opening a diagram from navigator on double click. Potentially the same provider can offer additional functionality for a diagram from navigator view.

Refiner - The Rodin platform provides a utility to automatically make a refinement from an Event-B machine as a starting point for modification. It is necessary to provide this facility for language extensions. The extensions framework extends the Rodin refine utility with a generic utility that reflectively performs a deep clone of a given part of the model. This is invoked for a particular modelling extension by a declaration in the plugin. Although cloning can be handled reflectively, references vary in the way they are handled and therefore the plugin also needs to contribute a class which defines how to handle references.

Generator - A generic Event-B generator is provided to simplify the task of defining a transformation from a model extension to Event-B. The generator is declared giving the meta-class of the root source element of the modelling extension. Rule classes are defined and declared for each meta-class that the generator is intended to work for. Each Rule must define 3 methods:

1. enabled - returns a boolean to indicate whether the rule is appropriate for a given source element
2. dependenciesOK - returns a boolean to indicate whether any pre-requisite generation has been completed by other rules. (If not the rule will be deferred and tried again later).
3. fire - returns a list of definitions which describe how references and containments need to be altered as part of the generation.

Systematic Development of Embedded System Designs using RRM Diagrams and UML-B

Manoranjan Satpathy[1], Colin Snook[2], Silky Arora[1]

[1]GM India Science Lab, Bangalore -560066, India

[2]School of Electronics & Computer Science

University of Southampton, UK, SO17 1BJ

(Extended Abstract)

We have developed a modeling notation called RRM (Requirement, Refinement and Modeling) diagrams, a graphical formal notation which can model requirements. A RRM diagram can be incrementally extended (or refined) in a step wise manner to consider gradual addition of requirements. From the final RRM diagram (RRM diagram when all requirements have been captured), a Simulink/Stateflow (SL/SF) Design [3] can be derived automatically. A set of mapping rules is used to guide this translation.

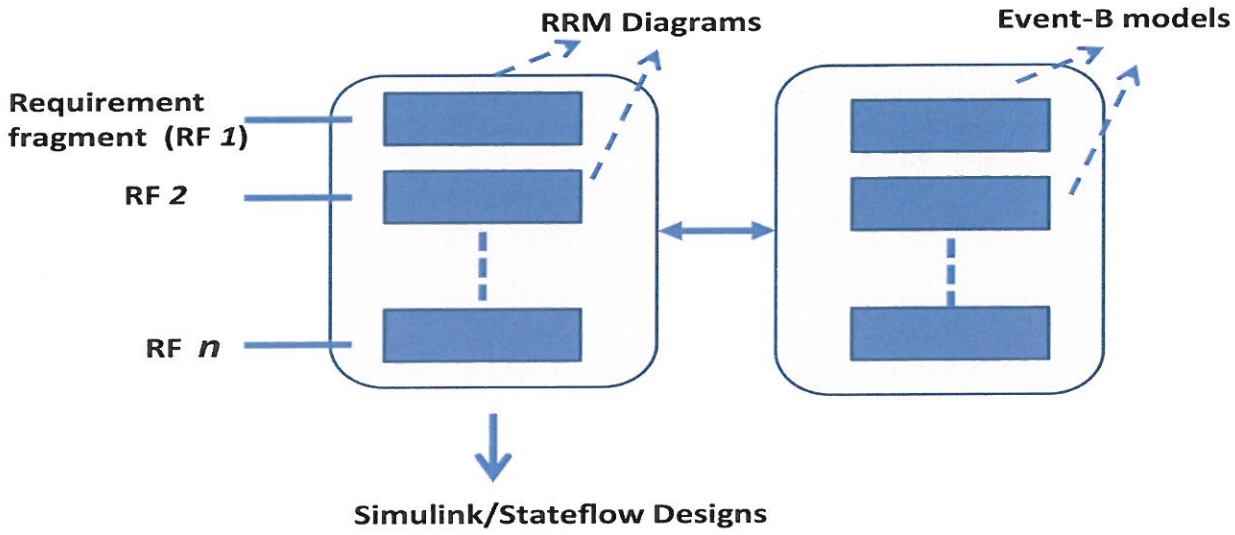


Figure 1. The development Process

Each RRM diagram can have an equivalent Event-B model [1]. The refinement notion of Event-B can be used to show the refinement relationship between RRM diagrams. Figure 1 outlines the whole development process. The Rodin

platform [4] is used to show the refinement relationship between the Event-B models related to the RRM diagrams.

UML-B [2] is a visual 'front-end' for the Event-B notation and includes a state machine diagram editor. Tool support for UML-B is provided by a plug-in to the Rodin platform. Event-B is generated automatically from the UML-B model so that the Rodin verification tools can be used to validate and verify the model.

RRM diagrams can be encoded as UML-B state-machines using a simple mapping between the diagram notations. Therefore, the verified consistency of a UML-B model supports the consistency of its corresponding RRM diagram. Algorithms for transforming RRM diagrams to Simulink/Stateflow (SL/SF) models have been developed. This translator within the UML-B environment is under development.

SL/SF designs are widely used in industry for the development of control applications. Verification and validation infrastructure around SL/SF like simulation, software-in-loop (SIL) testing and processor-in-loop (PIL) testing are available which the engineers are familiar and comfortable with. For example, directly providing the engineers with the C-code obtained from Event-B models or RRM diagrams would not be acceptable to the practitioners. Therefore obtaining a SL/SF design that is correct by construction would be a significant contribution to the development of control designs.

References:

1. Abrial J.R., Hallerstede S.(2006). Refinement, Decomposition, and Instantiation of discrete models, Fundamentae Informatica, 2006.
2. Snook C., Butler M. (2006). UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Method. Vol 15(1), pp. 92—122
3. The Mathworks, http://www.mathworks.com
4. RODIN. RODIN project homepage, http://rodin.ncl.ac.uk/.
5. Satpathy M, Ramesh S. (2009). Formal Foundation to Systematic Development of Simulink/Stateflow models, Dagstuhl Seminar on Refinement based methods for the construction of dependable systems.

CODA: A Formal, Event-B based Refinement Framework for High Integrity Embedded System Development

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

Neil Evans Neil Grant Helen Marshall
AWE, Aldermaston, Reading
{Neil.Evans, Neil.Grant, Helen.Marshall}@awe.co.uk

January 16, 2012

Abstract

Orchestra is a simulation environment for embedded systems [Wickstrom, 2007] within which system models can be refined and decomposed into a collection of communicating components, modelled as software, hardware (including emulated hardware) or as mechanical devices. The notion of refinement in Orchestra, however, is not formal and is not supported by any formal method. Event-B [Abrial, 2010] is a proof-based modelling language and method that enables the development of specifications using a formal notion of refinement. The Rodin platform [Abrial et al., 2010] is the Eclipse-based IDE that provides automated support for Event-B modelling, refinement and mathematical proof. UML-B [Snook and Butler, 2006] provides specialisations of UML entities, including UML state machines, to support formal model refinement within the Rodin platform. UML-B is translated directly to Event-B.

CODA is a framework for the formal refinement of Orchestra-like models that supports modelling and proof at different refinement levels with UML-B state machines and is integrated as a plug-in within the Rodin environment.

The CODA plug-in introduces a *Component View* which allows the embedded system developer to describe, graphically, the components that comprise the system and the connections between those components. The plug-in also provides the facility to associate delays with the connections, to specify deadlines, and manage component synchronisation. CODA supports a refinement-based method where the system can be specified initially as a single, untimed, top-level state machine. This specification can then be refined in a step-wise manner until all the components and connections of the system have a timed, concrete representation. This

is called the Orchestra *high-level* model. In this high-level model, finite state machines represent the functional specification of each component.

In the next stage of the CODA method, the high-level model is refined formally to produce the *I/O level* model. In this refinement, the handling of component port communication is separated from the functional model, as the first stage in separating the component function, which will be represented in software, from its timing, which will be represented in hardware. Existing Event-B formal refinement and decomposition techniques, developed for components communicating by shared variables [Butler, 2009], are extended to Orchestra components communicating via channels with delay. The three-way decomposition separates the *sending* component, the *receiving* component and the *middleware* which represents the abstract channel. Further decomposition of the middleware enables a formal refinement of the abstract, delayed data representation of the channel to the required, bit-level communication protocol at the Orchestra I/O module level that will be implemented in hardware. This I/O level model can then, in turn, be refined formally to derive a *register-level* representation. At this register level, all communication and synchronisation is managed by the hardware.

CODA provides a method for formal embedded system specification refinement that can be integrated within the Orchestra, simulation-based verification flow.

References

Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.*, 12:447–466, November 2010. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s10009-010-0145-y. URL `http://dx.doi.org/10.1007/s10009-010-0145-y`.

J.R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge Univ Pr, 2010. ISBN 0521895561.

M. Butler. Decomposition structures for Event-B. *Integrated Formal Methods iFM2009, Springer, LNCS*, 5423, 2009.

C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15 (1):92–122, 2006.

Gregory L. Wickstrom. Advanced system simulation, emulation and test (asset). In *CPA*, pages 443–464, 2007.

ADVANCE: Advanced Design and Verification Environment for Cyber-Physical System Engineering - The Multi-Simulation Framework

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

January 16, 2012

Abstract

Cyber-physical systems are integrations of computing and physical mechanisms engineered to provide physical services including transportation, energy distribution, manufacturing, medical care and management of critical infrastructure. The essential concept of the ADVANCE FP7 project [University of Southampton, 2011] is the key role played by modelling in cyber-physical systems engineering. Modelling should be used at all stages of the development process from requirements analysis to system acceptance testing. While formal proof can be used to verify that the software parts of a cyber-physical system correctly implement a formal model, it is not feasible to use proof to verify a complete cyber-physical system. Conventionally, correctness-by-construction refers to the use of formal models and refinement in the development of software. ADVANCE will go beyond this, supporting the construction of cyber-physical systems and augmenting formal modelling and verification with simulation and testing.

Simulation is the dominant technology for the industrial verification of digital hardware and embedded systems. The use of the Verilog and VHDL modelling languages is widespread in simulation-based digital hardware verification and SystemVerilog and SystemC are used increasingly for the design and simulation of systems. The design and verification of cyber-physical systems, however, introduce new challenges which cannot be easily addressed by existing simulation frameworks. First, it is necessary in a cyber-physical system to model and simulate in the continuous as well as the digital domain. Second, the complexity of highly-concurrent systems cannot be addressed by simulation techniques alone. Refinement-based formal methods, such as Event-B [Abrial, 2010] help considerably to manage this complexity and to verify that the implemented system meets its specification.

