# Chapter 12
# Tooling

**Michael Butler, Laurent Voisin, and Thomas Muller**

**Abstract** Together with many Rodin plug-ins, the Rodin platform supports the application of refinement-based development using Event-B and linked methods. This chapter outlines the management of the development and evolution of these tools during the lifetime of the DEPLOY project in response to deployment needs and methodological developments. The planning and maintenance process is described and a range of specific tool features developed to meet specific needs are outlined.

## 12.1 Introduction

The Rodin platform provides a range of features to support refinement-based development using Event-B. A high degree of extensibility was designed into the Rodin platform from the beginning in order to support the evolution of the Event-B method itself and of its links with other methods. There are two main ways in which extensibility is achieved in the platform. Firstly, Rodin is based on the highly extensible Eclipse platform. Secondly, the Rodin platform itself defines extensibility points to be used by Rodin plug-ins. Our experience from the DEPLOY project is that this extensibility has provided a major gain in distributed collaborative tool development. The modular architecture of Rodin allows for strong separation of concerns and functionalities, allowing many features to be developed independently of each other and making it easy to distribute responsibility for different features to different development sites.

M. Butler (✉)
Electronics and Computer Science, University of Southampton, Highfield, Southampton, UK
e-mail: mjb@ecs.soton.ac.uk

L. Voisin · T. Muller
Systerel, 1090 rue Descartes, Bt. A, 13857 Aix-en-Provence, France

L. Voisin
e-mail: laurent.voisin@systerel.fr

T. Muller
e-mail: thomas.muller@systerel.fr

A. Romanovsky, M. Thomas (eds.), *Industrial Deployment of System Engineering Methods*, DOI 10.1007/978-3-642-33170-1_12,
© Springer-Verlag Berlin Heidelberg 2013

With deployment of the platform in an industrial context being the main aim of the tool development work in DEPLOY, it was essential for the tool developers to provide mechanisms for collecting user feedback, particularly from our industrial deployment partners. In DEPLOY, we achieved this by using three means of communication: direct feedback reports to the relevant developers (contact addresses were provided for each major feature of the platform), mailing lists dedicated to the discussion of platform issues and web-based trackers for feature requests and bug reports. This chapter outlines the planning and maintenance process for Rodin tools in the DEPLOY project. We also outline a range of specific tool features developed to meet specific needs. These are grouped into tools for editing models, tools related to mathematical theory and proof, tools supporting automated analysis, tools supporting composition and reuse of models and tools for linking Event-B with other notations.

## 12.2 Planning and Maintenance

### *12.2.1 Strategic Planning*

While the original proposal of the DEPLOY project included an outline of the main expected tool developments, once the project got under way, we needed to develop more detailed plans, including time frames and allocation of responsibilities for various features. The planning of the tool development was mainly led by a dedicated workpackage of DEPLOY, with input from the workpackages on industrial deployment and on methods. Plans were updated during workpackage meetings that were held typically every six months. During the second year of the DEPLOY project a refocusing exercise was held involving all partners, which led to some new feature requests on tools. These requests were prioritised and incorporated into the planning, including the addition of two major new tasks on code generation and model-based testing. Technical specifications of major features were developed and made available on the public Event-B wiki so that the initial feedback could be incorporated into the development.

Biweekly teleconferences were held between the tool development sites, where progress was reported, and issues coming from tool developers and from the other workpackages were discussed and resolved. Minutes of these teleconferences were taken and distributed to project members to ensure visibility of the planning and progress.

### *12.2.2 Platform Stability and Performance*

Maintaining the stability and speed of the platform while continually increasing its capabilities has the potential to become unmanageable. One lesson learned in

the project is that focusing on feature implementation to satisfy user needs cannot be done without tightly linking it with performance. Several major stability and performance issues faced during the DEPLOY project were located in the core code of the Rodin platform and its user interface, causing crashes, loss of data, corruption in models, freezing and so on.

It was essential to tackle such issues in order to retain usability of the tools. An effective method of doing this was to conduct a two-pass investigation composed of code review and profiling, the first pass being a preliminary investigation and the second a deeper one. Each phase was then followed by some refactoring of the code. Moreover, solving the performance issues sometimes eliminated induced bugs as well, which meant that the scalability improvement tasks also contributed to the maintenance goals. The profiling strategy allowed the developers to achieve a better localisation of the performance loss in both the user interface and the core code, while the code reviews helped them understand the intrinsic misuses or drawbacks of particular components and architectures.

This process was used iteratively to ensure that performance was maintained while increasing the capabilities of the platform.

### *12.2.3 Processing Tool Requests*

Within the DEPLOY project, there were three ways for users to send requests to the developers:

- direct request to developers,
- a dedicated mailing list,
- a ticketing system tracking bugs and feature requests.

The requests were then prioritised and responsibility assigned with the aim of improving the toolset while following the initial workplan and addressing the feature requests.

During the final year of the DEPLOY project, this process was reviewed to give preference to some high-priority tasks and issues that were regarded as being essential to resolve by the end of DEPLOY. Preference was given to corrective maintenance and further improvement of usability. It was agreed to focus on addressing specific bugs and issues reported by the DEPLOY partners. Thus, no new feature or plug-in development was initiated after that time. The tasks to be performed by the tool developers were then listed, scheduled and prioritised. This list was regularly updated during the biweekly management meetings to rapidly capture and integrate some minor changes which were required by the DEPLOY partners to enhance the usability of the platform.

## 12.3 Model Editing

### 12.3.1 Basic Modelling Support

During the whole project lifetime, the developers of the Rodin platform have continually tried to meet user needs when designing its interface and its integrated features, to provide the user with a seamless experience. Some basic integrated features such as model navigation, editing and refactoring needed to be dealt with. These characteristics had to be taken into account when defining the initial architecture of the tool even if the tool features themselves were susceptible to change.

Indeed, after some experience of use, the notion of "ease of use" often turned out to be different from what was imagined during functionality design. The toolset refactoring and enhancements are guided more by the application of the method that it supports than by purely technical considerations.

Generally, when building a toolset dedicated to modelling, three essential feature aspects are important for ease of use:

- model editing: providing a seamless way to edit models, including refactoring support, undo/redo and other editing actions, and even auto-completion;
- model structuring: giving visual feedback on the model structure with dedicated outline views or diagram views;
- model metrics: dedicated tools to provide metrics and to ensure traceability between manipulated elements, among other things.

The experience from extensive tool use by the DEPLOY partners shows that users are influenced by their habits when choosing new technologies and tools. Editing, a basic feature in such a platform, is particularly sensitive to this kind of habits. It is obviously a good choice to take them into account, adapting the editors accordingly in order to increase the acceptance of the tools. The multiple editors that have been developed during the project originated from this consideration. Moreover, basic features such as copy/paste, undo/redo and auto-completion affect the success of particular editors.

Structuring in the Rodin platform is provided by dedicated views. For example, the Event-B Explorer gives an integrated view of the various elements and proofs of a given project. To give another example, the structure of a whole project can be viewed using the Project Diagram view. All these views are provided by either the core platform or by external plug-ins. Structuring helps the user understand and maintain models.

Metrics in the Rodin tool are provided by a set of features such as the statistic view of a project. Metrics give the modeller a measure of the complexity of their models, of how much verification has been performed and how much remains to be done, and of the degree of automation in verification.

