Proceedings of the 6th Rodin User and Developer Workshop, 2016

Linz, Austria
May 23rd, 2016

Editors:

Michael Butler University of Southampton, UK.
Thai Son Hoang University of Southampton, UK.

UNIVERSITY OF Southampton

Part I

Summary

Executive Summary

Event-B is a formal method for system-level modelling and analysis. The Rodin Platform is an Eclipse-based toolset for Event-B that provides effective support for modelling and automated proof. The platform is open source and is further extendable with plug-ins. A range of plug-ins have already been developed including ones that support animation, model checking and UML-B. While much of the development and use of Rodin takes place within EU FP7 Projects (RODIN, DEPLOY, ADVANCE), there is a growing group of users and plug-in developers outside these projects.

The purpose of the 6th Rodin User and Developer Workshop was to bring together existing and potential users and developers of the Rodin toolset and to foster a broader community of Rodin users and developers. For Rodin users the workshop provided an opportunity to share tool experiences and to gain an understanding of on'going tool developments. For plug'in developers the workshop provided an opportunity to showcase their tools and to achieve better coordination of tool development effort.

The one-day programme consisted presentations on tool development and tool usage. The presentations are delivered by participants from academia and industry. This volume contains the abstracts of the presentations at the Rodin workshop on May 23rd, 2016. The presentations are also available online at `http://wiki.event-b.org/index.php/Rodin_Workshop_2016`.

The workshop was co-located with the ABZ 2016 conference and held in Linz, Austria. The Rodin Workshop was supported by the University of Southampton.

Finally, we would like to thank the contributors and participants, the most important part of our successful workshop.

Organisers

Michael Butler, University of Southampton
Stefan Hallerstede, Aarhus University
Thai Son Hoang, University of Southampton
Michael Leuschel, University of Düsseldorf
Laurent Voisin, Systerel

Contents

Programme of the Rodin Workshop 2016

09:00–10:30

- Meta-Predicates for Rodin - ***Sebastian Krings***
- A Rodin plug-in for constructing reusable schematic lemmas - *Alexei Iliasov,* ***Paulius Stankaitis****, David Adjepon-Yamoah, and Alexander Romanovsky*
- Event-B Specification Templates for Defining Domain Specific Languages - ***Ulyana Tikhonova***

10:30–11:00 Break

11:00–12:30

- Towards Modular Development in Event-B - ***Thai Son Hoang****, Hironobu Kuruma, and Michael Butler*
- Crossed-Project Reference for Managing Model Variations - *Hironobu Kuruma and* ***Thai Son Hoang***
- SliceAndMerge: A Rodin Plug-in for Refactoring Refinement Structure of Event-B Machines - ***Tsutomu Kobayashi****, Aivar Kripsaar, Fuyuki Ishikawa and Shinichi Honiden*

12:30–14:00 Lunch

14:00–15:30

- Rodin in the field of railway system engineering - ***Tomas Fischer***
- Building Event-B Interlocking Theories: Lessons Learned using the Theory Plug-in - ***Yoann Guyot****, Renaud De Landtsheer, and Christophe Ponsard*
- Theory plug-in for Rodin 3.x - ***Thai Son Hoang****, Asieh Salehi, Michael Butler, and Laurent Voisin*

15:30–16:00 Break

16:00–17:30

- Extending Code Generation to Support Platform-Independent Event-B Models - ***Asieh Salehi****, Michael Butler, and Colin Snook*

- Using Rodin and BMotionStudio for Public Engagement - ***Dana Dghaym***, ***Asieh Salehi*** *and* ***Colin Snook***
- Translating SCXML Statecharts to iUML-B State-machines - *Karla Morris and* ***Colin Snook***

Part II

Contributions

Meta-Predicates for Rodin

Sebastian Krings

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

1 Introduction and Motivation

Event-B [1] provides a concise mathematical language for specifying invariants and guards. While both Event-B and Rodin [2] matured, certain patterns for specifying properties like deadlock freedom emerged and are in use. These patterns are often realized by copy and paste of guards into other guards or invariants, leading to duplicated code and incomprehensible specifications. Furthermore, it may lead to errors if predicates are not kept in sync. We observed errors like these during the case studies of ABZ 2014 [3] and started developing an extension to Event-B that allows accessing guards by corresponding event names. Since, it has been implemented as a plugin for the Rodin platform.

2 Meta-Predicates for Event-B

We added the following meta-predicates based on the syntax accepted by the LTL model checker of ProB [5,4]. To avoid loops, generated predicates are not copied again. With $guards(e)$ containing the guards of e that have been given directly by the user, the added predicates are

- *deadlock* enforces that all given elements are disabled, i.e., $deadlock(Events) = \bigwedge_{e \in Events} \bigvee_{g \in guards(e)} \neg g$.
- *enabled* enforces that all given events are enabled, i.e., $enabled(Events) = \bigwedge_{e \in Events} \bigwedge_{g \in guards(e)} g$.
- *controller* enforces that exactly one event is enabled, i.e., $controller(Events) = \bigvee_{e \in Events}(enabled(\{e\}) \wedge deadlock(Events \setminus \{e\}))$.
- *deterministic* enforces that the order of execution is deterministic, i.e., $deterministic(Events) = controller(Events) \vee deadlock(Events)$.

Embedding the predicates into Event-B involves more work than macro replacement as the following example will show: Let's take events `evt1`, featuring parameter `x1`, and `evt2` with parameter `x2`. Obviously, we can not just copy guards of `evt1` to `evt2`, as referencing `x1` inside `evt2` will result in an undeclared variable. For free variables the plugin thus has to decide between

- Adding a parameter to the surrounding event,
- Introducing them using existential quantification, or

– Skipping, removing or renaming them as far as possible.

We did not want to change existing Event-B code in order to keep semantics as simple as possible. Only new invariants and guards should be added. Thus, we decided to introduce free variables by existential quantification. A preference allows to decide if all variables are quantified or only those not yet defined.

3 Integration into Rodin

Our extended language has been integrated into Rodin as a plugin. Sources are available from `https://github.com/wysiib/RodinMetaPredicatesPlugin`, `https://www3.hhu.de/stups/rodin/meta_predicates/nightly/` contains an update site.

The plugin works as an additional static checker. It runs after the machine itself has been checked, relying only on information Rodin has not discarded.

4 Comparison to Related Plugins

In [6], the authors present `DFT-generator`, a tool for the automatic generation of proof obligations for deadlock freedom. Using `DFT-generator` is comparable to adding $deadlock(\{\ldots\})$ to the invariants. However, rather than extending the predicate syntax, information is stored externally. Our approach is more flexible, e.g., one can use predicates like $state = ERROR \Rightarrow deadlock(\{\ldots\})$.