It is not feasible to contemplate developing a single simulation language and verification environment that can meet all the requirements for

cyber-physical development and verification. Legacy designs must be reused and the specialised expertise of developers with existing tool chains leveraged. The primary objective of the ADVANCE multi-simulation framework is to address the needs for different design and verification tools, both discrete and continuous, test-based and formal to co-operate within a single development and verification framework. ADVANCE will implement a simulation framework, extending the existing RODIN platfom [Abrial et al., 2010], within which independently-developed, heterogeneous components and sub-systems can be composed in a secure manner to enable *Cyber-Physical System* design and development.

In ADVANCE, the models of the components and sub-systems, developed using formal techniques, are imported directly into the simulation framework using two, complementary techniques. In the first, a simulation model is generated automatically from the formal model using techniques that build upon the code-generation methods developed in the FP7 DEPLOY project[University, 2009]. This technique will be used when the component model is mature and less-prone to frequent changes and will have the advantage of fast and efficient simulation. In the second, the model will be executed by its own host simulator and the ADVANCE multi-simulation framework will manage the communication of data between multiple simulation hosts, enabling simulation and verification of the whole system. The ProB formal model animator and model checker[Leuschel and Butler, 2003], also developed in the FP7 DEPLOY project, will be extended and its performance improved so that it too can be integrated within the ADVANCE multi-simulation framework. This will facilitate early system integration while the model is still being developed and allow interactive de-bugging.

The partners in the ADVANCE Project are University of Southampton, Alstom Transport, Systerel, Heinrich-Heine Universitat Dusseldorf, Critical Software Technologies Ltd.

References

J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12:447–466, November 2010. ISSN 1433-2779.

J.R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge Univ Pr, 2010. ISBN 0521895561.

M. Leuschel and M. Butler. ProB: A model checker for B. *FME*, pages 855–874, 2003.

Newcastle University. DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity, May 2009. http://www.deploy-project.eu/.

University of Southampton. ADVANCE: Advanced design and verifica-

tion environment for cyber-physical system engineering, 2011. URL http://www.advance-ict.eu/.

Proving Consensus

Jeremy W. Bryans and Alexei Iliasov

School of Computing Science, Newcastle University, United Kingdom, NE1 7RU
{Jeremy.Bryans, Alexei.Iliasov}@ncl.ac.uk

Abstract. Consensus problems arise in any area of computing where distributed processes must come to a joint decision. Although solutions to consensus problems have similar aims, they vary according to the processor faults and network properties that must be taken into account, and modifying these assumptions will lead to different algorithms. Reasoning about consensus protocols is subtle, and the correctness proofs given in the literature are often informal. In this presentation we give some recent work on the development and proof of the Paxos algorithm using Event-B.

Verification and validation of BPEL processes. A proof and animation based approach.

Idir Ait-Sadoune*, Yamine Ait-Ameur**, and Mickael Baron***

E3S - SUPELEC, Gif-Sur-Yvette, France*
idir.aitsadoune@supelec.fr
IRIT - ENSEEIHT, Toulouse, France**
yamine@enseeiht.fr
LIAS - ENSMA, Poitiers, France***
baron@ensma.fr

1 Proposed approach

With the aim to provide a complete tool for the web services compositions validation, we have integrated various plugins in a single platform. This platform is based on the Eclipse core and contains the WSDL and BPEL editors plugins (1), the BPEL2B (2) and B2EXPRESS (3) developed plugins and the various plugins of the RODIN platform (4). Different views are offered by this platform : WSDL and BPEL editors to describe different web services and their orchestration graphically or using the XML syntax, BPEL2B plugin to transform the WSDL/BPEL specifications onto Event-B models, the different plugins of RODIN to perform web services composition validation on the obtained Event-B model and the B2EXPRESS plugin to animate the obtained Event-B model.

1.1 The BPEL2B plugin (2)

The Event-B based approach, proposed for BPEL processes verification [2, 3], defines different transformation rules from a BPEL description to an Event-B model. These rules are based on the transformation of an element or an attribute of BPEL to a specific clause in an Event-B model. We have automated this transformation process in the BPEL2B plugin. This plugin builds a RODIN project from WSDL and/or BPEL files. A RODIN project consists essentially of two types of components: **CONTEXT** and **MACHINE**. As described in [2], the **CONTEXT** is obtained from the WSDL description and the **MACHINE** from the BPEL specification.

The RODIN platform uses as input the obtained RODIN project, containing both **CONTEXT** and **MACHINE** components. When the Event-B models, formalizing a BPEL description are obtained, they may be enriched by the relevant properties that formalize the user requirements and the soundness of the BPEL defined process (for example adding a gluing invariant). In Event-B, these properties are defined in the INVARIANTS and the THEOREMS clauses. The proof activity is performed using the prover of the RODIN platform.

1.2 The B2EXPRESS plug-in (3)

The B2EXPRESS plugin implements the proposed approach [1] for animating Event-B model by data models expressed in the EXPRESS language. This approach takes the Event-B model as input and generates the corresponding EXPRESS model. Animation consists in instantiating the different entities of the obtained EXPRESS model which is hidden to the user.

Events are triggered on the graphical user interface by mouse clicks. Each time an event is triggered, B2EXPRESS produces instances of the corresponding EXPRESS schema. B2EXPRESS, checks if the instances are correct and fulfil all the local and global EXPRESS constraints. Error messages are returned and spied by B2EXPRESS in order to localise the errors on the Event-B source code.

Two animation modes are offered: *the guarded mode* where only those events, whose guard evaluates to true, can be triggered, and *the free mode*, which leaves freedom to the user to trigger any event even if its guard evaluates to false.

In addition, the Event-B model traces can be controlled by a process algebra expression (with interleaving semantics) to define the behaviour expected for this model. B2EXPRESS offers also the possibility to instantiate EXPRESS entities by triggering events corresponding to traces of a process algebra expressions. This possibility allows a developer to validate requirements expressed by behaviours.

2 Extensions

Our aim is to animate BPEL processes using BPEL2B and B2EXPRESS plugins by hiding them to the user. This is possible by developing a plugin, that takes as input the WSDL/BPEL descriptions and invokes BPEL2B and B2EXPRESS plugins to generate an Event-B model. Instead of triggering events of Event-B model, the user will trigger the BPEL activities. When an activity is triggered, the new developed plugin produces instances of the corresponding EXPRESS schema that is obtained from the Event-B model generated from the BPEL description. B2EXPRESS checks if the instances are correct and localises the errors on the BPEL source code instead of localizing them in the Event-B source code.

References

1. Ait-Sadoune, I., Ait-Ameur, Y.: Animating Event B Models by Formal Data Models. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA). pp. 37–55. Communications in Computer and Information Science, Kassandra Greece (2008)
2. Ait-Sadoune, I., Ait-Ameur, Y.: A Proof Based Approach for Modelling and Verifying Web Services Compositions. In: 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 1–10. IEEE Computer Society, Potsdam Germany (2009)
3. Ait-Sadoune, I., Ait-Ameur, Y.: Stepwise Design of BPEL Web Services Compositions, An Event B Refinement Based Approach. In: 8th ACIS International Conference on Software Engineering Research, Management and Applications (SERA). pp. 51–68. Studies in Computational Intelligence, Montreal, Canada (2010)

Formal Specification of a Mobile Diabetes Management Application Using the Rodin Platform and Event-B

Daniel Brown, Ian Bayley, Rachel Harrison, Clare Martin
Oxford Brookes University

[11062958, ibayley, rachel.harrison, cemartin]@brookes.ac.uk

Introduction

This paper describes a study performed to learn about the Rodin platform and Event-B language. The subject is the specification of the functionality of an existing mobile application for managing diabetes on a mobile device.

Our research is based on previous research in the same domain [1, 2]. Our aim is to utilize new approaches in formal methods, address issues with regards to un-discharged proof obligations in the original research, and to introduce further requirements to the specification. Approximately 4% of the population in the United Kingdom and an estimated 8% of the United States population have some form of diabetes [3, 4]. The monitoring of blood glucose levels is an important aspect in managing the condition to prevent high blood glucose (hyperglycemia) which can lead to diabetic coma (ketoacidosis), a potentially life-threatening condition and low blood glucose (hypoglycemia), which can cause seizures [5, 6]. Traditionally log books have been used to record and monitor blood glucose levels; however with the increased popularity of smart phones an effort has been made to develop applications for tracking and analysing patient's blood glucose levels [7]. This research aims to address the need for the development of correct and robust mobile applications to meet the needs of people with diabetes type 1 and 2.

We chose the application *Diabetes Personal Manager* for the iPhone for our study, which is available through Apple's App Store [8]. This application was selected because it includes functionality which provides the user with suggested insulin dosages based on their personal settings and details of the meals they input. In addition information on the formulas used to calculate the insulin doses are available.

Determining the Requirements

The requirements of the application were elicited and analysed before specifying the application using the Rodin platform. Two separate data domains are needed: the user's personal settings and the meal records containing information on consumed food and drink which is input by the user. The user's personal settings contain information relating to target blood glucose and correction factors which are used to calculate the amount of insulin needed to the target blood glucose level. The meal records contain details of the user's meals, such as carbohydrate intake and their current blood glucose level. A snapshot of the user's personal settings at the time the meal is recorded is also stored in the meal record, as well as the results of calculations relating to the insulin required. The specification includes operations which offer the ability to add and remove meal records, and the updating of personal settings. The data stored in the meal records is shown in Table 1.

Meal Record

Personal Settings
Glucose Measurement Unit
Target Blood Glucose
Correction Factor
Carbohydrate to Insulin Ratio

User Inputs
Date and Time
Meal Type (Breakfast, Lunch, etc)
Current Blood Glucose
Carbohydrates in Meal
Amount of Insulin Used
Additional Notes

Calculations
Insulin Required to Reach Target Blood Glucose
Insulin Required for Carbohydrates
Suggested Insulin Based on Carbohydrates and Blood Glucose Levels

Table 1. A breakdown of a meal record into the three sub-categories and the information stored within each category.

Specifying the App Using Rodin

Refinement was utilized in order to breakdown the detail of how a meal record is stored into small steps. In the initial machine, an abstract definition of a meal record was established by the use of functions to relate a meal record to the three sub categories of personal settings, user inputs and calculations. The operations for adding, removing and updating personal settings were also included in the initial machine, which was then extended and adjusted as necessary in each refinement stage. Below is a list explaining the purpose of each refinement step in the specification.