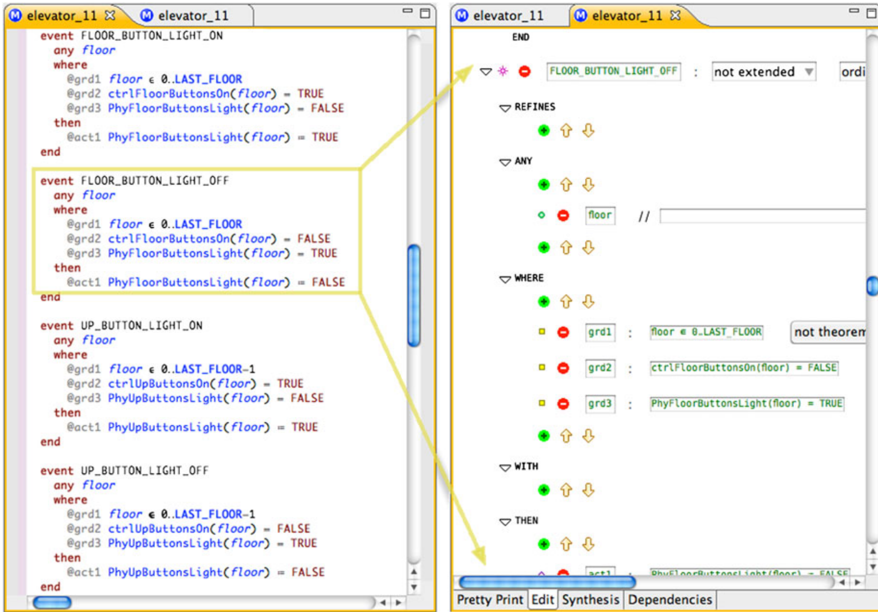


**Fig. 12.1** Text editor and structure editor compared for an ETH Zürich elevator model

## *12.3.2 Camille*

Initially, the Rodin platform for Event-B did not support a full textual representation of formal models. Model elements such as events, theorems and axioms are stored as elements in the Rodin database, and there is no classical text file for editing the models. Models are directly manipulated by a *structural form-based editor* that directly edits the underlying database and reflects its internal (tree) structure. This means the structure of a model is managed by adding and removing individual form elements, i.e., input fields, by using buttons within the editor (see the right-hand side of Fig. 12.1). The form shows the model structure and group elements of the same types, e.g., invariants, divided into sections. For example, when one wants to add a new invariant to a model, one has to press the *add* button within the invariant subsection, which will add a new set of form elements to the corresponding subsection (here comprising a label, an invariant and a comment field). The use of a database allows high extensibility and prevents a large number of syntactical and semantical errors. However, while this structural editor provides very useful guidance for novices, it became increasingly apparent that it was not able to cope with the needs of “power users”, especially the needs of the DEPLOY industrial partners, among which we will highlight the need for screen compactness, performance and flexibility.

The structural editor is very wasteful in *screen surface usage*. For example, according to our measurements, there is a difference of up to a factor of 30 in the

screen surface usage between the structural editor and a textual representation using the same font size. Figure 12.1 gives an illustration of this for a real-life model: while one event does not fully fit on the screen for the structural editor, we can see four complete events using the default layout of the textual editor. The structural editor is very slow at certain tasks (e.g., expanding all elements) and also has a limit on the number of elements that can be dealt with, mostly due to the use of resource-greedy graphical elements. A big drawback of the structural editor is the imposed *rigidity*. Indeed, many operations that come “for free” in a text editor are not available (and would require considerable effort to implement). Examples include arbitrary copy and paste, search and replace, movement of blocks of code, word count, detecting differences, balancing parentheses, and commenting on code blocks, to name just a few.

It quickly became apparent that a textual representation would have many uses, ranging from allowing simpler version control (e.g., using subversion or CVS) and sending models by email, to copying and pasting into other applications (e.g., presentation software). Some industrial users also require textual representation for internal documentation and auditing purposes. Textual representation is also less “fragile” with respect to changes to the database format. There was a strong demand from users for a text editor, which ultimately led to the implementation of Camille,[1] a semantic-aware text-editor for Rodin.

Camille features were mainly driven by power users who are experienced with IDE-based text editors. These include:

- Support of standard text editor features (cut/copy/paste, undo/redo and so on)
- Colour highlighting and error markers
- Auto-completion and templates

These features could be provided by leveraging the existing Eclipse infrastructure. The design was driven by the existing architecture and had to comply with its specific aspects. Firstly, an important design consideration was to be able to reuse the Rodin formula syntax and parser. The grammar for the Camille textual representation had to be carefully designed to be able to parse the structure of an Event-B model *independently* of the content of the predicates, expressions and actions. Another concern was that Camille should be able to co-exist with other editors. This means that all the user-relevant information has to be stored inside the Rodin database. A particularly tricky hurdle was the storing of layout information and comments. Indeed, comments have to be attached to specific elements in the Rodin database. Finally, another difficult issue was to keep as much proof information as possible when a model is changed, because one of the fundamental ideas of Rodin is incremental modelling. This means that the text editor has to update the Rodin model database in such a way that Rodin recognises that proofs can be reused. As such, the text editor needs to translate changes in the textual representation into “minimal” changes to the Rodin database.

[1]Named after Camille Claudel (1864–1943), a French sculptor and graphic artist. She also was Rodin’s source of inspiration, his model, confidante and lover.

Overall, the most challenging technical part of Camille was synchronisation with the Rodin database, particularly in the presence of other tools concurrently manipulating the same model. To achieve this, we took advantage of a new abstraction of the Rodin database as an Eclipse Modelling Framework (EMF) [5] data model. It is possible to switch back and forth between Camille and the original editor. If no text representation of the Event-B model exists upon opening Camille, then Camille will generate the initial version of the text. Users expect the text layout to be preserved in every text editor. This required some effort to implement since we do not save a text file but only a model in the Rodin database. To preserve the original layout, we decided to persist the complete input text and its rendering timestamp along with the model. We use EMF *annotations* to achieve this. Annotations allow attaching arbitrary objects to every model element and are automatically persisted. Thus, whenever Camille loads a model that has already been edited by the text editor, it shows the preserved text representation. It uses the timestamp to detect conflicts with changes that were made by other editors or tools. In those cases, the layout is discarded, and the text representation regenerated from the model.

Storing the text representation also allows saving the editor's content, even if it is not parsable, i.e., when no model can be derived due to faulty syntax or other problems. Use cases in which the user wants to save an intermediate, broken state of the model are likely in textual editing. Last, we also annotate model elements with text range information that stores their location in the formatted text. This is useful for providing visual feedback directly in Camille. For instance, the Rodin core identifies non-existent variables as a problem in the model. In order to forward that information to Camille (by underlining that variable in red), we must know the position of the offending element in the text.

Camille has proven to be very useful and has received very positive feedback from industrial and academic users. In particular, Bosch found the structural editor inadequate for realising their pilot project. In their public Deploy deliverable D19 [26] they state that "we found the text editor very helpful and prefer to use this representation". There is, however, plenty of room for improvement. In particular, Camille currently has no mechanism for supporting plug-ins that extend the syntax of Event-B. We are currently evaluating alternative architectures that would provide this.

### *12.3.3 Rodin Editor*

The original editor of the Rodin platform is a form-based one that supports the structure of Event-B models while allowing to edit its semantic elements. This editor suffers from a number of limitations. Its representation of models does not use the screen space efficiently, and navigation of large models can be uncomfortable. While a text editor (such as Camille) addresses these limitations very well, there are other desirable features that are not easily supported by a text editor. Firstly, it is important to be able to retain the ability of a plug-in to extend the syntactic structure

of Event-B models. In addition, in deploying the tool, other editor features were identified as being useful. One was the ability to display elements inherited from an abstract model by a refined model in an embedded way; this is useful when an event is extending an abstract event. Another feature identified is the ability to lock elements of a model to prevent them from being edited; this feature is useful for model elements that are generated from another source (e.g., a UML-B diagram that gives rise to generated invariants, guards and actions). The design of the original editor made it difficult to support these features, creating a need for a new editor based on an intermediary representation of the models. The new Rodin Editor was designed to go some way towards giving the look and feel of a text editor, while retaining the extensibility of the original editor as well as providing the ability to display elements inherited from an abstract model and to lock model-generated elements.

### *12.3.4 Teamwork*

For industrial-scale projects, models quickly reach a size where a team of modellers is required to work in parallel in order to meet timescales. Each modeller needs to be able to work on an aspect of the model in isolation from other modellers' changes until an appropriate point for merging their work into the main development is reached. Team-working methods like these are supported for program development by version control repository systems such as SVN. In the DEPLOY project, Bosch in particular were keen to explore team-based development of Event-B models.

The Rodin database does not support versioning and branching of development streams. While the files of a Rodin project could be stored into a version control system such as SVN, they are in an XML format which is not readily readable except by loading via the Rodin toolset. Once a Rodin resource has been moved out of the context of a Rodin project or the Rodin toolset, it is not longer readable (except as XML text strings). Therefore although a copy of a model can be stored in SVN, it is then not easy to identify its differences from a copy in a workspace. For effective teamwork it is essential that differences between copies in different development branches can be easily reviewed and merged.

The 'team working' plug-in provides tool support to enable Rodin resources to be saved in a version control repository (e.g., SVN) in a format which enables the versions of models to be compared in a differencing editor. To do this, when a project is designated for sharing, the team working plug-in produces a synchronised copy of each Rodin resource in an EMF-based format that is independent of the copy in the Rodin database. Thereafter, whenever the resource is changed in the Rodin database, the copy is automatically updated to match it and vice versa. This copy of the Rodin resource can be shared using a version control system, so that modifications are committed or updated from the repository in the usual way in which such repositories are used for code development. Differences between the repository version and the local workspace one can be reviewed in a structural editor/viewer,

allowing individual changes in the former to be merged into the latter. A limitation of the current version of the differencing editor is that it does not properly support a three-way comparison. This comparison is needed in order to review changes when these have been made in both the local workspace version and the repository one since the former was last synchronised with the latter.