5 Discussion and Conclusion

Our plugin adds a lightweight language extension to Event-B. Together with tool support, common errors leading to erroneous specifications can be avoided. Furthermore, expressiveness is increased as the intention of a predicate becomes more obvious. In future, we would like to implement a direct export to ProB's LTL model checker to be able to use different model checking algorithms.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. 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(6):447–466, 2010.
3. F. Boniol, V. Wiels, Y. Ameur, and K. Schewe, editors. *ABZ 2014: The Landing Gear Case Study*, CCIS 433. Springer, 2014.
4. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME'03*, LNCS 2805, pages 855–874. Springer, 2003.
5. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008.
6. F. Yang and J.-P. Jacquot. An Event-B plug-in for creating deadlock-freeness theorems. In *Proceedings SBMF'11.* Brazilian Computer Society, 2011.

A Rodin plug-in for constructing reusable schematic lemmas

Alexei Iliasov, Paulius Stankaitis,
David Adjepon-Yamoah, Alexander Romanovsky

Newcastle University, UK

Abstract. In the paper we present an approach and tool for making proofs more generic and thus less fragile and more reusable. The crux of the technique is offering an engineer an opportunity to complete a proof by positing and proving a generic lemma that may be reused in the same or even another project.

1 Introduction

There was a concerted effort, funded by a succession of EU research projects, to make Event-B [1] and its toolkit, Rodin Platform [2], appealing and competitive in an industrial setting. One of the lessons of this mainly positive exercise is the general aversion of industrial users to interactive proof. It is possible, in principle, to learn, through experience and determination, the ways of underlying verification tools and master refinement and decomposition to minimise proof effort. The methodological implications are far more serious: building a good model is necessarily a trial and error process; one often has to start from a scratch or do considerable refactoring to produce an adequate model. This, obviously, necessitates redoing proofs and makes time spent proving dead-end efforts seem pointlessly wasted. Hence, proof-shy engineers too often do not make a good use of formal specification stage as they tend to hold on to the very first, often incoherent design. We want to change the way proofs are done, at least in an industrial setting. In place of an interactive proof - something that is inherently a one-off effort in Event-B - we want to invite modellers to write and prove a non-model specific condition called a schematic lemma that would, once added to hypothesis set, discharge an open proof obligation. Such a lemma may not refer to any model variables or types and is, in essence, a property supporting the definition of the Event-B mathematical language. If such a lemma cannot be found or seems to be to difficult to prove, the model must be changed. Since every modelling project is likely to have a fairly distinctive usage of data types and mathematical constructs, we might also expect a distinctive set of traits in supporting lemmas. We hypothesise that such distinctness is pronounced and dictated by the modellers experience and background as well as the model subject domain. We have also observed that the style of informal requirements - structured text, hierarchical diagram, structural diagram - has an impact on a modelling style. A schematic lemma is a tangible and persistent outcome of any

modelling effort, even an abortive one. Being generic, a schematic lemma is likely to be useful in a next iteration and, as we might hope, there is a point when all relevant lemmas are collected and modelling, in a given domain and for a given engineer, is nearly completely free of interactive proofs.

2 Generic lemmas plug-in

We have built a prototype implementation of the schematic lemma mechanism as a plug-in to the Rodin Platform. It integrates into the prover perspective and offers an alternative way to conduct an interactive proof either at a root node level or indeed for any open sub-branch of a proof obligation. At the moment, the notation employed is the native notation of Why3 but the first release will support entering schematic lemma in the Event-B mathematical notation.

The plug-in automatically constructs the first attempt at a schematic lemma through a simple syntactic transformation of a context proof obligation. All the identifiers occurring in either hypotheses or goal of the proof obligation are mapped into schematic lemma identifiers and then this mapping is used to translate hypotheses and the goal.

From this starting point it is up to the modeller to construct a sensible lemma by changing identifier, hypotheses and goal definitions. A prepared lemma is committed where the Why3 plug-in is used to prove that the lemma holds,and also that adding to the proof obligation in context discharges the proof obligation. If either fails, a user gets an indication of what has happened and it is not until both generic and concrete proofs are carried out that the schematic lemma may be used in the local library and assigned a binding level (machine, project or global). In the case of a success, the current open goal is closed.

To aid in the construction of a schematic lemma, the plug-in provides some simple productivity mechanisms. A hypotheses can be deselected without removing it to check whether both the lemma goal and the context proof obligation are still provable. An identifier may also be deselected and this automatically deselects all the hypotheses mentioning the identifier.

In this work we have tried to weave the process of constructing generalised proofs into the very process of model construction and address two long standing challenges of model-based design: turning proofs into tangible artefacts that can survive deep model refactoring, and making interactive proof on organic part of model construction rather than an unfortunate side activity.

References

1. J.-R. Abrial. *Modelling in Event-B.* Cambridge University Press, 2010.
2. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.

Event-B Specification Templates for Defining Domain Specific Languages

Ulyana Tikhonova

Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
u.tikhonova@tue.nl

Domain-Specific Languages (DSLs) are a central concept of Model Driven Engineering (MDE). They are considered to be very effective in software development and are being widely adopted by industry nowadays. A DSL is a programming language specialized to a specific application domain. It captures domain knowledge and supports reuse of such knowledge via common domain notions and notation. In this way, the DSL raises the abstraction level of solving problems in the domain. A DSL is usually implemented as a translation from the domain concepts to the programming language of an execution platform, such as C/C++ or Java. In our work we investigate how translation of a DSL to the Event-B formalism can facilitate design and development of the DSL, and support understanding and debugging of DSL programs.

The Rodin platform offers various supporting tools for Event-B that can be applied to a DSL specification in order to implement a set of dedicated use cases. For example, the DSL semantics can be prototyped and then analyzed using automatic provers and model checkers (provided by AtelierB and ProB plug-ins); DSL programs can be simulated and debugged using animators and visualization tools (ProB and BMotion Studio). Although providing an extensive tool support, Event-B is not designed for specifying the DSL semantics. Therefore, in order to realize practical benefits of applying Event-B to a DSL, we adopt the specification formalism to the MDE context through the following model-to-model transformations.

- The *DSL-to-Event-B* transformation automatically generates an Event-B specification of an arbitrary DSL program from a set of Event-B specifications that define the DSL semantics on the meta-level [3]. Such a transformation composes the resulting specification out of the DSL semantic specifications using the technique of *shared event composition* [2].
- The *DSL-to-BMotion* transformation automatically generates a visualization for each concrete DSL program following (mimicing) the DSL graphical notation. The resulting visualization runs in the BMotion Studio together with the ProB animator and provides a graphical user interface (GUI) for the Event-B specification being animated.