1. Add detail to the data to be stored in the meal record with regard to the user's personal settings.
2. Add detail to the storing of user inputs for a meal record.
3. Add detail to the calculations stored in a meal record.
4. Include the link between the current state of the user's personal settings and recording this state in the meal record.
5. Add more detail to the calculations stored within the meal record, by replacing the abstract definitions with the actual formula to be used in the calculations.
6. Add safety warnings of high and blood glucose levels in the add meal operation.

The refinement steps are taken in order to make the creation of the specification a more manageable process, and to ensure at each stage that the added detail to the specification maintained the invariants. This approach allows for any issues encountered with regards to proof obligations and any implications of the operations to be isolated to a small area at each stage of refinement, and facilitates the process of creating the specification.

During the creation of the specification errors with the *Diabetes Personal Manager* application were identified, mainly with regards to unexpected behaviour and poor handling of errors during the calculations. These errors were addressed during the specification in order to ensure correctness. The automatic proof checker provided with the Rodin Platform was used to discharge the majority of the proof obligations, with the exceptions being with regards to the inclusion of the mathematical detail to the calculations. The Rodin platform was found to be intuitive to use, and creating this specification gave us better understanding of the differences between Event-B and B, and what could be achieved with the tool in future work. Throughout this example the Pro-B plugin was used to animate the specification. This proved effective at identifying what was would cause the proof checker to fail and where the operations broke the invariant.

Conclusions

This example proved invaluable in providing basic understanding of using the tool and language. Having only previous experience with Atelier-B, the transition to using Rodin was achieved with ease and the automatic proof checker was more effective [9]. The use of refinement in order to add detail in manageable stages aided the learning process and prevented the specification from becoming a steep learning curve. The availability and ease of installing plug-ins was a great asset to the Rodin platform, as Pro-B proved to be a useful problem solving tool through the use of animation. We will continue our use of Rodin in order to gain further understanding and unlock the full potential the tool and Event-B language provide.

References

[1] Poerschke, C., 2004. 'Development and Evaluation of an Intelligent Handheld Insulin Dose Advisor for Patients with Type-1 Diabetes'. Oxford: Oxford Brookes University.

[2] Poerschke, C., Lightfoot, D., Nealon, J.L. and Bert, D. ed., 2003. 'A Formal Specification in B of a Medical Decision Support System'. ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku, Finland, June 2003, pp. 497-512. Springer.

[3] Diabetes UK, 2011. 'Diabetes prevalence 2011 (Oct 2011)'. [online] Available at: <http://www.diabetes.org.uk/Professionals/Publications-reports-and-resources/Reports-statistics-and-case-studies/Reports/Diabetes-prevalence-2011-Oct-2011/> [Accessed 10th January 2012].

[4] National Institutes of Health, 2011. 'National Diabetes Statistics, 2011'. [online] Available at: <http://diabetes.niddk.nih.gov/DM/PUBS/statistics/DM_Statistics.pdf> [Accessed 10th January 2012].

[5] American Diabetes Association, 2011. 'Hyperglycemia (High blood glucose)'. [online] Available at: <http://www.diabetes.org/living-with-diabetes/treatment-and-care/blood-glucose-control/hyperglycemia.html> [Accessed 10th January 2012].

[6] American Diabetes Association, 2011. 'Hypoglycemia (Low blood glucose)'. [online] Available at: <http://www.diabetes.org/living-with-diabetes/treatment-and-care/blood-glucose-control/hypoglycemia-low-blood.html > [Accessed 10th January 2012].

[7] Garcia, E., Martin, C., Garcia, A., Harrison, R. and Flood, D., 2011. 'Systematic analysis of mobile diabetes management applications on different platforms'. , Information Quality in eHealth, 7th Conference of the Austrian Computer Society Workgroup: Human-Computer Interaction and Usability Engineering (USAB 2011), Graz, Austria. 25-26th November 2011

[8] iTenuto Soft, 2011. '*Diabetes Personal Manager* '. [online] Available at: < http://itunes.apple.com/us/app/diabetes-manager/id368455341?mt=8> [Accessed 13th January 2012]

[9] ClearSy System Engineering, 2011. '*Atelier-B'*. [online] Available at:< http://www.atelierb.eu/en/> [Accessed 13th January 2012]

Synthesis of Processor Instruction Sets from High-level ISA Specifications

Andrey Mokhov†, Alexei Iliasov†, Danil Sokolov‡, Maxim Rykunov‡, Alex Yakovlev‡, Alexander Romanovsky†

†School of Computing Science, Newcastle University, UK
‡School of Electrical, Electronic and Computer Engineering, Newcastle University, UK

***Abstract*—Instruction sets of modern processors contain hundreds of instructions defined on a relatively small set of datapath components and distinguished by their codes and the order in which they activate these components. Optimal design of an instruction set for a particular combination of available hardware components and software requirements is crucial for building systems with high performance and energy-efficiency, and is a challenging task involving a lot of heuristics and high-level design decisions. The overall design process is significantly complicated by inefficient representation of instructions which are usually described individually despite the fact that they share a lot of common behavioural patterns.**

We present a new methodology for compact graph representation of processor instruction sets which gives the designer a new high-level perspective for reasoning on large sets of instructions without having to look at each of them individually. This opens the way for various transformation and optimisation procedures, which are formally defined and explained on several examples, as well as practically evaluated on an FPGA platform. To guarantee formal correctness of the proposed techniques we use the Event-B modelling framework as a formal specification and verification back-end.

Modern processors become increasingly diversified in terms of power modes, heterogeneous hardware platforms, requirements for legacy software reuse, etc. This is amplified by the rapidly growing demand for low power consumption, high performance and small area of produced circuits. As a result, under the pressure of time-to-market constraints, a computer architect faces a *design productivity gap* [1]: the capacity of modern CAD tools is insufficient to explore the variety of possible architectural solutions and identify the optimal instruction set, which is a large part of a processor design.

One of the key difficulties in designing instruction sets is the necessity to comprehend and to deal with a very large number of instructions, whose microcontrol implementation can dynamically adapt to the current operation mode of the system. In order to overcome these difficulties, the instructions and groups of instructions have to be managed in a compositional way: the specification of an instruction set should be composed from specifications of its subsets. Furthermore, one should be able to transform and optimise the specifications in a fully formal way.

There are several well-established approaches for the functional-level description and formal verification of ISA. *Event-B* [6] is a widely adopted language for specifying *first-order logic* systems and doing refinement on that representations. Being combined with the RODIN theorem prover tool, it becomes a powerful platform for proving that a (refined) system satisfies the initial specification, e.g. does not leave a certain set of 'good' states during its operation. *HOL* [2] is a computer-assisted proving environment for constructing verifiably correct mathematical proofs. Although its expressiveness is unrivalled, the generic nature of a tool such as ISABELLE/HOL makes it more suitable for analysing individual instructions with deep mathematical properties; see, for example, verification of IA-64 division algorithm [3].

These formal ISA methods have a history of being used for reasoning about hardware implementations, however they are more targeted to the software-related aspects of processor functionality. No hardware implementation issues are usually taken into consideration apart from those directly visible to the instructions, such as the size of addressable memory, the number and type of available registers, etc. As a result, an ISA designer does not have the full control on how the specified functionality is achieved in hardware, what the costs of every instruction are in terms of energy consumption and computation resources, how to minimise latency of instruction decoding logic or how to dynamically adapt the processor to the current operating conditions. Modelling such low-level implementation details in Event-B or HOL is costly; a more targeted formalism is needed to interface the representation of knowledge about instruction sets with that of knowledge about their execution.

An instruction execution is associated with a sequence of atomic *actions* (usually acyclic) to complete the task. While a sequential run of actions is sufficient to achieve the instruction functionality, it is often practical to enable some of the actions concurrently in order to speed up the instruction execution and to efficiently utilise the available energy. The distinctive classes of instruction functionality are arithmetic operations, data handling, memory access and flow control. Being combined with various operation modes and resource restrictions, such a diversity of instruction functionality presents a real challenge to the efficient design of microprocessors.

There is clearly a niche in microprocessor EDA where the following design requirements need to be addressed:

- compact description of individual instruction functionalities as partial orders of atomic actions;
- efficient representation of complete instruction sets to allow their transformations (optimisation of encoding, retargeting for different hardware platforms, etc.);
- capturing of processor operation modes as explicit parameters of the instruction sets;

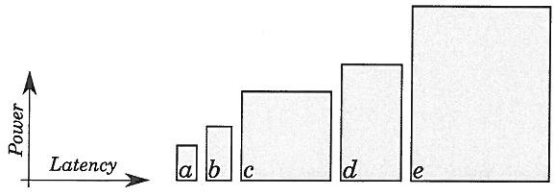


Figure 1: Datapath components for *DP3* implementation

- possibility to express the resource availability constraints;
- encoding of an instruction set for different optimisation criteria (code length minimisation, complexity of decoding logic, legacy software compatibility, etc.);

We propose to address these requirements using a graph model, called Conditional Partial Order Graphs (CPOGs) [5]. This model is particularly convenient for composition and representing large sets of partial orders in a compact form. It can be equipped with a set of mathematical tools for the refinement, optimisation, encoding and synthesis of the control hardware which implements the required instruction set, similar in spirit to the approach based on *control automata*. We envisage that the model can be used as a complementary formalism for the existing ISA methodologies providing a formal link between the software and hardware domains. Although general-purpose modelling languages and proving environments, such as Event-B or HOL, may be used to a similar effect, the CPOG model offers a superior mathematical construction permitting automated analysis and synthesis.

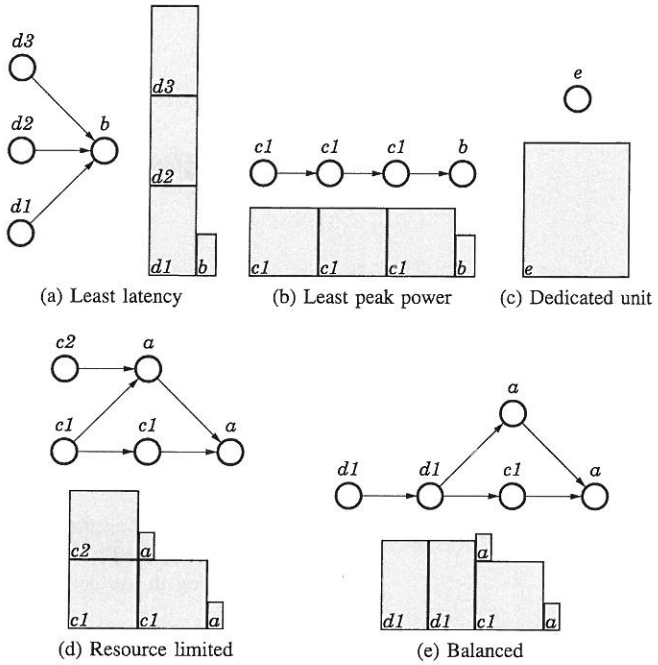
We present a significant contribution to the relatively new concept of CPOGs. The previous CPOG-related publications, focused on algebraic CPOG properties, controller synthesis, verification and optimal encoding of partial orders, while this work brings all these methods to the area of formal specification of processor instruction sets and introduces *CPOG transformations* as an efficient way of instruction set management. The full paper can be accessed as a technical report [4].