## 12.4 Theory and Proofs

### *12.4.1 Prover Enhancements*

**Rodin's Built-in Provers** Enhancing the built-in provers was an important task carried out in DEPLOY. Naturally, all of the industrial partners in DEPLOY are keen for the proof to be as automatic as possible. This was done in various ways: addition of tactics and rewriters, addition of plug-ins to link tactics with external provers and so on. However, the components manipulating the proofs need to be "trusted". The architecture of Rodin isolated such components in a set called the "trusted base". Firstly, the reasoning rules to be implemented were reviewed and proven externally using other proving tools, internal or external to Rodin. Secondly, as their implementation was responsible for the correctness of the entire Event-B development, it needed to be made concise and clear enough to be trusted by using code review (maximum 30 lines of code).

**Relevance Filtering** Proof obligations of Event-B models typically contain numerous irrelevant hypotheses, i.e., hypotheses that do not contribute to a proof. Rodin's automated theorem provers (PP, newPP and sometimes also ML) perform poorly in the presence of irrelevant hypotheses: even a modest number of them can increase the time required to find a proof from seconds to days (or even longer). In some models, typically used for education, it makes sense to pass all available hypotheses to Rodin's automated provers; but in models of industrial scale Rodin's automated provers become useless unless (most) irrelevant hypotheses are removed from the input. A problem arises beyond Event-B and Rodin, indicated by the fact that the CASC competition[2] for automated theorem provers in first-order logic has a dedicated division for problems with irrelevant hypotheses.

At the beginning of the DEPLOY project, Rodin provided simple heuristics to determine which hypotheses are relevant. But these heuristics proved to be ineffective on models from industrial partners. Users therefore had to manually indicate on a per proof obligation basis which hypotheses were relevant. This is a tedious and error-prone process that is unacceptable in industrial models which have at least hundreds of hypotheses and proof obligations.

As a solution, ETH Zurich has developed a relevance filter plug-in. The plug-in implements sophisticated heuristics for detecting irrelevant hypotheses [15, 29,

[2] http://www.cs.miami.edu/~tptp/CASC

36, 40]. It provides a proof tactic, the *meta-prover*, which in a first step removes irrelevant hypotheses from the proof obligation at hand and then inputs the reduced sequent to one or several of Rodin's automated provers (PP, newPP, ML). The power of the meta-prover stems from the fact that it attempts to prove a single proof obligation by using several filter heuristics and several provers. It was shown that the meta-prover significantly increases the rate of automatically discharged proof obligations on models from various domains [35]; this includes models from industrial partners that have not been used for fine-tuning the underlying heuristics.

Evaluation by ETH Zurich shows that the meta-prover is useful for a variety of domains. Nevertheless, we do not claim that the meta-prover (in its default configuration) is useful for every model. Some proof obligations cannot be automatically proved even with a perfect relevance filter; this concerns about 7 % of the proof obligations in the benchmarks analysed [35], but the numbers vary from model to model. Moreover, we would not be too surprised if the parameters of the meta-prover needed to be readjusted for some future application. Readjusting parameters is essentially searching within a set of possible configurations and therefore requires some computing power but no ingenious insights.

In summary, the relevance filter plug-in often increases prover performance without demanding user interaction.

### *12.4.2 Mathematical Extensions*

Although Rodin supports a rich built-in mathematical language of set theory, there is always a need for richer mathematical tools such as additional theories and proof rules arising from specific applications. For example, Siemens required the ability to specify and reason about some graph-theoretic operators in a generic way. For the code generation work (Sect. 12.7.3), it is desirable to express theories of implementation-level data type operations, such as bounded integer and array operations.

The theory plug-in provides a mechanism to enrich the mathematical language of Event-B, and enables users to contribute sound proof rules. It aims to address the following two issues:

1. The old proof infrastructure of Rodin had many hardwired proof rules. In order to add a useful rule, it was necessary to write Java code that contributes to a certain extension point in Rodin. Needless to say, this was cumbersome for users. A more complicated issue is the verification of the soundness of contributed rules.
2. The mathematical language of Event-B is implemented as a fixed grammar with an abstract syntax tree. In order to add certain useful operators (e.g., sequence operators), Rodin users tended to specify them axiomatically, using contexts. Two issues arise with this particular approach. Structures defined axiomatically in contexts are not reusable outside the scope in which they are defined. Secondly, defined structures are essentially monomorphic and can only be used with the carrier sets with which they were defined.

In order to address the aforementioned limitations of the Rodin toolset, the theory plug-in provides a construct called a 'theory', which is distinct from contexts and machines. The theory component enables Rodin users to

1. specify new datatypes, including enumerated datatypes (e.g., DIRECTION := NORTH, SOUTH, EAST, WEST) and inductive datatypes (e.g., lists and trees);
2. specify new operators that are polymorphic for the types for which they are defined;
3. specify polymorphic theorems that can serve two purposes: (1) verify the properties of defined operators and datatypes, and (2) be available for instantiation and incorporation in proofs;
4. specify rewrite rules, which are directed equations that can be used to simplify formulae in proofs;
5. specify inference rules which can be used to split, discharge or simplify proof obligation sequents.

Proof obligations are generated from definitions in theory components (e.g., to ensure soundness of theorems, and rewrite and inference rules). Datatypes, operators and proof rules can be used in models after the parent theory is deployed.

Certain decisions were taken as a response to some theoretical considerations. Starting from version 1.3 of the plug-in, mutually recursive datatype definitions were not supported, since this would require more theoretical groundwork. However, we anticipate that in the future such additions may be feasible. The usability issue involved making a particularly sensitive decision with regards to theory deployment. Two competing approaches emerged as a solution to this particular issue. Automatic discovery of theory extensions was deemed too complicated and confusing to the end user, and was also shown to be technically unusable because of the way that the static checkers in Rodin work. In the end, it was decided that the user should explicitly deploy a theory to make it available for modelling and proofs.

### *12.4.3 Isabelle in Rodin*

At the beginning of the DEPLOY project, the user had three choices when valid proof obligations were not discharged automatically: (1) do manual proofs, (2) reorganise the model to make it "simpler" for Rodin's theorem provers, or (3) implement an extension of Rodin's theorem provers in Java. None of these options is fully satisfactory for models of industrial scale:

1. The number of undischarged proof obligations is typically very high, which makes manual proving an expensive task; in particular, users have reported that they had to enter very similar proofs again and again (for different proof obligations), which they found frustrating.
2. Reorganising the model helps in some situations, but not in others, and requires considerable experience with and deep understanding of Rodin's theorem provers.

3. Extending Rodin's theorem provers is quite effective in principle. However, in Java prover extensions need to be defined in a procedural style; the source code of prover extensions is therefore hard to read and write. Moreover, it is quite difficult to ensure that prover extensions are sound. If a model has been verified with an extended version of Rodin's theorem provers, it is questionable whether the model is indeed correct.

One solution to this problem is the integration of Isabelle/HOL [31] into Rodin. Isabelle/HOL is the instantiation of the generic theorem prover Isabelle [32] to higher-order logic [1, 2, 11]. The main advantage of Isabelle is that it provides a high degree of extensibility whilst ensuring soundness. In a nutshell, in Isabelle every proof has to be approved by its core, which has matured over several decades and is therefore most likely to be correct, and user-supplied prover extensions are sound by construction.

Isabelle comes with a vast collection of automated proof tactics that can be customised by the user in a declarative manner, i.e., by declaring new inference and rewrite rules. It provides linkups to several competition-winning automated theorem provers such as Z3 [3] and Vampire [33]. Last but not least, the default configuration of Isabelle's proof tactics has matured over years or decades and has been applied in numerous verification projects.

The Isabelle plug-in translates a proof obligation to Isabelle/HOL, and then invokes a custom Isabelle tactic and reports the result back to Rodin. The translation to HOL, unlike those to other theorem provers (such as PP, newPP, and ML), preserves provability: translations of provable proof obligations are provable and those of unprovable proof obligations are unprovable.[3] If the performance of the default configuration is unsatisfactory, users can inspect the translated proof obligations in Isabelle/HOL, test the behaviour of various proof tactics and declare new inference and rewrite rules to meet the desired performance goals.