As a result, DSL end-users do not need to know formal notation of Event-B to create and run specifications of their programs. We validated this approach by means of a *user study* performed in the industrial context with the real-life mature DSL. The interviewed users confirmed that the DSL-based development

process will benefit from having such harnessed specification of the DSL. However, the users indicated that to realize these benefits one needs to keep the Event-B specification of the DSL consistent with the actual implementation of the DSL, following all its changes and updates. The latter requirement might cause a high overhead of applying the DSL-to-Event-B transformations, as they realize rather complex translation of the high-level DSL concepts to the low-level Event-B concepts. In other words, the semantic gap between a DSL and Event-B is quite wide.

To manage the wide semantic gap between the DSL and Event-B and to generalize the proposed approach so that it is applicable to other DSLs, we introduce *Constelle* – an intermediate language of *reusable specification templates*. An arbitrary DSL can be defined in Constelle using specification templates, which have been identified as reusable (successful) design solutions and stored in a library. On one hand, specification templates can be (re)used to define different DSLs. On the other hand, such a definition captures intentions of the DSL developer in a more clear way.

Constelle builds on top of the Event-B formalism. This means that all specification templates are implemented in Event-B. And the *Constelle-to-Event-B* transformation automatically generates Event-B specifications of a DSL from its Constelle definition. For this, the Constelle-to-Event-B transformation specializes the invoked templates using *generic instantiation* [1] and weaves them together using *shared event composition* [2]. In this way we ensure that specification templates are reused together with the proof obligations discharged for them.

The Constelle language is not restricted to the scope of DSLs. Our vision is that a library of Event-B specification templates can be shared and filled by Event-B practitioners working in various domains. Moreover, in future specification templates can be used as a source for automatic generation and/or for semi-automatic configuration of other artifacts. For example if each specification template is coupled with a visualization template for the BMotion Studio, then a specialized visualization for a DSL can be generated from the Constelle definition. If each specification template is coupled with a source code in C/C++, then we might be able to generate the corresponding DSL implementation in source code.

References

1. R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In K. Breitman and A. Cavalcanti, editors, *11th International Conference on Formal Engineering Methods, ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.
2. R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects (FMCO)*, pages 122–141. Springer, 2010.
3. U. Tikhonova, M. Manders, M. van den Brand, S. Andova, and T. Verhoeff. Applying Model Transformation and Event-B for Specifying an Industrial DSL. In *MoDeVVa@MoDELS*, pages 41–50, 2013.

Towards Modular Development in Event-B

Thai Son Hoang[1], Hironobu Kuruma[2], and Michael Butler[1]

[1] ECS, University of Southamtpon, U.K.
[2] Research and Development Group, Hitachi Ltd., Japan

Background. Event-B [2] developments are mostly structured around *refinement* and *decomposition* relationships [3]. This top-down development architecture enables system details to be gradually introduced into the formal model. More often, this result in large model with monolithic structures.

Motivation. Various composition approaches have been proposed [6, 4, 7, 5]. This proposal is inspired by these approaches and development methods such as classical-B [1], working towards modular development in Event-B.

Machine inclusion. Our first concept is *machine inclusions*. Machine A that includes machine B inherits B's variables and invariants. Variables of B cannot be modified directly, but only through event's synchronisation [7]. Multiple instance of the included machine can be achieved via prefixing, similar to [5]. The syntactical "flatten" of A can be seen in Fig. 1.

machine B
variables y
invariants $J(y)$
events
event f
any u where
$G_B(y, u)$
then
$BAP_B(y, u, y')$
end

machine A
includes p_B
variables x
invariants $I(x, p\_y)$
events
event e
synchronises p_f
any t where
$G_A(x, t)$
$H_{AB}(x, p\_y, t, p\_u)$
then
$BAP_A(x, t, x')$
end

machine (flatten_)A
variables x, p_y
invariants
$I(x, p\_y)$
$J(y)$
events
event e
any t, p_u where
$G_A(x, t)$
$H_{AB}(x, p\_y, t, p\_u)$
$G_B(y, u)$
then
$BAP_A(x, t, x')$
$BAP_B(p\_y, p\_u, p\_y')$
end

Fig. 1: Machine inclusion

Refinement-chain inclusion. While machine-inclusion mechanism gives us a direct reuse of machine, it is often the case that we want to reuse a refinement-chain. Approaches such as [4, 8] allow to incorporate refinement chains in to

the development. However, these approaches involve generating of model which is often cumbersome to accomodate changes. We propose to to include the refinement-chain inside the machine itself. Fig. 2 illustrates a situation where A includes a refinement chain from B_1 to B_m. Semantically, A has double "in-

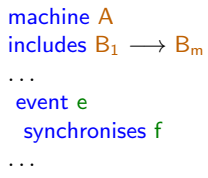


Fig. 2: Refinement-chain inclusion

terfaces". To its abstract machine, it acts as a machine with an inclusion of B_1. To its concrete machine, it acts as a machine with an inclusion of B_m. Since the refinement of B_1 by B_m has been proved separately, the refinement of $(A + B_1)$ by $(A + B_m)$ is almost correct-by-construction. The extra manual work required is the refinement of the extra guard, e.g., H_{AB} in Fig. 1.

Some evaluation. We have applied the idea (manually) to several examples. The result is quite encouraging with reduction of the modelling and proving efforts. In particular, this approach can be used several times, e.g., to have a hierarchy of inclusions. The resulting models also easier to comprehend and in particular, it should incorporate with changes to the imported model without any additional effort.

Conclusion. The concept here is not new and incorporate many existing work. We consider this as an effective way to have modular development in Event-B which will reduce the modelling and proving efforts. We are investigating how to extend Rodin to support the approach in the most efficient way.

References

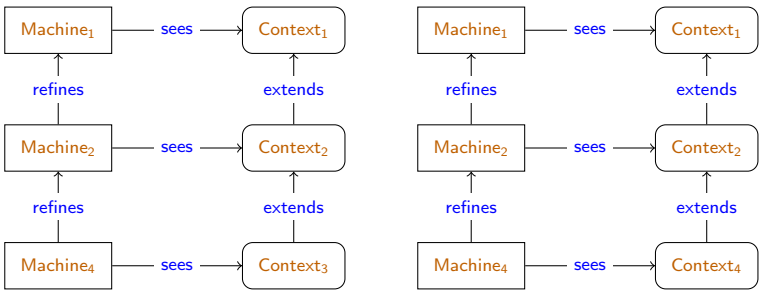
1. J-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
3. J-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
4. Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. *Software and Systems Modelling*, 12(2):229–244, May 2013. `http://dx.doi.org/10.1007/s10270-010-0183-7`.

Crossed-Project Reference for Managing Model Variations

Hironobu Kuruma[1] and Thai Son Hoang[2]

[1] Research and Development Group, Hitachi Ltd., Japan
[2] ECS, University of Southamtpon, U.K.