Example. Consider a common low-level GPU instruction, called *DP3*, which given two vectors $\mathbf{x} = (x_1, x_2, x_3)$ and $\mathbf{y} = (y_1, y_2, y_3)$ computes their dot product $\mathbf{x} \cdot \mathbf{y} = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$. There are many ways to achieve the required functionality in hardware; consider the following datapath components (denoted by $a \ldots e$) which can be used to fulfil this task:

a) 2-input adder;
b) 3-input adder;
c) 2-input multiplier;
d) fast 2-input multiplier;
e) dedicated DP3 unit.

Now we associate two attributes, *execution latency* and *power consumption,* with every component. Fig. 1 visualises them as labelled boxes, whose dimensions correspond to their attributes; the area of a box represents *energy* required for the computation.

Depending on the current operation mode and availability of the components, the processor has to schedule their activation in the appropriate partial order. Fig. 2 lists several possible partial orders together with their power/latency profiles. For



(a) Least latency (b) Least peak power (c) Dedicated unit

(d) Resource limited (e) Balanced

Figure 2: Different implementations of *DP3* instruction

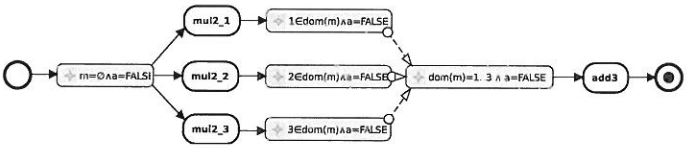


Figure 3: Least latency implementation (flow aspect)

example, the fastest way to implement the instruction is to compute multiplications $tmp_k = x_k \cdot y_k$ concurrently using three fast multipliers $d1$-$d3$ and then compute the final result $tmp_1 + tmp_2 + tmp_3$ with a 3-input adder b, as shown in Fig 2(a). Using the Event-B formalism, we can formally prove that this procedure correctly computes the dot product, and extract the corresponding partial order graph (see Figure 3).

Our approach allows us to verify functional correctness of every instruction implementation using Event-B, and then to compose them into an instruction set, thereby allowing the designer to operate with all of them as with a single mathematical object.

References

[1] International Technology Roadmap for Semiconductors: Design, 2009.
[2] A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving (ITP)*, pages 243–258, 2010.
[3] J. Harrison. Formal verification of IA-64 division algorithms. In *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), LNCS 1869*, pages 234–251. Springer, 2000.
[4] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, and A. Romanovsky. Synthesis of Processor Instruction Sets from High-level ISA Specifications. Technical report, Newcastle University, January 2012.
[5] Andrey Mokhov and Alex Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.
[6] F. Yuan and K. Eder. A Generic Instruction Set Architecture Model in Event-B for Early Design Space Exploration. Technical Report CSTR-09-006, University of Bristol, September 2009.

An Event-B Plug-in for Creating Deadlock-Freeness Theorems

Faqing Yang and Jean-Pierre Jacquot

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

Abstract. This paper presents DFT-generator, a small tool to generate Deadlock-Freeness Theorems (DFT) in Event-B specifications, to overcome the lack of deadlock checking in the core of Event-B and of its supporting environment, Rodin.

Event-B [6,2] is an evolution of the classic B method [1]. It has been designed for modeling complex systems such as concurrent, reactive systems, or complex algorithms.

Deadlock-freeness is not well integrated into the Event-B framework. As many other temporal properties, we can use "tricks." For deadlock-freeness, we build the disjunction of the guards of all events other than *INITIALISATION* and we prove it is a theorem. So, we can be sure that one event at least can always been fired.

There exists other techniques which do the same work, like:

- Animation with an interactive tool such as Brama[1] allows one to observe the occurrence of deadlocks.
- Modeling checking with a tool like ProB [4] allows one to check for (absence of) deadlocks in a finite space and partial transitions.

These techniques can help us to find a deadlock, but can't prove that the system is free of deadlocks. On the other hand, using theorem proving techniques on some theorems which express deadlock-freeness, we can formally assess that the system is actually free of deadlocks. DFT-generator, a little plug-in to create the deadlock-freeness theorems for Rodin platform[2], it is useful:

- for helping to specify systems that must be free of deadlocks,
- for helping to get large scale Event-B specifications correct, in particular by replacing the error-prone manual writing of ad-hoc formulae,
- for identifying missing properties or invariants in the specification.

Deadlock-freeness is not a monotonous property with respect to refinement: it should be established for each refinement. Automating the generation of the theorems is needed for several reasons:

[1] http://www.brama.fr
[2] http://sourceforge.net/projects/rodin-b-sharp/

- using "copy and paste" procedures is highly non trustable. The probability to introduce an error is high and the length of the formula makes spotting errors hard,
- any modification of the model requires a modification of the theorems. In particular, it should be easy to re-generate the formulae while correcting the model.
- manually generating the theorems is not intellectually challenging (it is boring in fact) and takes a lot of useful time.

The core of the generation is a syntactic manipulation of the model: extracting the guards, gluing them together in a single formula, and inserting it in the specification. Programming tools and techniques to cover such tasks are well known.

Thanks to Rodin, which is able to manage large formulae and specification, the generation goes without any difficulty. Discharging the proof obligations of DFT is of course the biggest time-consuming activity.

A very positive aspect of our experience is the assessment of the maturity of Rodin. The development effort we had to put into the project was reasonable. As is always the case with large API, the first use requires an important learning effort. This investment pays back afterwards since the API is easy to use and quite powerful. It is then reasonable the try to overcome limitations of Event-B by developing small assisting tools rather than waiting major evolution of the formal framework.

The size of the generated formulae, while not a surprise, still raises the issue of their manageability. The problem is actually not specific to our tool but comes from the formal core of Event-B. Assessing formal properties of large B models is a very active research domain. One way to avoid proving huge formulae is to use dynamic techniques such as model-checking; ProB [5] allows this. ProB has been used on our specifications. The model-checker did identify the deadlocks in the 1D specification of platooning [3]. However, it failed on the 2D model due to the complexity of the geometric space (a plane). So, at present, we do not have any other mean than to discharge the 637 line-long DFT formula to assess the safety of our model.

References

1. Abrial, J.R.: The B Book. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd International Symposium on Theoretical Aspects of Software Engineering (TASE'08). Nanjing, China (2008)
4. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10(2), 185–203 (2008)
5. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale b models with prob. Formal Aspects of Computing pp. 1–27 (2011), http://dx.doi.org/10.1007/s00165-010-0172-1, 10.1007/s00165-010-0172-1
6. Metayer, C., Voisin, L.: The Event-B Mathematical Language (Oct 2007)

The Theory plug-in and its applications

Issam Maamria and Michael Butler

ECS, University of Southampton, Southampton SO17 1BJ, UK
{im06r, mjb}@ecs.soton.ac.uk

Abstract. The Rodin platform provides an extensible toolset for modelling and reasoning in Event-B. Modelling and reasoning in Event-B go hand in hand to enable a profound understanding of the system in question. An important aspect of any toolset is its usability and extensibility. The Theory plug-in complements the Rodin platform by providing a mechanism by which users can easily extend its modelling and reasoning capabilities. The mathematical language of Event-B provides a number of useful operators, and the proof infrastructure of Rodin contains many powerful and effective proof procedures. However, prior to the Theory plug-in, extending the mathematical language and proof capabilities of Rodin was a cumbersome and impractical for normal platform users. This was due to the need of writing Java code to provide the simplest of extensions. The Theory plug-in aims at addressing the aforementioned issues in the following way. It provides a distinct construct called 'theory', similar in structure to contexts and machines. The theory construct can be used to define datatypes, operators, polymorphic theorems, rewrite rules as well as a special type of inference rules. In this presentation, I will discuss the capabilities of the Theory plug-in. Several examples of extensions will be examined, and a small case-study on how to use theories as part of the modelling activity will be presented. The use of proof obligation is emphasised to ensure that all extensions defined by theories are conservative with respect to the Event-B logic. Finally, I will discuss the different direction in which the Theory plug-in can be exploited to improve the modelling and proving experience of end-users.

Can rippling discover the missing lemmas for invariant proofs?

Gudmund Grov, Yuhui Lin & Alan Bundy
University of Edinburgh

ggrov@inf.ed.ac.uk Y.H.Lin-2@sms.ed.ac.uk bundy@inf.ed.ac.uk

The hypothesis of our ongoing AI4FM project is that proof obligations (POs) which require user interaction can often be grouped into families where each family can be proved by the same proof strategy. In cases where this family has a known generic proof strategy we will apply this — in other cases our goal is to learn a strategy from one (or at least a few) expert-provided proofs.

In order to achieve this we need a strategy language which allows us to both express high-level proof strategies, as well as enable learning of strategies from proofs. In this talk we will first outline our current ideas for this strategy language. We will then discuss how an existing generic proof strategy called rippling [1] can be utilised for the Event-B invariant preservation (INV) POs family. We observe that INV POs can account for a significant part of all of the POs that require human interaction. To illustrate, 188 out of 317 (59%) of the undischarged POs in the BepiColombo case study[1] were of this type. Moreover, we argue that rippling is more robust to changes of the Event-B model, since the meta-level strategy, which will remain the same, is stored, rather than the object-level proof, which is more likely to change.

Rippling is a rewriting technique which was developed originally for inductive proofs. It is applicable in any scenarios where one of the assumptions can be embedded in the goal. By forcing rewriting towards this assumption, we can guarantee termination. A key advantage of rippling is that the strong expectation of how the proof should succeed can help us to automatically discover missing lemmas, a process which has to be done manually in Rodin. We will here outline a novel approach which combines this lemma discovery mechanism with a theory formation tool called IsaScheme [4]. Our experiments are being implemented in IsaPlanner [2] using Schmalz's Rodin to Isabelle translator [5].

In certain cases, the required lemma has to be represented as an invariant of the machine we are working on. Time permitting, we will outline an idea on how such lemmas can be added to as a machine invariant, utilising ProB [3] as a counter-example checker.