The Isabelle plug-in was evaluated on the BepiColombo model [42] by Space Systems Finland. Isabelle discharged some of the proof obligations out of the box that could not be discharged automatically by Rodin. The automation rate was further increased by declaring new inference and rewrite rules within Isabelle. Adding rules to Isabelle had an important advantage over directly entering manual proofs: by declaring new rules, we did not only make the automated tactics of Isabelle prove the proof obligation under investigation, but also other proof obligations that we had not yet looked at. So the human effort on one proof obligation paid off on others. The details of this case study can be found in the PhD thesis of Schmalz [37].

The Isabelle plug-in currently has the following limitations:

- Proof search by Isabelle is slower than proof search by Rodin. There is still room for improvement. Even so, Isabelle often outperforms human proof engineers by several orders of magnitude.

[3] This claim rests on the assumption that the implementation of the translation is correct. We regard this a reasonable assumption, as the implementation of the translation is quite straightforward and concise.

- The full power of the Isabelle plug-in is obtained by tuning its proof tactics to the problem at hand. For this, one (but certainly not every) engineer needs to be familiar with Isabelle/HOL.
- The Isabelle plug-in does not interact with mathematical extensions (Sect. 12.4.2) in a useful way. A (limited) workaround is to unfold all definitions introduced by mathematical extensions before invoking Isabelle.
- If Isabelle is unable to prove a proof obligation, it outputs the subgoals that could not be proved. This output often makes it clear—to engineers familiar with Isabelle/HOL—why Isabelle was unable to complete the proof. Unfortunately, it is not clear how to translate this output into a format that can be understood by other Event-B users.

A way to customise Rodin's theorem provers for the user level, namely the theory plug-in, described in Sect. 12.4.2, was developed during the DEPLOY project. The theory plug-in avoids the overhead of integrating an external tool into Rodin and therefore has the potential of being faster and easier to use. Whether the Isabelle or the theory plug-in (or a combination of the two) is more suitable for a given task should be determined based on preliminary experiments. That said, Isabelle/HOL is an improvement over the current theory plug-in in the following ways:

- The theory plug-in is unable to express certain rules involving numerals (1, 2, 3, ...), enumerated sets ($\{a, b, c\}$), binders ($\forall$, $\{x \mid x > 0\}$), and Boolean connectives ($\wedge$, $\vee$, $\Rightarrow$). Isabelle/HOL does not have such limitations.
- Isabelle/HOL provides a default configuration of automated tactics that has proved itself over years.
- Isabelle's generic rewriter and classical reasoner are more powerful than the corresponding tactics of the theory plug-in.
- The theory plug-in has exhibited several unsoundness bugs. Unsoundness bugs in the Isabelle plug-in are unlikely (and have not been observed) because of the maturity of Isabelle and the intuitive nature of the translation.

What the Isabelle plug-in adds to Rodin is a theorem prover with a very good automation, and high degree of extensibility and soundness. To fully benefit from its power, familiarity with Isabelle/HOL is required.

## 12.5 Automated Analysis

### *12.5.1 Model Checking*

While a prover can ensure that a formal model is consistent, it cannot prove that a model behaves in the way the modeller wants it to. ProB supports a user in understanding and analysing a model. ProB is an animator and model checker that was originally developed for the B method and is now extended to also support Event-B [25].

An animator finds concrete values for a specified model. It gives the user the ability to "play" with a model by showing effects that events have. It displays a current state and lists the events that are enabled in that state. The user can then select events to "execute" and see how the state is changed. Thus the animator gives the user early feedback and helps him/her better understand the model, avoiding the high costs of changing it later in the development process. An example of a model property that is usually hard to express in a formal way is whether an event is enabled in certain situations. ProB has a low entry barrier; a few mouse clicks are usually enough to apply it to a model. Another benefit of animation is that it can help narrow the gap between domain and formal method experts by making the model more concrete and understandable. This can be further improved by creating visualisations with BMotionStudio (Sect. 12.5.2), which relies on ProB and has been significantly improved during the DEPLOY project.

In the model-checking mode, ProB can be used to automatically search for flaws such as invariant violations or deadlock situations. For model checking, proofs done in Rodin are used to reduce the effort of validating invariants in explored states. Adding support for animating refinements was necessary because refinement plays a very important role in Event-B. Now a user can inspect the relation between refinements with ProB.

ProB was extensively used in all of the deployment cases studies in the DEPLOY project, and its performance was continually improved in the course of the project. The main motivation for the enhancement of its core was an industrial case study from Siemens, in which ProB had to automatically validate large data sets that describe a railway topology. In the end, checking whether the values fulfilled the assumptions took a few minutes, whereas previously one person needed a whole month to do this manually. Another case study from Bosch, in which ProB could search for deadlock situations, led to further improvements in the models. These developments are also beneficial for other users. Not only did ProB become faster, it also enabled features such as constraint-based deadlock checking and invariant, or assertion checking, which search for counterexamples for certain properties. This helps the user spot flaws in the model sooner than in a full-blown proof.

### *12.5.2 Model Animation*

The communication between a developer and a domain expert (or manager) is very important for successful deployment of formal methods. On the one hand, to continue with development, it is crucial for the developer to get feedback from the domain expert. On the other hand, the domain expert needs to check whether his/her expectations are met. An animation tool such as ProB allows the presence of desired functionality to be validated, but requires knowledge about the mathematical notation. To avoid this problem, it is useful to create domain-specific visualisations. However, creating the code that defines the mapping between a state and its graphical representation is a rather time-consuming task. It can take several weeks to develop a custom visualisation.

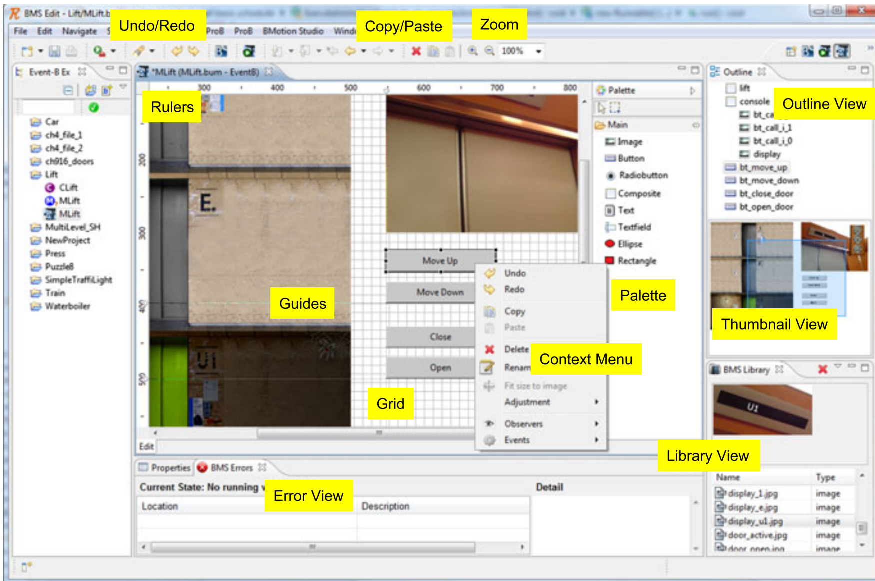


**Fig. 12.2** BMotion studio editor

B Motion Studio is a visual editor which enables the developer of a formal model to easily set up a domain-specific visualisation to demonstrate to and discuss with domain experts. B Motion Studio comes with a graphical editor (see Fig. 12.2) that allows a visualisation to be created within the modelling environment. Also, it does not require the use of a different notation for gluing the state and its visualisation.

The main advantages of B Motion Studio are:

- The modeller uses the same notation throughout. B Motion Studio uses Event-B predicates and expressions as gluing code.
- An easy-to-use graphical editor allows visualisations to be created with a few mouse clicks (see Fig. 12.2).
- B Motion Studio comes with a number of default observers and controls that are sufficient for most visualisations.
- It can be extended for specific domains.

B Motion Studio uses two concepts that are based on the model-view-control design pattern: *Controls* and *Observers*. A control is a graphical representation of some aspects of the model. Typically we use labels, images or buttons to represent information. For instance, if we model a system whose characteristics include temperature and a threshold temperature that triggers cooling down, we could simply use two labels to display both values, or we could incorporate both pieces of information into a gauge display. It is also possible to define new controls for domain-specific visualisations. Observers define the control behaviour based on information about the model state, for instance, what image it displays or what position it has.

B Motion Studio is based on the ProB animator and is integrated into the Rodin platform. The user can perform a state change by using ProB, i.e., by double-clicking on an enabled event in the Event View or by using B Motion Studio, i.e., by clicking on a button which is wired to an event. As the state changes, the observers start to evaluate expressions or predicates using ProB. The results are used to change the visualisation.

B Motion Studio is being actively developed at the University of Düsseldorf. In particular, support for other editors based on GMF/GEF [12, 13] in conjunction with B Motion Studio is being actively researched and will be implemented relatively soon.