Background. A model in Event-B [1] (typically located within a project) is composed of components combined using refines, extends or sees relations. Fig. 1 shows an example of model variations containing some common machines and contexts, i.e., $\mathsf{Machine}_1$, $\mathsf{Machine}_2$, $\mathsf{Context}_1$ and $\mathsf{Context}_2$. $\mathsf{Machine}_3$ and $\mathsf{Context}_3$ (resp. $\mathsf{Machine}_4$ and $\mathsf{Context}_4$) are additional component specific to project **A** (resp. **B**).



(a) Project **A** (b) Project **B**

Fig. 1: A family of models

Motivation. To maintain these models, it is desirable to separate the common components and to share them between the projects. In Fig. 2, project **C** is for the shared components, where the original projects **A** and **B** contain only the additional components. This structure is beneficial for composing a family of models that has common properties by sharing components.

Concept. We experimentally introduced a crossed-project reference mechanism into the RODIN platform [2] to manage collections of components and enable reference to components in different projects. The crossed-project reference mechanism uses a manifest to identify components to be imported from other projects (Fig. 2). The names of imported components are prefixed with their project name. The components in importing project refer the imported components by their names declared in the manifest. The imports relation between projects must be acyclic.

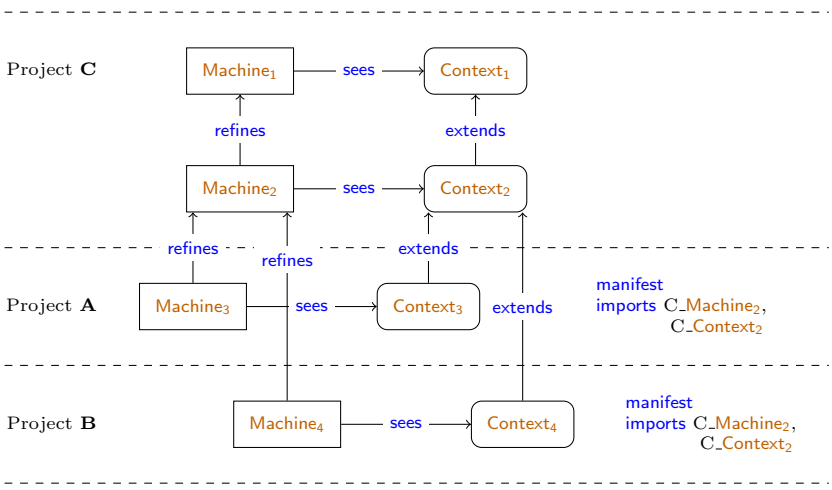


Fig. 2: Crossed-project Reference

Implementation. We implemented the cross project reference by *renaming and copying the statically checked files* of imported components into the importing project. In our example, two files in **C**, i.e., the statically checked files of $\mathsf{Machine}_2$ and $\mathsf{Context}_2$, are renamed to by prefexing, and copied into **A** and **B**. Since the imported components are expected to be verified in their source project, verification is required only for the additional components.

Conclusion and Limitation. The crossed-project reference mechanism reduces the risk of unintended modification of components since it separates the components into hierarchical collections. This property is useful for managing model variations. However, our implementation is experimental and insufficient for practical usage. Most RODIN plug-ins refer the unchecked file of components and are incompatible with our implementation. For example, the components that refer imported components cannot be easily edited by ordinary editors because they do not recognize the imported components. Although we expect the generic instantiation [3] is an effective way to compose variations, the generic instantiation plug-in also uses the unchecked files and does not work with the crossed-project reference. Furthermore, since our implementation does not propagate the change of original components, the imported file must be updated manually in the importing project. The propagation of the change of components and the collaboration with the generic instantiation are our future work.

References

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

SliceAndMerge: A Rodin Plug-in for Refactoring Refinement Structure of Event-B Machines

Tsutomu Kobayashi[1], Aivar Kripsaar[2], Fuyuki Ishikawa[3] and Shinichi Honiden[1,3]

[1]The University of Tokyo, Japan
[2]RWTH Aachen University, Germany
[3]National Institute of Informatics, Japan

Event-B supports flexible and rigorous modeling with a refinement mechanism based on proof obligation rules. The flexibility enables developers to decide a set of concepts and aspects of target systems focused on in each refinement step. The decision is important in modeling, because it has effects on understandability, maintainability, extensibility of Event-B models.

We focus on refactoring of refinement structure of machines, or re-deciding concepts and aspects of refinement steps in existing proved machines. We aim to realize it in a flexible way through slicing (decomposition) and merging (composition) of refinement steps.

In concrete, when the followings are given:

- Machines M_A and M_C such that M_C refines M_A
- V_B, which is a subset of M_C's variables

Slicing of the refinement produces an *intermediate machine* M_B such that:

- M_C refines M_B [1] and M_B refines M_A
- M_B's set of variables is a superset of V_B

Merging is the reverse operation of slicing.

Refinement slicing decomposes introduction of new variables and invariants through a refinement step into several steps. Moreover, it often reveals implicit properties of a concrete machine as explicit expressions of an intermediate machine. Therefore, it helps users to understand descriptions and proof of refinement steps, and thus improves maintainability. A typical problem that can be solved by refinement slicing is a refinement step with a large number of new variables and invariants, which tend to be difficult to understand.

Although Rodin platform supports refactoring expressions of machines and contexts, refactoring of refinement structures has not been tackled.

[1] **refines** clauses of events in M_C are changed through this process.

We implemented a plug-in of Rodin platform named SliceAndMerge to support slicing and merging.

Our approach to slicing is as follows:

1. Calculate variables in the intermediate machine considering dependencies of variables and invariants.
2. Find fragments of M_A and M_C that should also be specified in M_B.
3. Add complementary specifications to the fragments, so that M_B becomes consistent.

SliceAndMerge supports users by automating step 1 and step 2 of the above. Users of SliceAndMerge can select a part of invariants of an intermediate machine M_B from a list of invariants of a concrete machine M_C (Fig. 1). Then the tool resolves dependencies between invariants and variables, finds fragments of M_A and M_C that should be included in the intermediate machine M_B, and generates a part of M_B. The tool also supports merging of refinement steps.