Acknowledgements. This work is supported by EPSRC grant EP/H024204/1 (*AI4FM: using AI to aid automation of proof search in Formal Methods*). Thanks to our AI4FM partners and members of the Mathematical Reasoning Group in Edinburgh for valuable discussions.

[1] Available from the Deploy webpage: http://deploy-eprints.ecs.soton.ac.uk/

References

[1] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[2] Lucas Dixon and Jaques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[3] Michael Leuschel and Michael Butler. ProB: an Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, 10(2):185–203, March 2008.

[4] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based synthesis of inductive theories. In *MICAI*, volume 6437 of *LNCS*, pages 348–361, 2010.

[5] Matthias Schmalz. The Logic of Event-B. Technical Report 698, ETH Zurich, Information Security, 2011.

Proof Hints for Event-B Models – Extended Abstract

Thai Son Hoang

Institute of Information Security,
Department of Computer Science,
Swiss Federal Institute of Technology Zurich (ETH-Zurich),
CH-8092, Zurich, Switzerland
htson@inf.ethz.ch

The Event-B is a refinement-based modelling method, which can be used to develop various types of systems correct-by-construction. Being a formal method, various verification conditions are generated as proof obligations for Event-B models for proving properties of the model. Development in Event-B is supported by the RODIN platform (*RODIN*) [1], an Eclipse-based IDE, which contains tools for analysing and reasoning about the formal models. In particular, the task of discharging these proof obligations is first given to the automatic provers of *RODIN*. Afterwards, remaining proof obligations required to be discharged interactively. Usually, manual proofs are considered as "costs" of development, given the required human interaction for produced them, and the difficulty in maintaining them when the formal models evolve.

As the size of the models grows, the complexity of the proof obligations also increases, hence interactive proofs are inevitable. Improving the performance of the automatic provers has been considered with some success [2, 3]. Despite of the improvement of in the percentage of automatic proofs, interactive proofs are still an obstacle for developing formal models.

In this paper, we look at the problem of handling interactive proof from a different angle. Since the proof obligations are generated from the formal models, the question is that can we improve our formal models in such a way that helps with the proofs. After all, modelling using refinement is also just a way of structuring the proof of correctness of the models. We propose some notions to encapsulate some proof details in the formal model, hence reduce the reduce the effort required for proving. We call the additional information of the formal models "proof hints".

This concept of *proof hints* already exists in the form of "witnesses" or "theorems" in Event-B. These useful features are designed not only to help with proving the correctness of the model but also to give more information about the particular model: why it is correct. In fact, the "proof hints" should help to understand the formal model better.

In this paper, we consider the current state of *RODIN* and show two kinds of useful proof information that can be included in the formal models, namely, *hypotheses selection* and perform *case distinction*.

Hypotheses selection Indicates some facts (e.g., invariants or axioms) are required for discharging the obligations.

Case distinction Indicates that the proof can be discharged by consider different cases.

We show that the effect of the proof hints can be "simulated" at the moment by some modelling "tricks" in Event-B.

In the long term, we propose to have an extension to Event-B and to *RODIN*, to have proof hints as part of the model and a plug-in for interpreting the proof hints and applying these hints appropriately for the proofs.

References

1. Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.
2. Jann Röder. Relevance filters for Event-B. Master's thesis, ETH Zurich, 2010. `http://e-collection.library.ethz.ch/view/eth:2278`.
3. Matthias Schmalz. Export to Isabelle plug-in. `http://wiki.event-b.org/index.php/Export_to_Isabelle`, 2011.

Requirements Traceability between Textual Requirements and Event-B Using ProR

Michael Jastram, Lukas Ladenberger, Michael Leuschel

Tracing between informal requirements and formal models is challenging. A method for such tracing should permit to deal efficiently with changes to both the requirements and the model. A particular challenge is posed by the persisting interplay of formal and informal elements.

We developed an approach that is based on the structuring of requirements according to the WRSPM reference model. We provide a system for traceability with a state-based formal method that supports refinement. We do not require all specification elements to be modelled formally and support incremental incorporation of new specification elements into the formal model. Refinement is used to deal with larger amounts of requirements in a structured way.

We created rudimentary tool support for this kind of traceability by integrating Rodin with ProR, the GUI of the Eclipse Requirements Modeling Platform (RMF).

In this workshop, we will give a brief overview of our approach. We will then iteratively develop a small model based on a set of requirements. Once completed, we show how change management can be supported both by our approach and the software tools.

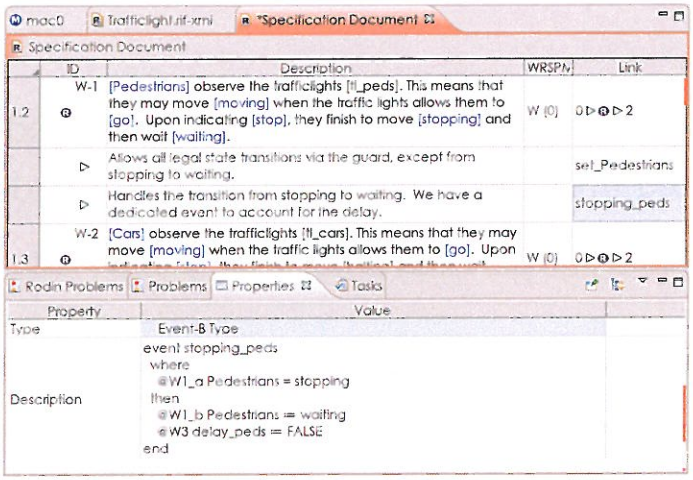
Objectives and Goals

This workshop targets users who are interested in not only creating a correct specification, but in specifying the right system (with respect to requirements). Thereby, it indirectly provides some guidance with respect to structuring their models. Further, attendees will learn how to structure and improve requirements, how to apply quality criteria and how to record them with a tool that supports advanced data structures.

The Eclipse Requirements Modeling Framework

The Requirements Modeling Framework (http://eclipse.org/rmf) is a new Eclipse Foundation project that is based on the recently published Requirements Interchange Format (ReqIF). It grew out of the ProR Requirements Engineering Platform, which was was already presented at the 2010 Rodin Workshop.

ReqIF is more than an exchange format: it is a flexible meta-model that is adaptable to the various needs in requirements engineering. We use ReqIF as the underlying data model for the ProR tool. This allows interoperability with industry tools that support ReqIF. A number of tools, like IBM Rational DOORS, currently support the predecessor of ReqIF. RMF could act as a robust



Integration of ProR (the RMF GUI) with an Event-B formal model. Model elements are highlighted in the requirement text, and annoted traces to the model show the model element in the property view.

OpenSource implementation of the standard upon which various tools can be built, both academic and industrial.

RMF in Complex Projects

Two research projects already take advantage of ProR. In the Deploy project, we establish traceability between formal models and natural language requirements [JHL11]. The formal modeling is done in Event-B using the Eclipse-based Rodin platform. Integration is seamless via drag and drop, and custom renderers support color highlighting of model elements (see Figure).

The Verde project developed a tracing tool that can connect ReqIF requirements with arbitrary artefacts like source code or UML modeling elements [GJ11]. The integration is done by custom adapters.

An industry implementor forum, headed by ProSTEP,[1] currently strives to ensure interoperability amongst commercial ReqIF implementations. RMF is represented in this group and the RMF framework is actively employed, e.g. by generating test data.

References

[GJ11] Andreas Graf and Michael Jastram, *Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (RIF/ReqIF)*, Dagstuhl-Workshop MBEES, 2011.

[JHL11] Michael Jastram, Stefan Hallerstede, and Lukas Ladenberger, *Mixing Formal and Informal Model Elements for Tracing Requirements*, Automated Verification of Critical Systems (AVoCS), 2011.

[1] http://www.prostep.org/en.html

Towards Relating Sub-Problems of a Control System to Sub-Models in Event-B

Sanaz Yeganefard and Michael Butler
Electronics and Computer Science
University of Southampton
UK, SO17 1BJ

Control systems are usually complex as they continually interact with and react to the evolving environment. Besides their complexity, their application is usually in safety critical systems. Such reasons have resulted in requiring a comprehensive requirement document and a rigorous design process for these systems. Formal modelling is known to help with improving system understanding as well as providing mathematical proofs for properties of the system. However, one difficulty that modellers encounter is formalising an informal requirement document. This includes problems such as where to start the modelling and how to refine a model in order to introduce requirements gradually (horizontal refinement) or to introduce design details (vertical refinement).

Based on some conducted case studies, the existing modelling guidelines [2, 3] and the 4-variable model [5], we proposed a structuring approach [6] where requirements of a control system are divided into 3 sections each representing requirements of *monitored*, *commanded* (user settings) and *controlled* phenomena. This structuring of requirement document can facilitate the formal modelling of the system. As the result of further investigation the requirement structuring approach is expanded by dividing the requirements into the following four sections:

- **Monitored:** This section represents requirements of monitored phenomena. Values of such phenomena are determined by the environment/plant.
- **User setting:** To represent requirements of user setting phenomena. These are:
  1. User setting events: User requests which are to be received and responded by the controller.
  2. User setting variables: In some cases it might be necessary to define variables which store the values of user requests.
- **Controlled:** These are requirements of controlled phenomena whose values are set by the controller.
- **Mode:** These are requirements related to the mode phenomenon which represents the status of the controller. Different factors can cause the value of the variable mode to change. For instance a change in the value of a monitored variable (environment) or a user request can result in updating the mode variable.

Our ultimate goal is to model the structured requirements of a control system. In order to decompose the complexity and the effort required for this goal, we take advantage of the idea of sub-problems, which was introduced in Problem Frames (PF) [4], by modelling a system as composable sub-models. PF takes the view that a system can be regarded as two main parts. The *machine* which is the computer including its software, and the *problem world* where the problem is located. In PF sub-problems are defined to help with understanding a complex problem by breaking it. Also each defined sub-problem should have a degree of unity.

Borrowing the definition of sub-problems from PF, each of the four sections introduced above can to some extent represent a sub-problem. This means from the modelling point of view each

sub-problem can be treated as a separate sub-model. Thus, the sub-problems of *user setting*, *controlled* and *mode* can be modelled distinctly. Notice that if one of these sub-problems is adequately straightforward, it can be merged with one of the other two sub-models. Also notice that since the *monitored* sub-problem (i.e. environment) is required to be accessed by all other sub-problems no separate sub-model for environment will be defined.

The three defined sub-models are likely to share variables. The behaviour of a shared variable is usually modelled in details in one of the sub-models, while the other sub-models use the shared variable abstractly, for instance by setting it non-deterministically.