In summary, we hope that B Motion Studio provides a way to quickly generate domain-specific visualisations for a formal model, enabling domain experts and managers to understand and validate the model. We also believe that our tool will be of use in teaching formal methods, both in lectures and as a way to motivate students to write their own formal models.

### *12.5.3 Model-Based Testing*

Model-based testing (MBT) for Event-B models was investigated in response to direct requirements of the deployment partners, in particular SAP. Such requirements are based on the fact that testing is currently the main software validation method used in industry. It is therefore a very important part of the software development cycle that consumes large proportions of project budgets, and any reliable methods that bring improvements to it are welcomed by practitioners. The possibility of MBT for Event-B relies on the ability to automatically generate tests from formal models, with the effect of reducing manual testing and increasing testing coverage. Thus, the Event-B framework is used not only to prove the consistency and correctness of the specifications but also to generate tests that can be used against existing implementations or during the implementation process using a test-driven approach.

Our MBT plug-in [4] takes an Event-B model together with its different levels of refinement, if available, as an input, and outputs a set of test cases (a test suite). The test cases consist of sequences of events together with appropriate test data that enable them. Commonly, MBT algorithms generate tests by traversing the state space of the given model, guided by coverage criteria. To address the challenge of what is usually a large state space, our solutions construct finite approximations based on machine learning algorithms. These finite state approximations provide the basis for test generation. Moreover, test suite reduction techniques are applied to reduce the size of the generated test suite. Last but not least, the approach is iterative, following the refinement methodology of Event-B.

One of the main difficulties in test generation for Event-B models is the large number of possibilities and data to be explored in order to find an appropriate test suite (cf. the state space explosion problem). Since application of explicit model checking as supported by ProB does not usually scale to large test data domains,

we explored the possibilities of (a) abstracting away data and using the built-in constraint solver of ProB (this is investigated by the University of Düsseldorf) and (b) using model learning (this is investigated by the University of Pitesti). Model learning can address the problem at hand incrementally and can take advantage of a human tester that provides input with the goal of influencing or improving the test generation. Human intervention comes at a price, requiring domain knowledge and some test generation expertise to obtain a better test suite in less time by guiding the search algorithm.

## 12.6 Composition and Reuse

### *12.6.1 Decomposition/Composition Plug-in*

Decomposing an Event-B model is a syntactic process that distributes model elements (variables, invariants, events) among several submodels. Depending on the nature of the decomposition, these submodels may interact via either shared variables or shared events. This process is intended to be used so that several refinements of an abstract model may be performed, and that when the model is sufficiently detailed, it is decomposed into submodels. The submodels can then be further refined and decomposed. As submodels are refined, external events are used to represent the assumptions about the behaviour of the other submodels (the environment of a submodel) [38]. In addition to allowing the models to reflect architectural structure, decomposition makes it possible to keep models reasonably small: refinement tends to lead to an increase in model size, so decomposition can help prevent such excessive growth.

To support the decomposition process, a plug-in was developed to automate the construction of submodels from the source model [38]. The modeller provides names for the targeted submodels, chooses whether to use shared variable or shared event interaction between them and then decides how the variables or events should be distributed amongst them. The plug-in then uses this information to generate the submodels from the source model. Initially decomposition was defined by the modeller in a wizard, and decomposition decisions were not saved. But after doing this for a while, we realised that a “decomposition configuration file” was required to save and rerun decomposition whenever necessary. A composition file is also created automatically after a decomposition has been applied to allow the user to see how the events were split. The decomposition plug-in was applied to the BepiColombo system developed at Space Systems Finland [9]; this experiment demonstrated that it is feasible to decompose the model following the architectural principles of the BepiColombo system and to then refine submodels independently.

The composition plug-in was initially developed to study how to combine machines in a structural way while preserving their properties. That study led to the development of the decomposition plug-in, since both tackle the same situation from different perspectives: composition combines machines to generate a bigger

machine while decomposition splits a bigger machine into smaller ones. A current limitation is having proof obligations generated directly into the composition file. At the moment, a new machine is generated as a result of composition, which contains proof obligations that need to be discharged to ensure valid composition. In the future, proof obligations will be defined directly in the composition file.

Currently, the best approach to using decomposition is to tailor the model to a decomposition style that allows an easy split. When decomposition has been completed, submodels are usually easier to manage and proof obligations easier to discharge. The challenge is to apply decomposition: it is not always easy from a user's point of view to do so after several refinements. Often the model and its variables are so entangled that it is hard to separate them. We are going to address this issue by improving the automation of some steps in the tool as well as the user interface. The propagation of changes from the abstract, or non-decomposed, machine to its subcomponents is also desirable.

### *12.6.2 Design Pattern Management/Generic Instantiation*

Formal methods are applicable to various domains for constructing models of complex systems. However, often they lack a systematic methodological approach, in particular in reusing existing models, to helping the development process. The objective of introducing design patterns within formal methods in general, and in Event-B in particular, is to overcome this limitation.

The idea behind design patterns in software engineering is to have a general and reusable solution to commonly occurring problems. In general, a design pattern is not necessarily a finished product, but rather a template of how to solve a problem, which can be used in many different situations. The idea is to have some predefined solutions, and to incorporate them into the development with some modification and/or instantiation. With the design pattern tool (which can also be referred to as the generic instantiation tool as it performs generic instantiation of design patterns) we provide a semi-automatic way of reusing developed models in Event-B. Moreover, the typical elements that we are able to reuse are not only the models themselves, but also (more importantly) their correctness in terms of proofs associated with the models.

In our notion of design patterns, a pattern is just a development in Event-B, including an abstract model called specification and a refinement [14]. In the course of development it might be possible to apply a pattern to the current development that was already developed before. The steps involved in using the tool are as follows:

**Matching.** The developer starts the design pattern plug-in and manually matches the specification of the pattern with the current development. In this linking of pattern and problem the developer has to define which variable in the current development corresponds to which variable in the pattern specification. Furthermore, the developer has to ensure the consistency of the variable matching by matching the events of the pattern specification with those in the development.

**Renaming.** Once the matching is done, the developer has the possibility of adapting the pattern refinement, such as renaming the variables and events or merging events of the pattern refinement with uninvolved events of the development. Renaming can become mandatory if there are name clashes, meaning that the pattern refinement includes variables or events having the same names as those of the existing elements in the development.

After the adaptation of the pattern refinement, the tool automatically generates a new Event-B machine as a refinement of the machine where the developer started the design pattern tool. The correctness of the generated machine as a refinement is guaranteed by the generation process of the tool.

Design patterns in Event-B are in fact ordinary Event-B models and are not restricted in size. As the pattern has to be manually matched to the development by the developer, its size affects usability. Furthermore, if the pattern is too specific, either it cannot be used in most situations or the developer is forced to tune his/her development in such a way that further application of the pattern is possible. A pattern-driven development could thus result in models including unnecessary elements, which would be avoided if no patterns were used.

### *12.6.3 Transformation Pattern Plug-in*

In addition to the model instantiation approach outlined in the previous section, another approach to treating modelling patterns represents refinement patterns as transformations of a model. The transformation pattern plug-in supports writing (and executing) model transformation code in a simplified programming language without compilation. The plug-in addresses the need for scalable tool-supported reuse of already established modelling practices within domains or families of projects. Technically, the transformation patterns are scripts that are written in the EOL language [6], and are intended to introduce the desired behaviour into the models upon instantiation. However, the tool can also be used for general automation of modelling routines, e.g., model generation by developers and other tools. The tool support for transformation patterns includes a text editor, an interface for choosing a pattern for instantiation, and an interface for user input during execution, for example, for specifying Event-B elements required by the pattern. To develop patterns, the API provides facilities for accessing Event-B elements. During instantiation, a pattern script is executed against the models and the Event-B elements specified by user. More technical details can be found in [41].

Modellers should consider using transformation patterns when

- they can separate and explicitly formulate a specific aspect of their modelling;
- writing and executing a script is cost-effective in terms of effort and time compared to repetitive modelling.

For reusing proofs together with plain instantiation, see the design pattern and generic instantiation plug-in. The transformation pattern plug-in is not intended for

proof reuse and, based on the imperative programming paradigm, it is rich in terms of the available instantiation operations.

The transformation pattern tool was successfully used for the implementation of our work on FMEA patterns [28], and proved to be a pragmatic way of reusing the established modelling practices.