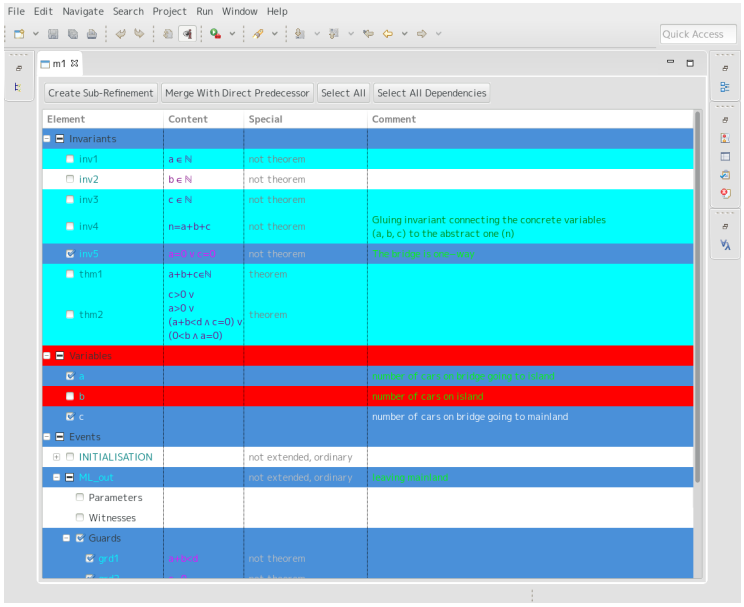


Figure 1: Interface for selection of invariants in intermediate machine

In this tool development presentation, we describe details of SliceAndMerge and give a demonstration of the tool.

Rodin in the field of railway system engineering

Tomas Fischer

Thales Austria GmbH, Handelskai 92, 1200 Vienna, Austria,
tomas.fischer@thalesgroup.com

Abstract. Railway signaling systems are required to provide the highest safety level due to the risk of loss of human life. Formal methods can contribute to the reliability and robustness of specification, design and implementation of such systems and support their verification and validation. Moreover, the CENELEC standards [1–3], which define the certification process of safety critical applications in the railway domain, qualify the use of formal methods as highly recommended.
However, the application of formal methods is very labor intensive, and thus expensive. A formal model of the respective system has to be created, maintained over the long product lifespan (25+ years) and reasoned about not only during the development phase, but recurringly at each modification of the evolving specification. Qualified experts with versatile skills are inevitable for this job. These experts are required to transform a semi-formal domain model (as understood by the domain experts) into the mathematical one and keep both of them aligned. They are also expected to interpret the verification and validation results and to communicate them to experts from other domains. These tasks demand a good tool support assisting users in all development phases with the aim to reduce manual efforts and thus decrease overall costs (see also [4]).
In the industrial context formal methods can be used as a one-time shot (e.g. prove the correctness of one particular algorithm) or continuously, as an integral part of the development process and therefore integrated into the development toolchain. Whilst the former usage has already been studied well and there are some very promising results available, the latter one encounters several obstacles.
At the workshop we present our experiences with introducing formal methods (in particular Event-B with the Rodin toolset) for the development of a railway interlocking system and discuss our current technical maturity assessment of the Event-B tools. On a small model we demonstrate the identified mismatch between the engineering demands and business needs on one side and current state of the Rodin toolset on the other side. Finally we propose some measures how to increase Rodin's usability (and hence the productivity) without sacrificing its profound theoretical foundation.

References

1. CENELEC, E.N.: 50126-Railway Applications: The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS). European Committee for Electrotechnical Standardization (1999)

2. CENELEC, E.N.: 50129-Railway Applications: Communication, signalling and processing systems - Safety related electronic systems for signalling. European Committee for Electrotechnical Standardization (2003)
3. CENELEC, E.N.: 50128-Railway Applications: Software for Railway Control and Protection Systems. European Committee for Electrotechnical Standardization (2011)
4. Romanovsky, A., Thomas, M.: Industrial deployment of system engineering methods. Springer (2013)

Building Event-B Interlocking Theories: Lessons Learned using the Theory Plug-in

Yoann Guyot, Renaud De Landtsheer, Christophe Ponsard
CETIC Research Center, Charleroi, Belgium
{yoann.guyot, renaud.delandtsheer, christophe.ponsard}@cetic.be

A computer-based interlocking is a railway signalling system that automatically controls the objects of a railway network, such as signals or points, in order to let trains move on its tracks without colliding with other trains nor derailing. A number of approaches based on the Event-B method [1] have already been proposed to model and prove interlocking systems [2, 3]. However its use in the industrial world is still a challenge. First, engineers responsible for specifying these systems are generally not fluent in using formal methods such as the Event-B method. Representing railway specific concepts such as routes, tracks, signals or points using primitive Event-B constructs such as sets, relations and functions quickly leads to models that are both hard to understand and manage. Second, the loss of the rail structure of the problem makes the proof obligations difficult to discharge, hence requires a lot of manual proof-work, also restricting its use to specialists.

A relevant way of making the approach practical for railway engineers is to raise the level of abstraction from the set theory of Event-B to the interlocking domain and also to provide efficient, yet generic enough, proof automation at this level. This can be achieved using the Theory plug-in of the Rodin Platform for system modelling in Event-B [4].

This talk presents our experience and discusses a number of open questions on the use of the Theory plug-in in the context of a work-in-progress aiming at defining a set of interlocking theories ready to be used by signalling engineers. These theories are made of train-specific constructs with a set of theorems and proof rules. They form a domain specific language (DSL) for modelling interlocking systems that has a fully formal semantics enabling to carry out verification activities but also to perform animation and to generate interlocking systems form the model based on a number of standard Rodin plug-ins [5].

Based on the well-known train example of [1], we progressively factorized key domain concepts such as blocks, routes, trains, points, and signals into new theories. Inspired by [6], we have first defined theories for manipulating chains and subchains, in order to be able to express routes properties and relations in a dedicated theory for routes. We notably introduced a new route reservation theory which lets the user manipulate the set of all possible route reservations in the modelled network by providing operators such as:

- **validRouteRes** to select the set of all valid route reservations on a network made of the given blocks and routes

- **compatibleRoutesOnly** to guarantee that two incompatible routes will never be reserved at the same time
- **res_blocks** to get the set of reserved blocks of a given route
- **isReserved** to check whether a given route is reserved.

On the proving side, the definition of theorems partly automates the discharging of proof obligations and also cuts down the effort of manual proving thanks to the ability to reason at the domain level. However, we are still facing a major challenge related to the number of proofs because about only forty percent of the total proof obligations are currently discharged automatically in our model as well as in our theories. In the last part of the talk, we highlight possible directions to tackle this, such as enriching the theorems and defining proof tactics. This might also initiate some discussion about future developments of the Theory plug-in itself.

Acknowledgment

This work was partly funded by the Walloon Region under the INOGRAMS project (grant nr 7171).

References

[1] Jean-Raymond Abrial. *Modeling in Event-B.* Cambridge University Press, 2010.

[2] Minh-Thang Khuu, Laurent Voisin, and Luis-Fernando Meija. Modeling a Safe Interlocking Using the Event-B Theory Plug-in. *Proceedings of the 5th Rodin User and Developer Workshop*, 2014.