In the initial stage of this work, each sub-model was treated according to the shared variable decomposition style in Event-B [1], by introducing external events[1]. This achieved satisfying results in the horizontal refinement. However, vertical refinement, where design details such as sensors and actuators are introduced, was challenging. This is mainly because of the limitation of this style of decomposition which is not to refine external events.

In the second attempt, each sub-model was treated according to the shared event decomposition style in Event-B. So, we composed the sub-models using the shared variable composition plugin in Event-B. The composed model provides further insight into the system as it gives the overall behaviour (specification) of the system. Besides, the composition can be used as a way of identifying inconsistencies in shared variables of the sub-models using proofs.

The process of structuring requirements into sub-problems and mapping sub-problems to sub-models of the system have been applied to the example of a lane centring controller (LCC). The results of this case study are satisfying and hopeful. By defining sub-models the focus required for modelling the entire requirement document was divided. Also this approach can provide the basis of teamwork, as sub-problems can be modelled separately. However, one case study is not sufficient and we are planning to research further on this approach by using other case studies.

Acknowledgment

This work is supported by GM India Science Lab and partly by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[2] Michael Butler. Chapter 8: Modelling Guidelines for Discrete Control Systems, Deploy Deliverable D15, D6.1 Advances in Methods Public Document. lhttp://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf, 2009. cited 2011 Jan.

[3] Michael Butler. Towards a Cookbook for Modelling and Refinement of Control Problems. Unpublished: in Working Paper. ECS, University of Southampton, 2009.

[4] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.

[6] Sanaz Yeganefard and Michael Butler. Structuring functional requirements of control systems to facilitate refinement-based formalisation. In *Proceedings of the 11th Workshop on Automated Verification of Critical Systems*, 2011.

[1] Events which simulate the behaviour of a shared variable.

Lessons from DEPLOYment

Manuel Mazzara, Cliff Jones, Alexei Iliasov

Overview

This paper reviews the major lessons learnt during two significant pilot projects by Bosch Research during the DEPLOY project [1]. Principally, the use of a single formalism, even when it comes together with a rigorous refinement methodology like Event-B, cannot offer a complete solution. Unfortunately (but not unexpectedly), we cannot offer a panacea to cover every phase from requirements to code; in fact any specific formalism or language (or tool) should be used only where and when it is really suitable and not necessarily (and somehow forcibly) over the entire lifecycle.

DEPLOY and its scenarios

DEPLOY is an ambitious project addressing diverse major industrial areas: automotive, train transportation, business and aerospace software. Putting all of them under the same hat is a difficult (if not impossible) task for both instrinsic and extrinsic reasons. We recognised from the very beginning that each deployment scenario would be different and would need (at least partially) different approaches, concepts and tools. Industries are different, development process differs, internal organizations are varied and, to some extent, business models are different as well as politics. Most importantly, target applications and in-house engineering tools/standards show little similarity. Therefore, integration needs are different.

The RODIN project [2] consituted a solid basis for DEPLOY giving us valid reasons to claim that the Rodin tools [3] were moderately mature. The tools were well designed by an outstanding team and the experience, including with industrial partners (although RODIN was a STREP rather than an IP), taught us that tool support is essential for most technology transfer undertakings. Although at the end of RODIN we were aware of the need to support the tools and their evolution, the reality we had to face in DEPLOY was tougher than expected for both academia and industry. Effort was immediately put in place to train industry partners via a "block course" to provide knowledge and skills leading to mini-pilots first, and then major case studies (in the case of Bosch specifically two: Cruise Control and Start/Stop system [4]).

Problems and opportunities

The Event-B notation favours a style of modelling that may be loosely characterised as event-based or reactive. There are examples where this works well and models appear natural and elegant. Unsuprisingly, there are also cases where there is a misfit between the style and developers' expectations or system requirements. This issue is more pronounced where an industrial user is involved. In an academic setting, there is a greater degree of flexibility of how a model is developed since the construction of a piece of software is rarely an end in itself. A researcher has the advantage of a broader perspective giving the ability to recognise, without investing considerable effort, that a method is not suitable for a given problem. In the DEPLOY project, few industrial users had prior exposure to formal methods and their perspective was limited to Event-B method and tools. Obviously, they also did not have an opportunity nor the desire to avoid, or adapt their problems to the Event-B style. This steadfastness, while at times frustrating for academic partners, has resulted in a stream of feature requests, some of which were, at least partially, addressed by offering methodological and tooling extensions.

Once beyond the stage of the mini-pilots, all the industrial users raised the issue of notation expressiviness. Many concepts found in programming and specification languages (records, procedure calls, macro definitions, modules, polymorphic types, meta-theorems, etc.) are not part of core Event-B. This is not a fundamental problem since the Rodin Platform is designed to be extensible and the core Event-B was purposely made compact. Still, the tool developers were surprised when industrial users indicated what they believed to be essential features missing in the Platform. The reaction took some time as effort had to be reallocated from already planned activities. In the meanwhile, industrial users still proceeded without these additional tools.

At least four tools were developed in response to Bosch team requests (group refinement, records, flow and team work) and some methodological work was inspired by the problems the Bosch team has encountered applying Event-B. With hindsight, the tool developers should have spent more time with the industrial partners compiling tool requirements. Initial version of these additional tools did not fit the industry expectations.

The transition from mini-pilots to larger case studies has identified a number of weaknesses in the tool implementation. The user interface did not cope well with machines comprising even a few dozens of events; this was further aggravated by hard to reproduce resource handling bugs. Such problems would, perhaps, not have been taken seriously for a long time were there not some commited users who constructed the first large-scale Event-B models. For the Bosch team, the main weak points of the tool turned out to be text editing; slow interactive prover UI; and liveness proof obligations in the form of gigantic disjunctions. The text editor, a suprisingly involved tool due to the Platform design, has improved immensily in the past year while the prover UI has been redesigned using a different rendering technology. There is no easy solution for proving complex liveness proof obligations but, in many cases, the SMT solving facility of the ProB plug-in has had spectacular success.

• *Manuel Mazzara, Cliff Jones and Alexei Iliasov are with the School of Computing Science, Newcastle University, Newcastle Upon Tyne, UK*

This work has been funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at www.deploy-project.eu.

These problems lead us to note a contrast between models from academia and industry. Research models are rarely large (and need not be for the focus is on challenging aspects that are best studied in isolation). On the other hand, an industrial development aims at a product and works from a real requirements document. This results in a large model which is often fairly "shallow" in the sense that many parts are merely descriptive and do not entail deep verification properties. It turns out that large models require a different treatment. As one example, in a terse, academic model, a message of a communication protocol would be some value $m \in M$. Message contents, like destination address or payload, could be added when necessary by defining mappings from M. But an industrial user deals with a detailed description of protocols where messages have dozens of attributes. Not only it is harder to find good abstraction and then carry out many (trivial but cumbersome) data refinement steps but the end result is far from elegant as a large number of variables are necessary to define what is conceptually a single entity. This specific issue has been addressed by the records plug-in but one can find many similar concerns stemming from the same problem of method, notation and tool scalability.

To make a development viable, a large model must be split at some point into sub-models. There are technical solutions for this but industrial partners have found them difficult to use. At the moment, formal model decomposition remains an art that requires skill and experience. To develop a real-life product, it is essential to be able to share the development effort among a team of modellers (this is partially addressed by the team work plug-in) and reuse existing modelling artefacts (to a limited extent addressed by the three decomposition techniques).

There are methodological issues arising from attempting large scale developments. It is not known how to plan a refinement strategy when faced with a detailed requirements document. It is impossible to foresee all the major design decisions which turn development into a trial and error. This itself would not be such a problem were it not so difficult and expensive (in terms of lost proofs) to refactor refinement chains in Rodin.

Achievements and Lessons

The intense collaboration with Bosch Research has been a valuable learning experience for both sides. We had the chance to approach several software engineering issues, contributing to some and, unavoidably, leaving others open. We believe our work has clarifed several aspects of industrial deployment of formal methods in automotive applications. Most important of all, we realized the limitations of Event-B, both as a formalism and as a method. The lack of a rigorous and repeatable approach of many other "formal methods" is well known. In [5] and [6] this issue is historically investigated and the requirements over a "formal method" are identified to discover that many methods are actually just notations, i.e. just formalisms without an attached rigorously defined and repeatable, systematic approach. Event-B is *not* one of those. Its refinement strategy was demonstrated to be useful when applied to several case studies in a number of projects like RODIN and DEPLOY. However, not even Event-B is a panacea applicable to every phase of software development. Instead of attacking every problem with a single weapon, we opted to use a portfolio of different instruments, which is an idea also supported by other researchers ([7], [8]). The overall strategy was demonstrated to be successful and, given the thorough documentation generated by the project ([9], [10], [4]), it promises to be repeatable by engineers with an initially limited knowledge of formal methods. However, the role of training cannot be underestimated and limitations of the current "knowledge transfer" approach have been identified. For example, the block course organized to train industry partners in 2008 was a valid choice, but more specific industry needs have emerged showing how closer and prolonged interactions are generally preferable. The documentation available at the beginning of DEPLOY had weaknesses, leading to the decision to generate a new user manual [11]. The importance of well written, high quality and complete documentation can never be emphasized enough (and Bosch, in particular, raised this point since the very beginning). The actual deployment consisted in formal modelling of two major relevant applications for Bosch: the Cruise Control and the Start/Stop system. Two different methodologies have been applied to the case studies as described in detail in [4], which also describes the motivations behind the choices. This process gave us a much better understanding of the links between requirements and Problem Frames and, in turn, the relationships with Event-B models.

Several lessons have been learnt during this intense and exciting experience. First of all, we had confirmation of other researchers' experience, i.e. the use of a single formalism cannot offer a complete solution to large problems. The primary importance of tool support stands not only in its mere existence, but in its ability to meet specific industry needs. Usability and performance, for example, have not been considered sufficiently before deployment and therefore became critical aspects in the process. Having tool support without users being able to actually use it is of little consolation. However, during the project several industry requirements have been satisfied thanks to the good feedback received. Finally, documentation and training are major aspects of deployment and crash courses do not seem to offer a solution. This is why "education strategies" should be implemented differently in future projects.

REFERENCES

[1] "Deploy: Industrial deployment of system engineering methods providing high dependability and productivity." [Online]. Available: http://www.deploy-project.eu/

[2] "Rodin: Rigorous open development environment for complex systems." [Online]. Available: rodin.cs.ncl.ac.uk

[3] "Event-B and the Rodin platform." [Online]. Available: www.event-b.org

[4] K. Grau, R. Gmehlich, F. Loesch, J.-C. Deprez, R. D. Landtsheer, and C. Ponsard, "Report on enhanced deployment in the automotive sector (d31)," DEPLOY Project, Tech. Rep. D38, 2011.

