Ref. Ares(2012)620980 - 24/05/2012

Project DEPLOY
Grant Agreement 214158
*"Industrial deployment of advanced system engineering methods for high productivity and dependability"*

***DEPLOY Deliverable D45***

# D9.4 Model Construction Tools and Analysis Tools IV


***Editor:*** *Thomas Muller (Systerel)*
***Internal reviewers:*** *Michael Jastram(Düsseldorf University),*
*Thai Son Hoang(ETH Zürich)*
***Contributors:*** *Laurent Voisin (Systerel), Renato Silva (Southampton University) Nicolas Beauger (Systerel), Issam Maamria (Southampton University), Colin Snook (Southampton University), Matthias Schmaltz (ETH Zürich), Vitaly Savicks (Southampton University), Andy Edmunds (Southampton University), Alexei Iliasov (Newcastle University), Ilya Lopatkin (Newcastle University), Thai Son Hoang (ETH Zürich), Michael Leuschel (Düsseldorf University), Daniel Plagge (Düsseldorf University), Lukas Ladenberger(Düsseldorf University),*
*Alin Stefanescu (Pitesti University)*


***Public Document***

30 April 2012

# Contents


Introduction 1

General Platform Maintenance 3

- Core Platform Maintenance 3
- Plug-in incompatibilities 6
- Mathematical extensions/ Theory Plug-in 7
- Decomposition 8
- Team-Based Development 9
- Modularisation 10
- UML-B 11
- ProR 12
- BMotion Studio 13
- Mode/FT Views 15
- Shared references 16

Scalability 17

- Overview 17
- Motivations 17
  - Improved Performance 17
  - Design Pattern Management / Generic Instantiation 17
  - Editors 17
- Choices / Decisions 18
  - Improved Performance 18
  - Design Pattern Management / Generic Instantiation 18
  - Editors 19
- Available Documentation 20
- Status 20
  - Improved Performance 20
  - Design Pattern Management / Generic Instantiation 21
  - Editors 21
- References 21

Prover Enhancement 22

- Overview 22

Motivations 22
New rewriting and inference rules 22
Advanced Preferences for Auto-tactics 22
Isabelle Plug-in 23
ProB Disprover 23
SMT Solver Integration 23
Incorporating SAT Solvers via Kodkod 23
Choices / Decisions 24
New rewriting and inference rules 24
Advanced Preferences for Auto-tactics 24
Isabelle Plug-in 24
ProB Disprover 25
SMT Solver Integration 25
Incorporating SAT Solvers via Kodkod 25
Available Documentation 26
Status 26
New rewriting and inference rules 27
Advanced Preferences for Auto-tactics 27
Isabelle Plug-in 27
ProB Disprover 27
SMT Solver Integration 28
Incorporating SAT Solvers via Kodkod 28
References 28
Code Generation 30
Overview 30
Motivations 30
Choices / Decisions 30
Tasking Event-B and its edition 31
Allowing Extensibility 31
Targeting new Languages 31
Using Abstract Translators 32
Available Documentation 32
Status 33
References 33

Model-based testing 34
Overview 34
Motivations 34
Choices / Decisions 34
Available Documentation 35
Status 35
Model Checking 37
Overview 37
State Space Reduction, Compression and Hashing 37
Constraint-Based Checking 37
New Features 37
Experiments 38
Motivations 38
Choices / Decisions 38
Available Documentation 38
Status 39
References 39

# 1 Introduction

The DEPLOY deliverable D45 is made of this document, the Rodin core platform, and its plug-ins (i.e. the DEPLOY tools). The Rodin platform can be retrieved from the SourceForge site[1]. The documentation of the DEPLOY tools is available from two sites:

- the Event-B wiki[2].
- the Event-B Handbook[3].

This document gives an overview of the work done within the WP9 *Tooling research and development* work package, during the fourth and last period of the DEPLOY project (Feb 2011-Apr 2012). It also gives a status of the toolset by the end of the DEPLOY project.

At the end of the third year of the project, the DEPLOY industrial partners reported some issues they encountered and expressed some specific requests which were then considered mandatory to be addressed before the end of the DEPLOY project. Hence, the various tasks to be done by the WP9 partners have been early scheduled, prioritized, and periodically updated to both fit with the original Description of Work (DoW) and give answers to DEPLOY partner issues.