[3] Michael Leuschel, Jens Bendisposto, and Dominik Hansen. Unlocking the Mysteries of a Formal Model of an Interlocking System. *Proceedings of the 5th Rodin User and Developer Workshop*, 2014.

[4] Michael Butler and Issam Maamria. *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, chapter Practical Theory Extension in Event-B. Springer Berlin Heidelberg, 2013.

[5] Rodin Community. Rodin Plug-ins. http://wiki.event-b.org/index.php/Rodin_Plug-ins.

[6] Luis-Fernando Mejia, Minh-Thang Khuu, and Asieh Salehi. Application in Railway Domain. *ADVANCE Project Deliverable*, 2014.

Theory Plug-in for Rodin 3.x

Thai Son Hoang[1], Asieh Salehi[1], Michael Butler[1], and Laurent Voisin[2]

[1] ECS, University of Southamtpon, U.K.
[2] Systerel, France

In Rodin 3.0, there are major changes within the Rodin Core. In particular, the following changes directly effect the Theory plug-in (`http://wiki.event-b.org/index.php/Rodin_Platform_3.0_Release_Notes`).

- **Stronger AST Library**: The API of the AST library has been strengthened to mitigate risks of unsoundness when mixing several formula factories. Now, every AST node carries the formula factory with which it was built, and the AST operations that combine several formulas check that formula factories are compatible.
- **Stronger sequent prover**: In order to improve the reliability of the proof status when working with mathematical extensions, the reasoners can be declared as context-dependent. The proofs that use a context dependent reasoner will not be trusted merely based on their dependencies, but instead they will be replayed in order to update their status. This applies in particular to Theory Plug-in reasoners, that depend on the mathematical language and proof rules defined in theories, which change over time.

Several problems have been reported for the Theory plug-in (with Rodin 3.0)

- Exceptions when openning proof obligation.
- Exceptions when applying rule-based prover reasoners, e.g., **XD** (expand definition).
- Changing the model, e.g., proof rules, theory path has no effects on existing proofs.
- Losing proofs when saving (the problem is in fact in loading previously saved proofs) .

The theory plug-in and its associate rule-based provers need to be upgraded to accommodate the changes in the Rodin Platform core.

We have been working in the last few months on ensuring compatibility of the Theory plug-in with the Rodin Platform core.

- The matching facility has been upgraded to use the *ISpecialization* API (insteads of *IInstantiation*) which allows to specialize types consistently. This fixes several exeptions when applying reasoners.
- Improving the matching facitlity for *associative operators*.
- Implement equality between datatype/operator extensions to ensure that the formula factory will assign the same "ID" for datatype/operator with the same definition. This ensures that formula factories coming from different origins can be correctly compared. In particular, this ensures that proofs can be loaded with the correct formula factory.

The upgrades are required some fixed in the Rodin Core hence will be available after the next release of the Rodin Platform (Rodin 3.3?)

We have identified the following future research/improvement for the Theory plug-in.

- Support for *infix predicate operators*.
- Support for *predicate variables* in theory.
- Improve matching facility for *associative commutative operators* (currently does not take into account commutativity).
- *Tactics* for theory
- Theory *instantiation*
- Improving *usability* of the theory plug-in, e.g., performance in interactive proofs.

Some of the improvements are straightforward while the others, e.g. tactics, instantiation, requires further investigation.

Extending Code Generation to Support Platform-Independent Event-B Models

Asieh Salehi, Michael Butler and Colin Snook
University of Southampton

6th Rodin User and Developer Workshop
May 23*th*, 2016, Linz, Austria

Summary:
Code generation was introduced in the Event-B formalism to address the gap between the lowest level Event-B refinement and an implementation. However, the code generation supports generating a single implementation for a refined Event-B model. This results in dependency between the Event-B model and target platform architecture.
To address this limitation, we present an extension of the Event-B code generation technique supporting generation of different platform-specific implementations from the same Event-B model. A refined Event-B model is treated as platform-independent through parameterisation. The platform parameters are instantiated in order to generate a platform-specific implementation and these are used by the code generator to produce an implementation that is tailored to the platform.
We applied our approach to model an embedded Run-Time Management (RTM) system; and generated three different RTM implementations for three hardware platforms with different specifications.

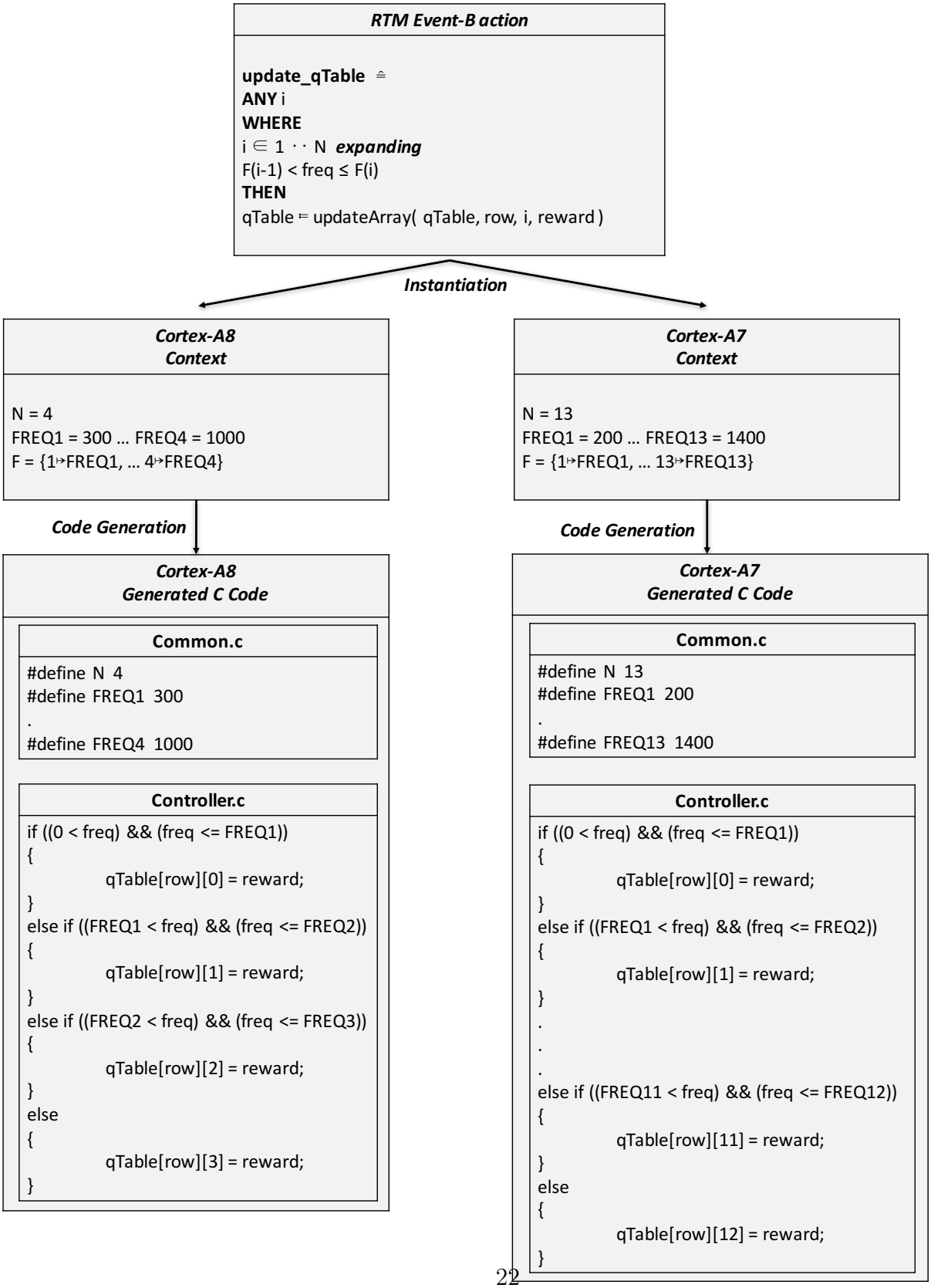
Motivation:
The figure presents generation of part of the RTM implementation for two ARM hardware platforms: Cortext_A8 and Cortex_A7, from a platform-independent Event-B action. The *update_qTable* event updates the look-up table used in the machine learning algorithm during run-time. The number of qTable columns depends on the number of frequencies each platform supports. The platforms support different number of frequencies (N); The Cortext_A8 supports 4 frequencies, whereas Cortex_A7 supports 13 (frequency values specified as constants *FREQi)*. There is a guard of the update_qTable event indicated as an expanding guard. Variable *N*, used in the expanding guard, is instantiated during code generation and results in generating a collection of N conditional branches in C to modify the qTable with N columns.