[5] M. Mazzara, "Deriving specifications of dependable systems: toward a method," in *Proceedings of the 12th European Workshop on Dependable Computing (EWDC)*, 2009.

[6] ——, "On methods for the formal specification of fault tolerant systems," in *DEPEND, International Conference on Dependability*, 2011.

[7] M. Broy, "Challenges in automotive software engineering." in *ICSE'06*, 2006, pp. 33–42.

[8] ——, "Seamless method- and model-based software and systems engineering," in *The Future of Software Engineering*. Springer, 2010, pp. 33–47.

[9] "Deploy deliverable d15: Advances in methodological wps."

[10] F. Loesch, R. Gmehlich, K. Grau, M. Mazzara, and C. Jones, "Pilot deployment in the automotive sector (d19)," DEPLOY Project, Tech. Rep., 2010.

[11] RodinHandbook, "User manual for Rodin (v2.3)," 2011, http://handbook.event-b.org.

Assessment of the Evolution of the RODIN Open Source platform

Christophe Ponsard, Jean Christophe Deprez, Jacques Flamand

Extended Abstract

The RODIN platform is an Open Source extensible Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. RODIN emerged out of the RODIN FP6 Project (2004-2007). It is still actively supported by the European Commission through the Deploy FP7 project (2008-2012) and the FP7 ADVANCED project (2011-2014). Over the years RODIN has known a growing success from a usage point of view, as show by its download statistics on figure 1. However in order to assess the long term sustainability of RODIN, it is important to pay a closer look at key Open Source characteristics of the project and of the community that formed around it.

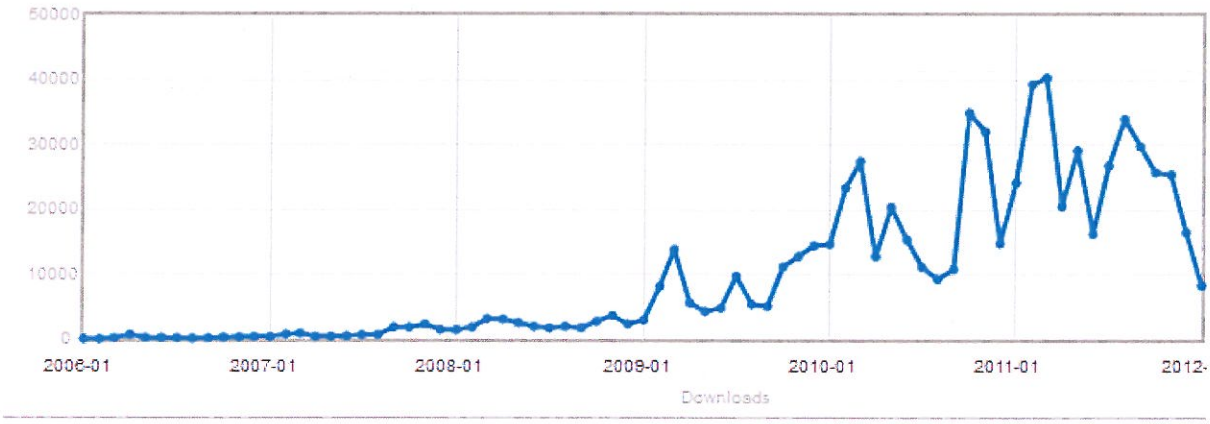


Figure 1. Evolution of the RODIN download statistics.

A number of assessment methods are available for this purpose include QSOS [1] and OpenBRR [2], sponsored by Atos Origin and Intel respectively[1][2]. However, they tend to be too light and do not really pay attention to the code. More elaborated methods were developed by the Qualipso [3] and QualOSS [4] projects. QualOSS will be considered here as it is oriented towards adoption. It is based on the notion of Free/Libre Open Source Software (FL/OSS) endeavour which is the undertaking of community members using tools and following software processes to produce work products related to one or more F/OSS components. More precisely the concern is about assessing the **robustness and evolvability**, based on a well-defined quality model.

In 2010, a first QualOSS assessment of the RODIN platform was produced [5]. It covered several dimensions such as maintainability, reliability, documentation, testability, community, change and release management. It globally revealed a good level of maturity but also pointed out a few risks in some areas like process management and community composition (see figure 2).

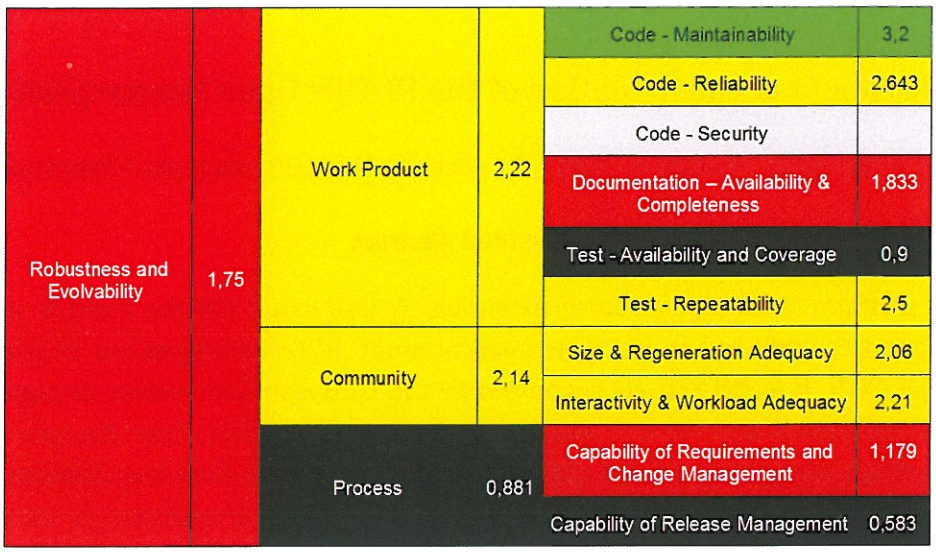


Figure 2. 2010 RODIN Maturity Assessment.

The objective of this talk is to present an updated assessment and show the evolution since the 2010 assessment, especially with respect to the previously identified risks. In addition to the core platform, the ecosystem of plug-ins that developed around the RODIN platform will also be investigated from that perspective. Finally as the workshop audience will gather key people with respect to the existing and future RODIN community, some concrete guidelines to start contributing or to improve the way of contributing to an Open Source project and more specifically to RODIN will be reminded.

References

[1] Method for Qualification and Selection of Open Source software (QSOS) version 1.6 © Atos Origin, April 2006 (http://qsos.org/).

[2] Business Readiness Rating for Open Source © OpenBRR.org, BRR 2005 – Request for Comment 1, 2005, (http://www.openbrr.org).

[3] Qualipso, Trust and Quality in Open Source Systems, http://www.qualipso.org

[4] Jean-Christophe Deprez, Kirsten Haaland, Flora Kamseu, The QualOSS Methodology, version 1, October 2008 - http://www.qualoss.org

[5] RODIN assessment on the Deploy Evidence FAQ, CETIC, 2010, https://forge.pallavi.be/projects/formalmethodevidence/wiki/TOOL-HM-1

A Rodin Plugin for automata learning and test generation for Event-B

Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu
University of Pitesti, Department of Computer Science
Str. Targu din Vale 1, 110040 Pitesti, Romania
{name.surname}@upit.ro

***Abstract*—In order to complement the current techniques supported by the Rodin platform, such as theorem-proving, composition, model-checking, for the purpose of test generation and state model inference from Event-B models, we developed a new feature as a Rodin plugin, which uses a model-learning approach to iteratively construct an approximate automaton model together with an associated test suite. Test suite optimization is further applied according to different optimization criteria.**

I. Introduction

This short abstract presents the implementation[1] in the Rodin platform of the general method described in our paper [1]. For a given Event-B model, the method constructs, in parallel, an *approximate automaton* model and a *test suite* for the system. The approximate model construction relies on a variant of Angluin's automata learning algorithm [2], adapted to finite cover automata [3]. A *finite cover automaton* [4] represents an approximation of the system which only considers sequences of length up to an established upper bound ℓ. Crucially, the size of the cover automaton, which normally depends on ℓ, can be significantly lower than the size of the exact automaton model. In this way, by appropriately setting the value of the upper bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be addressed. The proposed approach also allows for a gradual construction of the model and of the associated test suite, which fits well with the central notion of refinement in Event-B [5].

II. Model Learning and Test Generation

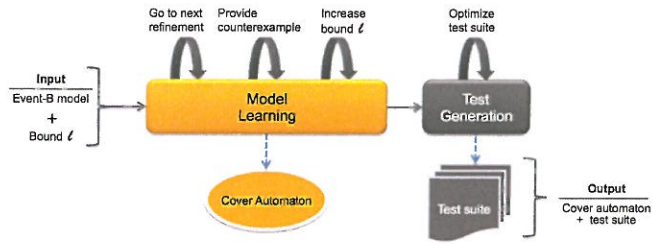


Figure 1. Overview of the implemented approach.

[1]Installation and screenshots at: http://wiki.event-b.org/index.php/MBT_plugin

The general overview of the tool is depicted in Fig. 1. We take as input an Event-B model M and a finite bound ℓ and we output a finite cover automaton approximating the set of feasible sequences of events of M of length up to ℓ and a test suite, i.e. a set of sequences including test data that make the sequences executable. The core procedure of "Model Learning" that generates a cover automaton is based on a variant of automata learning from queries [2], [3]. The cover automaton can be incrementally improved by providing more information according to the three loops in the figure. Thus one can (a) use the "next refinement" of the Event-B model that contains more information; or (b) one can "provide a counterexample" by manually or automatically providing sequences that are feasible in the Event-B model, but are not in the cover automaton or vice-versa; these counterexamples are used by the learning procedure to make the automaton approximating the Event-B model more precise; or (c) one can increase the bound ℓ and implicitly feed the learning engine with longer sequences which again will increase the precision of the finite state approximation.

At any point in the time, one can use the generated cover automaton to generate tests that exercise different sequences through the Event-B model. There are many existing methods for test generation from finite state models. In our case, we use internal information from the learning procedure, which maintains a so-called "observation table" that keeps track of the learned feasible sequences. Sets of feasible sequences in this table will provide the desired test suite. Note that during the feasibility check of the sequences in Event-B, test data are also generated. This is implemented by using the ProB constraint-solver [6] and is one of the most time consuming part of the algorithm. However, in some situations, the process of finding appropriate test data may last longer than a couple of seconds. To address this problem, we set a limit of 20 seconds on the time this process can last.