Here is a short list of important results that WP9 partners committed to achieve and are worth citing in this introduction:

- Increasing usability and performance of the toolset was the main targets regarding scalability. They have been obtained through major core and UI refactorings. More scalability was gained with the addition of the Generic Instantiation Plug-in (previously mentioned as Design Pattern Management).
- A Model-Based Testing plug-in whose development was guided by the needs of WP1-4 partners has been released and documented.
- Increasing prover performance through the addition of several rewriting or inference rules, as well as two plug-ins, the *Export to Isabelle* and *SMT Solvers* which both help to raise the number of automatically discharged proof obligations.
- An enhanced proving experience, and proving ability with the introduction of user customizable and parametrizable tactic profiles.

This document is made of the following parts: general platform maintenance, scalability, prover enhancement, code generation, model-based testing, and model-checking.

Note that each of these parts describes the improvements performed, and is structured, except the general platform maintenance part, as follows:

- Overview. The involved partners are identified and an overview of the contribution is given.
- Motivations. The motivations for each improvement are expressed.
- Choices / decisions. The decisions (e.g. design decisions) are justified.
- Available documentation. Some pointers to the Event-B wiki, handbook, or related publications are listed.
- Status. The status reached on the given topic at the end of the Deploy project (as of 30 Apr 2012) is given.
- References. The references for further details.

This structure has been chosen, as in the previous tooling deliverables (D23, D32), in order to highlight the answers given to each tasks mentioned in the Description of Work (DoW). The first part being focused on tool maintenance is structured upon the same template but for each considered plug-in, in order to ease the reading.

**References**


[1] https://sourceforge.net/projects/rodin-b-sharp/files/Core_Rodin_Platform
[2] http://wiki.event-b.org
[3] http://handbook.event-b.org

# 2 General Platform Maintenance

## 2.1. Core Platform Maintenance

### 2.1.1. Overview

The Rodin platform versions concerned by this deliverable are:

- 2.1(08.02.2011),
- 2.2(01.06.2011),
- 2.2.2(01.08.2011),
- 2.3(04.10.2011),
- 2.4(31.01.2012),
- 2.5(30.04.2012).

This year, the maintenance carried on fixing identified bugs, although an emphasis was put on rectifying usability issues. Indeed, during the annual meeting in Nice, the WP9 members agreed to refocus on addressing some specific bugs and issues reported by DEPLOY partners, and wished resolved by the end of DEPLOY. Thus, no new features were implemented but those mentioned in the description of work. The tasks to be performed by the WP9 members were then scheduled, prioritized and regularly updated during the WP9 bi-weekly teleconferences. The updates allowed to capture and integrate rapidly some minor changes to enhance the usability of the platform which were required by the DEPLOY partners. The following paragraphs will give an overview of the the work that has been performed concerning maintenance on the existing platform components (i.e. core platform and plug-ins).

See the Release Notes[1] and the SourceForge[1] databases (bugs and feature requests) for details about the previous and upcoming releases of the Rodin platform.

Some other which were later added and prioritized are worth to mention:

- Possibility to highlight patterns in the Proving UI,
- A better output providing warnings and errors in case of wrong or missing building configurations,
- The switch to Eclipse 3.7,
- A Handbook to complete and enhance the existing documentation.

### 2.1.2. Motivations

The tasks to resolve the issues faced by the DEPLOY industrial partners have been listed and have been assigned to groups according to their priority. A high priority means a high need in the outcome of a given task. Group 1 has the highest priority, group 2 has an intermediate priority, and group 3 has the lowest priority. Group 4 concerns topics that could not be resourced during the lifetime of DEPLOY.