References:
[1] PRiME: Power-efficient, Reliable, Many-core Embedded systems. http://www.prime-project.org.

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

[3] Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES, 2011

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



Using Rodin and BMotionStudio for Public Engagement

Dana Dghaym, Asieh Salehi and Colin Snook
University of Southampton, Southampton, United Kingdom

As part of its public engagement activities the University of Southampton's Faculty of Physical Science and Engineering holds an annual 'Science Day' when the campus is open to the public and researchers demonstrate aspects of science related to their research. Many aspects of science are demonstrated and the event is very popular and well attended. For Science Day 2016 we used Rodin and Event-B to demonstrate how mathematics can help to analyse problems. The Science Day is primarily targeted at children up to year 11 but many older siblings also attend and we also wish to engage with parents. The day is advertised within local schools and naturally appeals to teachers. The event is therefore a good opportunity to initiate on-going engagement with children via their schools. Therefore, the demonstration must be designed to appeal to, and be accessible to, people of many ages and mathematical abilities from young children through to professional scientists and mathematicians. We used BMotion Studio to provide two simultaneous visualisations. The first visualisation was a cartoon style representation of the real-world problem designed to appeal to young children and not requiring any mathematical abstraction skills. The other visualisation was a simple Venn diagram representation of the sets and counters involved in the mathematical model which older children and adults could easily follow. For the younger children we would point out the mathematical representation to make them aware of it but not attempt to explain it unless they were interested. For those that appeared to be particularly adept and interested in the underlying mathematical system we gave a brief overview of the Event-B model and verification by proof.

We designed a simple safety related problem based on parking two cars in two parking bays with an unprotected crossing and bays protected by signal. The children were given a scenario for a particular car to park in a bay and asked to select from a list of conditions that needed to be satisfied for that scenario to be safe. They were then asked to configure an Event-B context to reflect their selected conditions using a purpose built editor. The configuration used boolean constants to enable guards in the corresponding machine. If the correct guards were selected the scenario could be performed safely. That is, any attempt to crash the two cars by moving them to the same location would be prevented by the model. If important conditions were not selected cars could be crashed and if superfluous guards were enabled the scenario could, in some cases, not be completed. We gave a toy car to any child that completed the exercise.

The exercise was very popular throughout the day and at times children queued to try their selections. In all, two hundred children performed the exercise. Several teachers commented on how useful they thought the exercise was for the children. Parents enthusiastically encouraged their children and chatted to us about the underlying research.

It was interesting to note that children often interpreted the problem in different ways to us. For example selecting that the unused parking bay should have a red light. Their reasoning was that the car might otherwise enter the wrong bay. While recognising that we had not completely specified the

requirements (in terms of behaviour of cars), we pointed out that, according to our intended requirements, they had designed a safe system but not a very useful one. Another interesting interpretation was that, in the case where several scenarios were attempted, the previous scenario may result in a bay being already occupied, so some children automatically changed the scenario and set the lights to send the car to the other parking bay. We had not intended the scenarios to be interpreted sequentially.

The children seemed to take the exercise too seriously, perhaps seeing it as a test and believing that they would not get a prize car if they allowed the cars to crash. For example in some cases they were reluctant to test their selection by trying to crash the cars. Possibly we should avoid associating success with the system being safe and emphasize exploration and understanding of the problem.

In conclusion the demonstration was a huge success at engaging public interaction with Event-B modelling and we hope to build on this in future by developing other model-based problems and interacting with local schools.