The test suite obtained satisfy strong criteria for conformance testing (usually required in the embedded system domain) and may be large. If weaker test coverage like state-, transition- or event-coverage are desired, optimization algorithms are applied on the test suite (see rightmost loop in the Fig. 1), producing a subset of the initial set of test cases that preserves the original test adequacy criterion

by removing the redundant test cases with respect to the considered optimization criteria. Our tool implements five of the six optimization problems proposed in [7], where the multi-objective test suite optimization problem for Event-B models was first introduced by using Multi-Objective Evolutionary Algorithms.

III. Experiments

For experimentation, we used a broad range of models, from the embedded systems, transportation and aerospace industries as well as academic and pedagogical Event-B models used in the literature. The Event-B models are publicly available in the model repository of the DEPLOY project at: http://deploy-eprints.ecs.soton.ac.uk. Table 1 shows the results for different refinements and ℓ's, i.e. no. of states of the generated cover automata, size of conformance test suites (i.e. before optimization) and the corresponding running times. Note that some of the models in the benchmark can be large exhibiting many events and variables, factors that increase the complexity of the procedure.

Table 1. Sizes and execution times for different refinement levels and ℓ's (no. of states, no. of test cases and exec. time)

Subject	M	ℓ	No. of states	No. of tests	Execution time
A2A	M2	11	5	79	0.06
	M3	11	8	285	0.52
	M4	11	8	445	0.77
	M6	11	14	1518	1.07
BepiColombo	M2	9	47	4798	71.87
	M3	12	126	24202	3543.01
	M2	11	48	5098	81.47
	M3	15	131	26958	4402.01
CarsOnBridge	M1	13	5	79	0.08
	M2	13	9	429	0.91
	M3	13	38	7407	81.26
CircArbiter	M4	12	5	141	0.51
Choreography	M0	13	5	120	0.47
	M1	13	10	957	10.45
MobileAgent	M2	12	13	445	1.72
	M3	12	25	1162	9.03
PressCtrller	M0	8	5	81	0.04
	M1	8	43	4857	31.82
	M2	8	127	22017	2545.58
ResponseCoP	M0	3	4	41	0.061
	M1	8	37	3230	28.57
	M0	4	5	51	0.10
	M1	9	49	5767	79.02
SSFPilot	M0	10	54	7541	116.87
	M1	10	96	20430	2371.34
	M1	11	107	25370	3652.08
TrainCtrller	M5	13	12	3117	33.83
	M6	13	12	4437	110.46
	M7	13	18	9054	451.11
	M8	13	20	11203	536.62

IV. Conclusion

We have briefly presented our approach for automata-learning and test generation for Event-B, which has been materialized in a concrete implementation as a plugin for the Rodin platform, making use of libraries such as Java Universal Network/Graph Framework (JUNG)[2] and jMetal[3] for automaton drawing purposes and multi-objective optimisation using genetic algorithms, respectively. We will continue to improve the tool on several dimensions, both by increasing its reach, e.g. by tackling not only refinement, but also different types of Event-B decompositions (shared-event or shared-variable), by improving the automation via model-checking for counterexample search, and by boosting its performance using parallelization (independent membership queries during the learning algorithm can be computed in parallel).

Acknowledgments

This work was supported by project DEPLOY, FP7 EC grant no. 214158, and Romanian Grant CNCS-UEFISCDI no. 7/05.08.2010.

References

[1] F. Ipate, I. Dinca, and A. Stefanescu, "Model learning and test generation using cover automata," 2012, submitted to IEEE Transactions on Software Engineering.

[2] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[3] F. Ipate, "Learning finite cover automata from queries," *Journal of Computer and System Sciences*, vol. 78, pp. 221–244, 2012, in Press.

[4] C. Câmpeanu, N. Sântean, and S. Yu, "Minimal cover-automata for finite languages," *Theoret. Comput. Sci.*, vol. 267, no. 1–2, pp. 3–16, 2001.

[5] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[6] M. Leuschel and M. J. Butler, "ProB: an automated analysis toolset for the B method," *Int. J. Softw. Tools Technol. Transf.*, vol. 10, no. 2, pp. 185–203, 2008, tool webpage: `http://www.stups.uni-duesseldorf.de/ProB`.

[7] I. Dinca, "Multi-objective test suite optimization for Event-B models," in *Proc. of Int. Conf. on Informatics Engineering and Information Science (ICIEIS'11)*, ser. CCIS, vol. 251. Springer, 2011, pp. 551–565.

[2] `http://jung.sourceforge.net`
[3] `http://jmetal.sourceforge.net`

VTG : Vulnerability Test cases Generator, a Plug-In for Rodin

Aymerick Savary, Jean-Louis Lanet,
Marc Frappier and Tiana Razafindralambo

January 16, 2012

This project aims to develop a plugin within the Rodin platform in order to generate vulnerability tests using Event B and ProB. Vulnerability testing consists in testing behaviors which should not be accepted by a system. In the context of this work, a vulnerability test is an event trace which contains at least one invalid event. In this first version of the tool, we want to generate traces containing exactly one invalid event. Our approach is to build an abstract model M of the system which describes only the valid behaviors. Then we pick one event to test, called the *event under test* (EUT), and add it to model M by negating its guard. Thus, the execution of this altered event will generate an invalid trace. We use ProB to generate traces, using temporal formulas to guide the search of traces. If one wants a trace that reaches a state s satisfying a conditions $c(s)$, we simply ask ProB to check whether the LTL formula $G(\neg c(s))$ holds; if it finds a counter example, then this counter example is a test case.

The crux of the tool is to generate negation of guards. The tool takes in input a set of rewrite rules for this purpose. There are several possibilities to generate the negation of a guard: for instance, $a \wedge b$ can be rewritten simply as $\neg(a \wedge b)$, but this does not necessarily result into "useful" test cases with a good coverage when ProB is used to find a trace. The rewrite rules allow the specifier to guide the generation of test cases by describing several variants of a negation, in order to obtain specific test cases; in essence, rewrite rules describe test criteria in a generic manner. For instance, $a \wedge b$ could be rewritten into three test cases using three rules: i) $a \wedge \neg b$, ii) $\neg a \wedge b$, iii) $\neg a \wedge \neg b$. Rewrite rules can be defined for each operator of the Event B language, including logical connectives and predicates (*e.g.*, $=$, $<$, $\in$, ...). Directives can also be added to indicate which part of a guard should be rewritten and which part should not, in order to avoid meaningless or unfeasible test cases.

The tool can take as input a model M and it iterates over each event of the model, applying all possible rewrite rules to the event's guard and calling ProB to generate traces for each possible rewriting of the event's guard.

We are using this tool to generate vulnerability tests for Java Card Byte Code Verifiers (JBCV). A JBCV is responsible for checking that JBC programs satisfy the security constraints of the Java specification. Thus, a vulnerability

test case for a JBCV is a JBC program that contains at least one JBC instruction violating the JVM specification. It can be generated using our tool by creating a machine M that contains one event for each instruction of the JBC language. The security constraints of the JVM specification are represented as event guards for JBC instructions. We have defined rewrite rules for this application domain, but most of these rules are general enough to be re-used in other domains as well. We use a single temporal formula to generate test cases: $G(\ not(\ \{eut = TRUE\ \&\ halt = TRUE\}\ )\ )$. Variable *eut* is set to *FALSE* in the initialization of the machine. The guard of the EUT checks that $eut = FALSE$ and sets it to *TRUE* in its actions, ensuring that the EUT is executed only once. Variable *halt* indicates that the JVM has reached a terminal state, thus the trace represents a complete JBC program.

This plugin enables the use of Event B and ProB to rapidly generate complex test cases using an abstract model of the system to test. This combination provides a simple and very flexible mechanism for generating vulnerability test cases for various application domains.

Visualisation of LTL Counterexamples with ProB

Andriy Tolstoy Daniel Plagge Michael Leuschel

January 16, 2012

The animator and model checker ProB [1] does support checking of Event-B models (among other specification languages) against LTL (Linear Temporal Logic) formulas [2].

If the model does not satisfy the property formalised by the LTL formula, it returns a counterexample. A major problem when using a the model checker in general and ProB in particular, is that it is very hard for a user to understand why the returned sequence of events is actually a counterexample to the specified property. Understanding the outcome of an LTL formula is usually difficult because the result depends on many possible states of the model. When several temporal operators are combined, the complexity of a formula is even harder to cope with.

To overcome that hurdle we developed an interactive graphical user interface to visualise LTL counterexamples. The new visualisation provides information on why the formula or parts of it evaluate to true or false. E.g. for a property like "always holds P" it highlights a state in the sequence where P does not hold. Similarly, for a property "eventually Q will hold" it shows a never-ending loop where Q never holds.

We present how the model checker can be used to validate temporal formulas. We show how the new visualisation (see Figure 1 for a screenshot) is integrated with ProB's other Rodin plug-in components and how they can be used to get a deeper understanding of the model. We will also briefly present a few other improvements that have recently been added to ProB's model checker, such as improved hashing and a limited form of state compression.

References

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

[2] D. Plagge, M. Leuschel: *Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more*, Software Tools for Technology Transfer (STTT), Volume 12(1), 2010

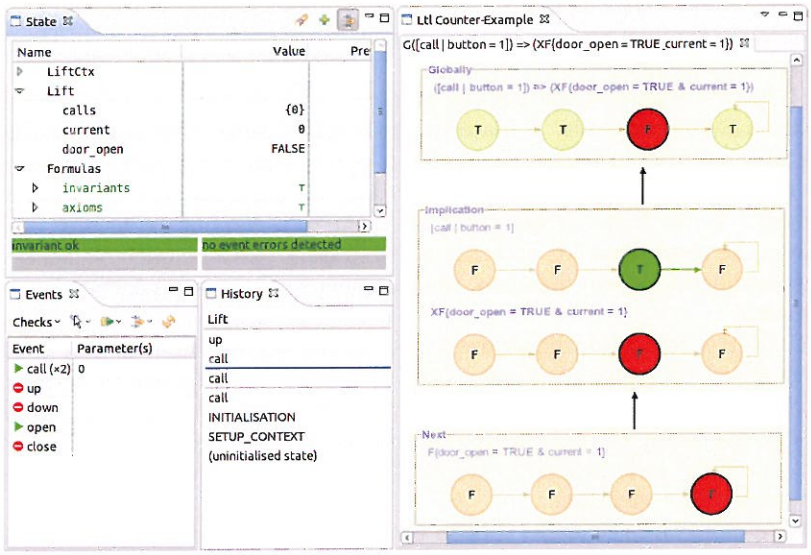


Figure 1: A screenshot of an LTL visualisation