## 12.7 Linking Event-B with Other Notations

### *12.7.1 Requirements Management/ProR*

The traceability between natural language requirements and formal models was one issue that the DEPLOY partners were struggling with. ProR [24] (Fig. 12.3) is an extensible requirements engineering platform which is part of the Eclipse Requirements Modeling Framework (RMF) [20, 34]. An integration plug-in for Rodin exists for creating traceability between Event-B models and natural language requirements.

A formal model acts as a specification for a system to be built. Since a system works correctly if it satisfies its requirements, the specification itself must meet them. By managing the requirements together with the formal model and by providing traceability between the model and the requirements, the task of verifying that the requirements have been satisfied correctly is facilitated. We developed an approach that iteratively formalises informal requirements and that creates traceability in the process (Sect. 12.7.1).

ProR was specifically developed with requirements traceability for Rodin in mind. We took a more general approach and built a generic tool that could be integrated into Rodin. The benefit of this approach is that the tool has applications beyond Rodin, thereby attracting interest and contributors from beyond the Rodin community.

We based the tool data model on the Requirements Interchange Format (ReqIF), a standard driven by the automotive industry. This gives us interoperability with a number of existing tools used in industry. Cooperation with the ITEA-project "Verde" [43], which supplied an implementation of the ReqIF data model, has been established. Interested parties include such companies as Airbus, Thales, MKS and many others. This has also triggered interest from other Eclipse-based projects, including Topcased for UML/SysML integration [21].

Tracing between informal requirements and formal models is challenging. A method for such tracing should permit us to deal with changes to both the requirements and the model efficiently. A particular challenge is posed by the interplay of formal and informal elements. We developed an incremental approach to requirements validation and systems modelling. Formal modelling facilitates a high degree of automation: it serves validation and traceability.

The foundation for our approach is requirements that are structured according to the reference model [22, 23]. We provide a system for traceability with a state-based formal method that supports refinement. We do not require all specification

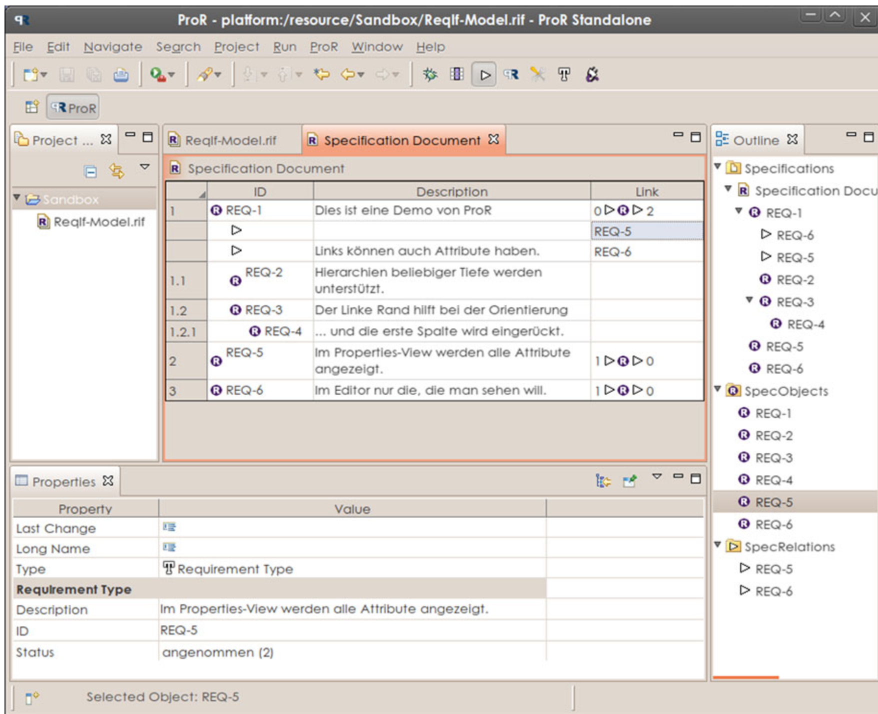


**Fig. 12.3** The ProR graphical user interface, shown with some sample data

elements to be modelled formally, and we 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.

The integration plug-in for Rodin and ProR supports this approach by allowing the manual creation of traces between Event-B model elements and individual requirements using drag and drop. Users get additional guidance from the colour highlighting of model elements (see Fig. 12.4). This has been helpful in managing traceability. We plan on extending the existing functionality by providing advanced reporting (e.g., by finding untraced requirements or model elements) and data management (e.g., by automatic marking of traces where source or target have changed since the last validation).

### *12.7.2 UML Integration*

The Event-B notation was developed with the goal of making the process of automated proof practical and efficient. For this goal it was important to keep the notation simple and closely based on the underlying concepts of set theory and predicate

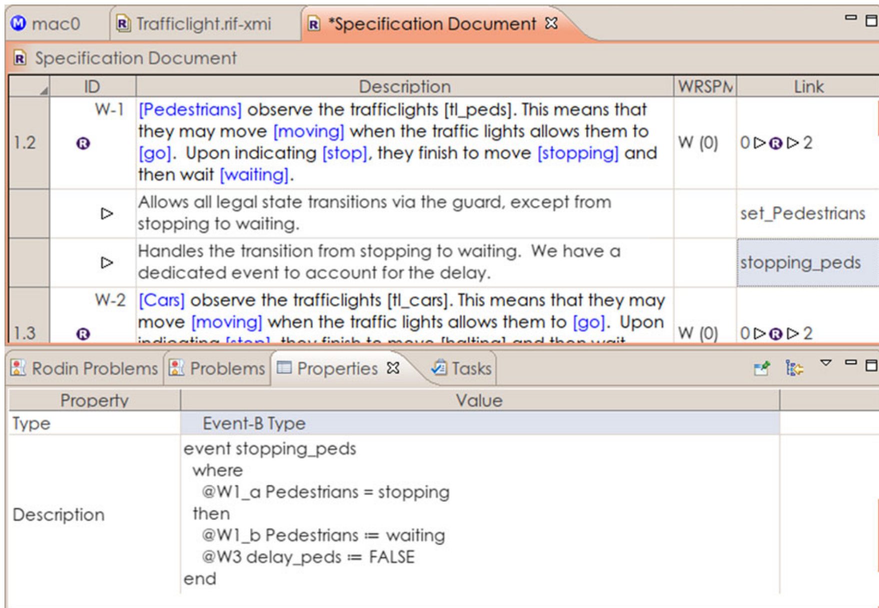


**Fig. 12.4** Integration of ProR with an Event-B formal model. Model elements are highlighted in the requirements text, and annotated traces to the model show the model element in the property view

calculus, without introducing features that complicate the process of proof. Furthermore, the modelling notation is based on a paradigm of spontaneous guarded events without any means of expressing the ordering of events within a behaviour. While this philosophy achieves a simple and flexible notation and allows strong tool support, it can also give a 'low-level' feel to the language, leading to models that are difficult to create and understand. Many users feel there is a need for a higher-level notation, for example, for expressing the structure of data, related families of events and the sequencing of events, as 'syntactic sugar' to make the modelling language more useable.

UML comes with a variety of notations for expressing models in this way. The UML-B plug-in provides a UML-style diagrammatic editor visualisation of a model to help inexperienced modellers quickly develop model abstractions and experiment with them [39]. Class Diagrams support visual structuring of data and events with a lifting mechanism to 'promote' behaviours to a set of instances. Hierarchical state-machines express ordering constraints on events and allow localised state invariants to be expressed. Modelling is performed entirely via diagrammatic editors, with Event-B expressions being annotated in detail where necessary. While there is a need for the modeller to understand Event-B (in particular, proof verification is carried out entirely within generated Event-B), UML-B gives the modeller another view on the model, which can be very helpful for his/her understanding of it. Fur-

thermore, since UML is now widely adopted by engineers, the UML-B view of the model can be more easily communicated to other modellers.

The UML-B approach has strongly influenced SAP's work on integrating Rodin with SAP's in-house graphical modelling notation. The plug-in was also applied to train control modelling by Siemens and to a simplified version of the stop/start controller from Bosch.

However, more experienced modellers require greater flexibility and would like to be able to directly edit Event-B models that are enhanced with diagrams. The new state machine plug-in allows hierarchical state machines to be added to an Event-B machine and superimpose event sequencing on existing events. Modelling is mostly carried out in the conventional Event-B notation; only the sequencing of events (and localised state invariants) is expressed as a state machine. The transitions of a state machine contribute guards and actions (representing state changes) to existing events in a flexible many-transitions-to-many-events relationship. This mechanism allows state machines to synchronise at particular points (events). The plan is to add further diagrammatic notations in this contributory fashion. For example, support for class diagrams is under way.