| Group 1 (highest priority) | Responsible |
|---|---|
| Performance<br>- Core (large models, etc.)<br>- GUI (incl. prover UI, edition, etc.) | SYSTEREL |
| Prover Performances<br>- New rewriting rules / inference rules<br>- Automatic tactics (preferences, timeout, etc.) | SYSTEREL |
| ProB Disprover (incl. counter examples to DLF POs) | Düsseldorf |
| Stability (crash, corruption, etc.) | SYSTEREL |
| Editors | SYSTEREL / Düsseldorf |

| Group 2 | Responsible |
|---|---|
| Prover Performances | |
| - SMT provers integration | SYSTEREL |
| - connection with Isabelle | ETH Zürich |
| - Mathematical extensions | Southampton / SYSTEREL |
| - ProB | Düsseldorf |
| Scalability | |
| - Decomposition | Southampton |
| - Modularisation plug-in | Newcastle |
| - Team-Based Development | Southampton |
| Plug-in incompatibilities | Newcastle |
| Model-based testing | Pitesti/Düsseldorf |

| Group 3 | Responsible |
|---|---|
| Scalability | |
| - Generic instantiation | ETH Zürich / Southampton |
| - UML-B maintenance | Southampton |
| Code Generation | Southampton |

| Group 4 |
|---|
| Prover Integrity |
| Integrity of Code Generation |
| |
| |

The platform maintenance, as it can be deduced from the above tables, mainly concerned stability and performance improvement. These topics will be discussed and detailed in a separate chapter about scalability improvements.

Other improvements of utmost importance were made on the platform. These improvements either came from DEPLOY partners specific needs, or were corresponding to previously identified needs (listed in D32 - Model Construction tools & Analysis III Deliverable [2]). Hence we review below the motivations of some noteworthy implemented features:

**Possibility to highlight patterns in the Prover UI**.
This feature came from a request of DEPLOY partners[3], often facing the need to find particular patterns such as expressions in long predicates (e.g. long goals). Since Rodin 2.2, and its new Proving UI interface, a feature has been added, allowing to search and highlight a string pattern into the whole Proving UI views and editors. This function has also been enabled on direct selection of text in this UI.

**A better output providing warnings and errors in case of wrong or missing building configurations**.
This issue, often being seen as a bug or as a plug-in incompatibility, was raised when a user imports and tries to use a model on a platform with some missing required plug-ins. The user often thought his models corrupted whereas Rodin was not able to build them, and hid this information to the user. This is why, since Rodin 2.3, an output has been provided in such case, taking the form of warnings or errors that any user can understand and review.

**The switch to Eclipse 3.7**.
Due to the major improvements made every year in Eclipse releases and the continuously growing number of contributing projects, some of them used as basis for Rodin plug-ins, the

Rodin platform follows the evolution and is adapted every year quickly to the latest Eclipse version available. This year, Rodin 2.3 originated the switch from Eclipse 3.6 to Eclipse 3.7.

**A Handbook to complete and enhance the existing documentation**.

At the DEPLOY Plenary Meeting in Zürich in 2010, it has been stated that the current documentation, in its state at that time, would not support an engineer starting using the tools without significant help of an expert[4]. Significant efforts to improve the documentation were performed and coordinated by Düsseldorf, and took form of a handbook[5]. The Rodin handbook has the aim to minimize the access to an expert, by providing the necessary assistance to an engineer in the need to be productive using Event-B and the Rodin toolset. The contents of the handbook, user oriented, were originated by some contents of the Event-B wiki.

### 2.1.3. Choices / Decisions

**Revisited task priority**

This year, the process of giving priority to maintenance tasks was revisited according to the refocus mentioned above. The aim was to address all the major scalability issues before the end of DEPLOY. Thus, the requests coming from DEPLOY partners were given high priorities, and they were also prioritized against the already planned tasks coming from both DEPLOY partners and the Description of Work.

**Keep 32-bit versions of the Rodin platform on Linux and Windows systems**

It was asked by end users to make both 32-bit and 64-bit versions of the Rodin platform available for Linux and Windows platforms. Only a 64-bit version of Rodin is available on Mac platforms as 32-bit Mac (early 2006) platforms are no longer maintained. The request to offer 64-bit was motivated by the possibility to increase for them the available Java heap size for some memory greedy platforms (these before Rodin 2.3). After having ensured to get all physical platforms to test the different versions, and despite the drawbacks of assembling and maintaining more platforms (5 platforms instead of 3), Rodin 2.4 has started a new era where Rodin platforms are available on Linux and Windows 64-bit as well as 32-bit.