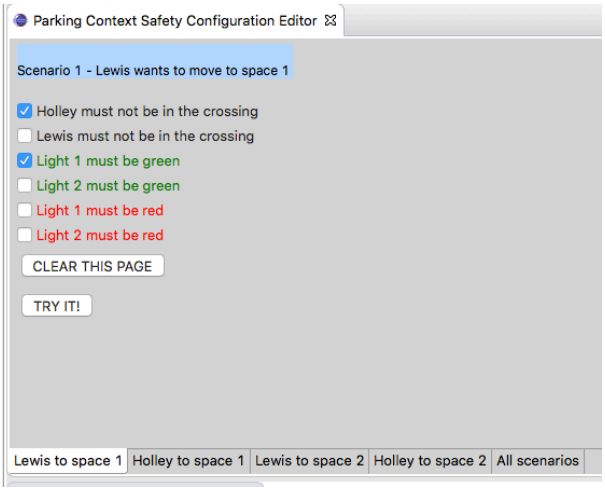


Figure 1 - Purpose built context editor for entering selections

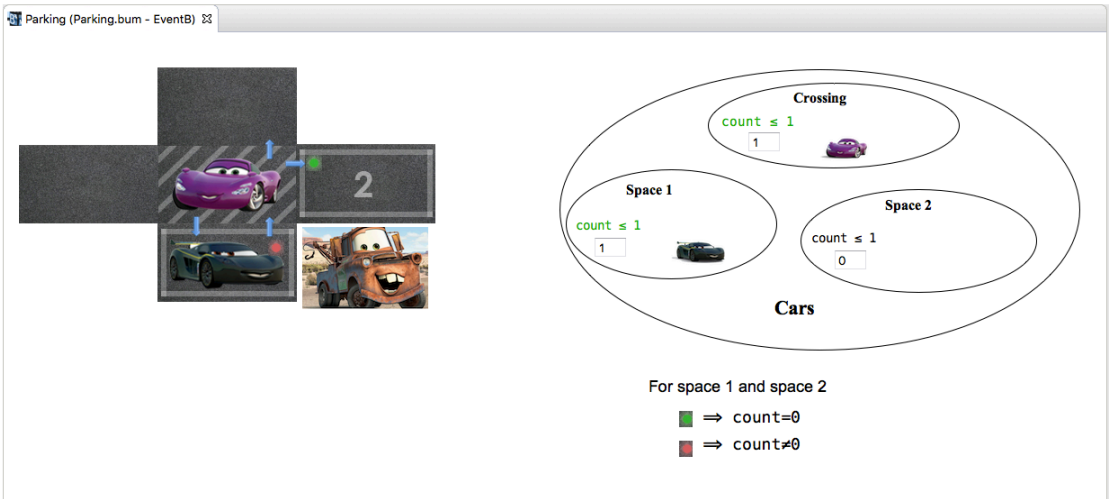


Figure 2 - BMotion Studio visualisations
(cars are moved by clicking the blue arrows)

Translating SCXML Statecharts to iUML-B State-machines

Karla Morris[1] and Colin Snook[2]

[1] Sandia National Laboratories, Livermore, California, U.S.A.
knmorri@sandia.gov
[2] University of Southampton, Southampton, United Kingdom
cfs@ecs.soton.ac.uk

To facilitate automatic proof, the Event-B notation is restricted to a simple guarded action behaviour. While iUML-B goes some way to provide an intuitive state-transition representation of Event-B models, its notation follows the semantics of Event-B. Engineers that are used to the richer semantics of Harel style statecharts may find these restrictions difficult to accept. Conversely, Event-B has features such as refinement and invariant properties that are not considered in most statechart notations. It may be cumbersome for engineers to re-model existing systems into iUML-B for verification.

In order to explore the feasibility of this model transformation, we have developed a translation from a statechart representation, *State Chart XML: State Machine Notation for Control Abstraction* (SCXML) [3], into iUML-B. For this initial work we do not support features of SCXML associated with more problematic areas of the semantic mismatch, such as 'run to completion' transition sequencing. Nevertheless, the translation provides an interesting first step towards interchange between the two notations.

SCXML is an XML notation for Harel (hence UML) style statecharts extended with a general purpose action language. The concrete syntax for SCXML is based on XML and includes a data modelling facility and an action language. An example of SCXML syntax is shown in figure 1.

To facilitate Event-B formal verification, extensions to the SCXML modelling notation are necessary so that additional modelling features required by Event-B can be integrated with the SCXML model. The SCXML schema allows extension elements and attributes belonging to a different namespace to be added. The SCXML tooling provides fallback mechanisms so that these extensions are supported without the need for syntactic definition. We define a new namespace, *iumlb* and add two new elements, *iumlb:invariant* and *iumlb:guard* as well as a number of new attributes. Invariants are not supported in SCXML and SCXML transitions only have a single *cond* attribute whereas we may need to introduce conjuncts of a transition condition at various refinement steps. The concept of refinement does not exist in SCXML. We introduce a new integer valued attribute, *iumlb:refinement*, which may be attached to any element of either namespace in order to specify the refinement level of that element.

The iUML-B tools are based on the *Eclipse Modelling Framework* (EMF) [2]. It is beneficial to load the SCXML model into EMF so that our existing model transformation technology can be used to implement the SCXML to iUML-B

```
<iumlb:invariant iumlb:refinement="1" predicate="TRUE = TRUE" name="inv_top_level"/>
<datamodel iumlb:refinement="2">
  <data expr="false" id="Gate_In.Block" iumlb:type="BOOL"/>
</datamodel>
<!-- Other model details -->
<state id="BLOCKED">
  <transition cond="[On_In.CardAccept==true]" target="UNBLOCKED">
    <iumlb:guard name="gd1" predicate="On_In.CardAccept==true" refinement="2"/>
    <assign expr="true" location="Gate_In.Block" iumlb:refinement="3"/>
  </transition>
  <onentry>
    <assign expr="true" location="Gate_In.Block"/>
    <assign expr="false" location="On_In.Reset"/>
  </onentry>
  <onexit>
    <assign expr="false" location="Gate_In.Block"/>
  </onexit>
  <iumlb:invariant predicate="Gate_In.Block == TRUE" name="GateCondition"/>
</state>
```

Fig. 1. Part of SCXML model including iumlb extension elements

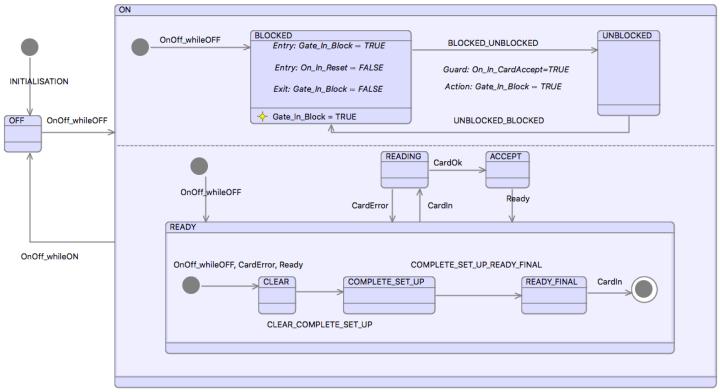


Fig. 2. State-machine diagram in iUML-B at refinement level 3 (partially annotated with guards and actions)

translation. An EMF meta-model for SCXML is available from the Sirius [1] project. It supports generic model loading capabilities for new namespace extensions. Hierarchical nested state charts are translated to similar corresponding state-machine structures in iUML-B in a series of refinement levels as directed in the SCXML *iumlb* extensions.

References

1. Eclipse Foundation. Sirius project website. https://eclipse.org/sirius/overview.html, 2016.
2. D. Steinberg, F. Budinsky, and E. Merks. *EMF: Eclipse Modeling Framework*. Eclipse (Addison-Wesley). Addison-Wesley, 2009.
3. W3C. SCXML specification website. http://www.w3.org/TR/scxml/, 2015.