### *12.7.3 Code Generation*

Event-B is typically used to model software-based systems, and these require code to be produced that can be compiled. Automatic code generation tools enable implementations to be generated from Event-B. Users can therefore benefit from the use of formal development, and from the efficiency gains provided by automatic code generation. In many cases, there is a semantic gap between the modelling concepts used in Event-B, and the implementation components or source code constructs. To address this, we extend Event-B with a tasking language [8]. This approach was formulated to be of use for engineers developing multi-tasking, real-time embedded systems, so we chose the constructs of the Ada programming language as a basis for the tasking structure. We make use of high-level concepts, such as tasks and shared machines, and we model the environment in such a way that simulators can be generated as part of the development. The tasking Event-B feature bridges the gap between Event-B specification and implementation by providing implementation specification and code generation tools. Translators are currently available for Ada and C, and more can be added since extensibility has been considered during the development.

Tasking Event-B is an extension to Event-B, but includes restrictions to ensure the code is implementable. To use the tasking Event-B approach successfully, development must be structured in a particular way, which is achieved using decomposition. We decompose development into a number of Event-B machines; this makes reasoning about large developments easier since each decomposed machine may be refined further, and this, too, can be decomposed again. We use the Rodin decomposition tool (Sect. 12.6.1) to perform model decomposition, and the code generation

tool uses structural information from decomposition to structure implementation. In previous work [7] we found that models quickly turned intractable as they became very large and generated a large number of proof obligations. In that work, we described an object-oriented intermediate language that was used to guide code generation. However, we encountered difficulties due to the large semantic gap between Event-B and the intermediate specification language. In the current tool, the methodology maintains a small semantic gap; we achieve this by adding only a minimal number of constructs to the Event-B language.

Following decomposition and refinements at the implementation level, we identify implementation features such as AutoTask, Environ or Shared machines. AutoTask machines model controller tasks in the implementation. Shared Machines model protected objects (or a similar mutual exclusion mechanism), and Environ machines model the environment. We then specify some tasking features, such as the task type (e.g., periodic, triggered, one-shot, repeating), priority and the task body. We use a task body specification to write the flow control for an individual task. The flow control may be an event, sequence, loop or branch. An output construct is also provided, to allow text to be output to a console during simulation. During the early stages of the project, we realised that a self-imposed restriction on inter-task communication had forced us to adopt a particular modelling style, and this did not reflect the way that systems are implemented in industry. We removed the restriction on inter-task communication for task-environment communication since environ machines can be used to generate simulators. The main driver for the restriction on the deployable part of the system is that we wish to generate safe multi-tasking code. We aim to make the deployable code Ravenscar-compliant, and the restrictions remain in place.

The current tool also includes a solution for modelling low-level interaction with the environment, that is, sensing and actuating. The sensing and actuating events give rise to two styles of implementation: one using entry calls, and one using addressed variables. In the entry call style, calls are a way of updating sensed variables in a controller task, and controlled variables in the environment task. In the latter style, addressed variables associate address information with each of the parameters of sensing or actuating events in the controller task.

The development of the code generation approach was strongly guided by input from the industrial partners, especially Space Systems Finland, Siemens and Bosch. There was a common requirement from these partners that the tool should support generation of multi-tasking implementations with periodic and aperiodic tasks as well as variable sharing between tasks. Before the plug-in was developed, we devised a case study of a heater controller, including an Event-B development and a multi-tasking implementation in Ada. Feedback from the industrial partners indicated that the Ada implementation was representative of the typical implementation they would follow, so this served as a good benchmark for defining the structure of generated implementations. Once the code generation plug-in was developed, we were able to generate Ada and C versions of the heater controller. We also generated Ada and C implementations of a simplified version of the start/stop controller from Bosch.

### 12.7.4 Flow Plug-in

There are situations where the event-based modelling style of Event-B appears awkward due to the large number of event-ordering constraints necessary to express event-ordering properties. Such constraints, although trivial in their nature, can pollute a model with auxiliary variables, invariants and actions, which affects model legibility and proof automation (due to an increased number of hypotheses). The flow plug-in [10, 16, 18] helps us construct Event-B models with rich control flow properties in a concise manner. It greatly simplifies certain kinds of proofs and offers proof techniques (most importantly, assertion propagation along an event chain) that are difficult to exercise consistently in plain Event-B. The flow plug-in was used in the Bosch cruise control and stop/start system case studies.

The plug-in supports two development methods. One is to apply the graphical notation to express event orderings [18]. The effect is a set of proof obligations demonstrating that the given orderings are found among machine traces. One can effectively express point conditions (safety invariants that hold between the execution of certain two events) and prove liveness properties. The other approach is to translate use case scenarios, informally defined in a requirements specification, into the formal notation of the flow plug-in. Such scenarios are detailed in a refinement-like manner along with the refinement of the host Event-B machine [17].

The plug-in provides a modelling environment for working with graph-like diagrams describing event-ordering properties. In the simplest case, a node in such a graph is an event of the associated Event-B machine; an edge is a statement about the relative properties of the connected nodes/events.

A use case diagram is only defined in association with one Event-B model; it does not exist on its own. The use case plug-in automatically generates all the relevant proof obligations. A change in a diagram or its Event-B model leads to the recomputation of all affected proof obligations. These proof obligations are dealt with, as are all other proof obligation types, using a combination of automated provers and interactive proofs. As in the proofs of model consistency and refinement, the feedback from an undischarged use case proof obligation may often be interpreted as a suggestion for a diagram change, such as the introduction of additional assumptions or assertions—predicate annotations on graph edges that propagate properties along the graph structure.

### 12.7.5 Modes Plug-in

In the early stages of DEPLOY it was recognised that many challenging developments need to deal with dynamic system reconfiguration. Such models typically describe several "stable" phases of system behaviour and some activities that lead from one phase to another. This has led to the design and implementation of the modes plug-in [27]. The plug-in extends the Event-B modelling notation with a superstructure describing system modes and transitions between modes. It employs

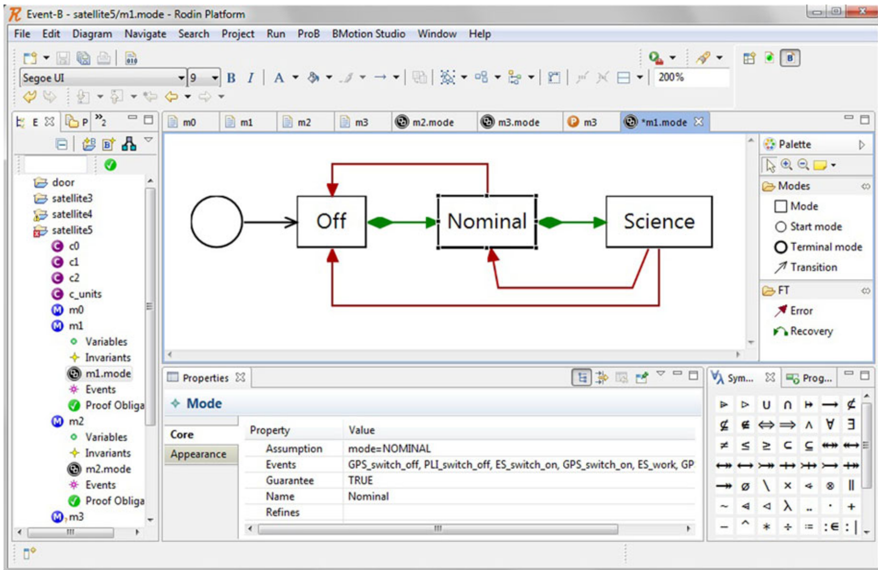


**Fig. 12.5** The tool screenshot

a simple visual notation based on modecharts [19]. The graphical notation, called the modal view of a system, coexists with an Event-B machine; the two define differing viewpoints on the same design. A modal view has formal semantics (proof semantics and operational semantics) from which a set of consistency conditions are derived. These demonstrate that a modal view and the corresponding Event-B machine are in agreement.

A mode in a modal view is an island of relatively stable system behaviour. A system still evolves within a mode, but within far stricter limits than those of the safety invariant of an Event-B machine. Such limits are defined by a pair of assumption and guarantee predicates. The assumption predicate is normally interpreted as a set of conditions under which a system is able to stay in the mode; the guarantee describes what the system is doing while in the mode. The guarantee is a before-after predicate: it references both current and next states. Modes are related via mode transitions; these are also characterised formally and, in this respect, are similar to Event-B events (Fig. 12.5). The state model is borrowed from an Event-B machine.

The central feature of the tool is support for stepwise development of the modal view along with the process of Event-B refinement. When a machine is refined, a developer also needs to refine the modal view to reflect the changes in the machine (or state view). There are a number of refinement laws describing possible ways of refining a modal view; these give rise to transformational patterns offered to a developer. Hence, a refined modal view is produced by transforming an abstract modal view (Fig. 12.6).

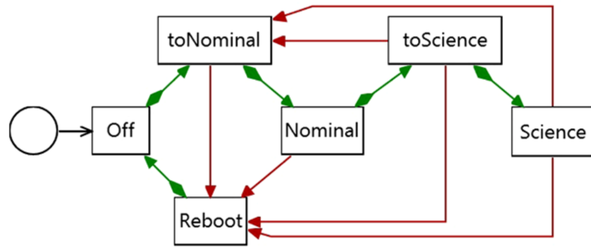


**Fig. 12.6** The modal view refined

The tool automatically generates verification conditions that are necessary for ensuring that a given modal view is sound and consistent with an Event-B machine. A number of case studies have been developed using the tool to ascertain the scalability of the method and the implementation [27, 30].

The modes plug-in was used in the BepiColombo case study by Space Systems Finland. One conclusion to draw from the experience with this tool is that it is generally gratifying to stratify a design into aspects, or, as we call them, views. This permits a focused analysis and discussion of properties pertinent to a given view and an explicit connection to any requirements about modal properties.

## 12.8 Conclusions

At the start of the DEPLOY project we already had a version of the Rodin platform that had been developed as part of the RODIN FP6 project (2004 to 2007). Exposing the tool to serious industrial users in the DEPLOY project drove the developers to implement significant improvements in performance, usability and stability of Rodin. The experiences and expectations of the DEPLOY industrial partners also led to key innovations in tool functionality through a range of plug-ins, as outlined in this chapter.

The ease with which, in order to provide seamless functionality, the core Rodin platform may be extended with plug-ins by a range of teams indicates that the use of Eclipse, together with the Rodin extension points, provides a very effective architecture for tool extensibility. In addition to those developed within DEPLOY, several plug-ins have been developed by sites outside of DEPLOY. An up-to-date list of Rodin plug-ins is maintained on the Rodin wiki (wiki.event-b.org). It is clear that the Rodin platform has supported the evolution of a rich ecosystem of integrated tools whose development has been tempered by experiences of industrial usage. This ecosystem of tools continues to evolve.

**Acknowledgements** Contributions to this chapter were made by Andy Edmunds, Thai Son Hoang, Alexei Iliasov, Florian Ipate, Michael Jastram, Lukas Ladenberger, Michael Leuschel, Ilya Lopatkin, Chris Lowell, Issam Maamria, Carine Pascal, Daniel Plagge, Jann Röder, Vitaly Savicks, Matthias Schmalz, Renato Silva, Colin Snook and Alin Stefanescu.

## References


1. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory. Springer, Berlin (2002)
2. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940)
3. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin (2008)
4. Dinca, I., Ipate, F., Mierla, L., Stefanescu, A.: Learn and test for Event-B—A Rodin plug-in. In: Proc. ABZ'12 Conference, Lecture Notes in Computer Science. Springer, Berlin (2012). http://deploy-eprints.ecs.soton.ac.uk/379/
5. Eclipse modeling framework. http://www.eclipse.org/emf
6. Eclipse Object Language (2011). http://www.eclipse.org/gmt/epsilon/doc/eol/
7. Edmunds, A., Butler, M.: Linking Event-B and concurrent object-oriented programs. In: Proc. Refine 2008—International Refinement Workshop (2008). http://eprints.ecs.soton.ac.uk/16003/
8. Edmunds, A., Rezazadeh, A., Butler, M.: Formal modelling for Ada implementations: Tasking Event-B. In: Proc. Ada Europe 2012, Lecture Notes in Computer Science. Springer, Berlin (2012)
9. Fathabadi, A.S., Rezazadeh, A., Butler, M.: Applying atomicity and model decomposition to a space craft system in Event-B. In: Proc. Third NASA Formal Methods Symposium (2011). http://eprints.ecs.soton.ac.uk/22048/
10. Plug-in, F.: Event-B wiki page. http://wiki.event-b.org/index.php/Flows
11. Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge University Press, Cambridge (1993)
12. Graphical editing framework. http://www.eclipse.org/gef
13. Graphical modeling framework. http://www.eclipse.org/gmf
14. Hoang, T.S., Fürst, A., Abrial, J.-R.: Event-B patterns and their tool support. In: Proc. Seventh IEEE International Conference on Software Engineering and Formal Methods, pp. 210–219 (2009). http://deploy-eprints.ecs.soton.ac.uk/204/
15. Hoder, K.: SUMO inference engine. http://www.cs.manchester.ac.uk/~hoderk/sine
16. Iliasov, A.: Augmenting Event-B specifications with control flow information. In: Proc. NODES'10 (2010)
17. Iliasov, A.: Augmenting formal development with use case reasoning. In: Proc. Ada Europe 2012 (2012)
18. Iliasov, A.: Use case scenarios as verification conditions: Event-B/flow approach. In: Proc. 3rd International Workshop on Software Engineering for Resilient Systems, SERENE'11 (2011)
19. Jahanian, F., Mok, A.K.: Modechart: A specification language for real-time systems. IEEE Trans. Softw. Eng. **20**, 933–947 (1994). http://dx.doi.org/10.1109/32.368134
20. Jastram, M., Graf, A.: Requirements Modeling Framework. Eclipse Mag. **6.11**, 87–92 (2011)
21. Jastram, M., Graf, A.: Requirement traceability in topcased with the requirements interchange format (RIF/ReqIF). In: First Topcased Days Toulouse (2011)
22. Jastram, M., Hallerstede, S., Ladenberger, L.: Mixing formal and informal model elements for tracing requirements. In: Proc. Automated Verification of Critical Systems (AVoCS) (2011)
23. Jastram, M., Hallerstede, S., Leuschel, M., Russo, A.G.: An approach of requirements tracing in formal refinement. In: Proc. VSTTE. Springer, Berlin (2010)
24. Jastram, M.: ProR, an open source platform for requirements engineering based on RIF. In: SEISCONF (2010)
25. Leuschel, M., Butler, M.J.: ProB: An automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008). Tool webpage: http://www.stups.uni-duesseldorf.de/ProB
26. Loesch, F., Gmehlich, R., Grau, K., Mazzara, M., Jones, C.: DEPLOY deliverable D1.1: Report on pilot deployment in automotive sector (D19) (2010)

27. Lopatkin, I., Iliasov, A., Romanovsky, A.: On fault tolerance reuse during refinement. In: Proc. 2nd International Workshop on Software Engineering for Resilient Systems, SERENE'10, London, UK (2010). Available as CS-TR-1188 at Newcastle University, UK
28. Lopatkin, I., Prokhorova, Y., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Patterns for representing FMEA in formal specification of control systems. Technical report 1003, TUCS Turku, Finland (2003)
29. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Appl. Log. **7**(1), 41–57 (2009)
30. Mode/FT Views wiki page. http://wiki.event-b.org/index.php/Mode/FT_Views
31. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, Berlin (2002)
32. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reason. **5**(3), 363–397 (1989)
33. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Commun. **15**(2–3), 91–110 (2002)
34. RMF: Requirements modeling framework. http://eclipse.org/rmf
35. Röder, J.: Relevance filters for Event-B. Master Thesis, ETH Zurich (2010)
36. Roederer, A., Puzis, Y., Sutcliffe, G.: Divvy: An ATP meta-system based on axiom relevance ordering. In: Proc. CADE, Lecture Notes in Computer Science, vol. 5663, pp. 157–162. Springer, Berlin (2009)
37. Schmalz, M.: Formalizing the logic of Event-B: Partial functions, definitional extensions, and automated theorem proving. PhD Thesis, ETH Zurich (2012)
38. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for Event-B. Softw. Pract. Exp. **41**(2), 199–208 (2011). http://deploy-eprints.ecs.soton.ac.uk/293/
39. Snook, C., Savicks, V., Butler, M.: Verification of UML models by translation to UML-B. Lect. Notes Comput. Sci. **6957**, 251 (2011). http://eprints.ecs.soton.ac.uk/22921/
40. Sutcliffe, G., Puzis, Y.: SRASS—A semantic relevance axiom selection system. In: Proc. CADE. Lecture Notes in Computer Science, vol. 4603, pp. 295–310. Springer, Berlin (2007)
41. Transformation patterns plug-in wiki page. http://wiki.event-b.org/index.php/Transformation_patterns
42. Varpaaniemi, K.: BepiColombo models v6.4. http://deploy-eprints.ecs.soton.ac.uk/244
43. Verde, I.P.: Validation-driven design for component-based architectures. http://www.itea-verde.org/