### 2.1.4. Available Documentation

The release note pages give useful information about the Rodin platform releases[6]. Two trackers follow and document the platform status in terms of knows bugs and feature requests:

- a sourceforge bug tracker,[7]
- a sourceforge feature requests tracker[8].

The Rodin handbook is available as a PDF version, a HTML version, and help contents within Rodin.[5]

### 2.1.5. Status

By the end of DEPLOY, the ultimate version of the Rodin platform is 2.5. One can download it on the sourceforge repository [9] and read the release notes [10] on the wiki.

## 2.2. Plug-in incompatibilities

### 2.2.1. Overview

Some plug-in incompatibilities occured and were continuously handled throught the lifetime of the project. Most incompatibilities are related to the difficulty to make plug-ins work together and occured directly during the installation or at first launch.

### 2.2.2. Motivations

By its extensibility nature, the Rodin platform is susceptible to incompatibilities. Indeed, there are many ways in which incompatibilities could occur, and some occurred in the lifetime of DEPLOY. A good example, is the dependency management. Suppose that a bundle x_v1.0 is needed by a plug-in A (i.e. a dependency from A has been defined to x in at most the version 1.0) and installed in Rodin. Furthermore the plug-in x_v1.1 is needed by a plug-in B. Both versions 1.0 and 1.1 of x could not be installed and used at the same time and thus create some usage incompatibility.

### 2.2.3. Choices / Decisions

It has been decided in cooperation with all the WP9 partners to find better ways to address the plug-in incompatibility issues. First of all, the various partners refined the concept of "plug-in incompatibility". Hence, various aspects could be identified and some specific answers were given to each of them. The user could then defined more clearly the incompatibility faced. Plug-in incompatibilities can be separated in two categories:

- Rodin platform/plug-in incompatibilities, due to some incorrect matches between Rodin included packages and the plug-in dependencies (i.e. required packages). These incompatibilities, when reported, allowed the plug-in developers to contact SYSTEREL in charge of managing the packages shipped with a given version of Rodin. It could also allow traceability of incompatibilities and information to the user through a specific and actualized table on each Rodin release notes page on the Wiki[11].
- Plug-in/plug-in incompatibilities, due to some incorrect matches between needed/installed packages, or API/resources incompatible usage. A table was created on each release notes wiki page, and a procedure was defined[12] so that identified incompatibilities are listed and corrected by the concerned developers.

It appeared that cases of using a model which references some missing plug-ins were formerly often seen as compatibility issues although they were not.
After the incompatibilities have been identified, the concerned developing counterparts assigned special tasks and coordinated to solve issues as soon as possible. Incompatibilities are often due to little glitches or desynchronisation. As a result, direct coordination of counterparts appeared to be appropriate because of its promptness and effectiveness.

### 2.2.4. Available Documentation

The process to report plug-in incompatibilities is documented on each release notes page right after the plug-in availability tables. [13][14]

### 2.2.5. Status

As the time of writing this deliverable, no plug-in incompatibilities are left or have been reported between the platform and plug-ins or between plug-ins.

## 2.3. Mathematical extensions/ Theory Plug-in

### 2.3.1. Overview

Mathematical extensions have been co-developed by SYSTEREL (for the core Rodin platform) and Southampton (for the Theory plug-in). The main purpose of this new feature was to provide the Rodin user with a way to extend the standard Event-B mathematical language by supporting user-defined operators, basic predicates and algebraic types. Along with these additional notations, the user can also define new proof rules (prover extensions). The scope of the ongoing work on the Theory plug-in centers around bug fixes, improving usability and performance and exploring other venues for operator definitions. Please note that we will distinguish the "Theory" plug-in from the "theory" component (using this capitilization).

### 2.3.2. Motivations

The Theory plug-in provides a high-level interface to the Rodin core platform capabilities which enables the definition of mathematical and prover extensions grouped into modules called theories. These mathematical and prover extensions are new algebraic types, new operators/predicates and new proof rules. Theories are developed in the Rodin workspace, and proof obligations are generated to validate prover and mathematical extensions. When a theory is completed and (optionally) validated, the user can make it available for use in models (this action is called the deployment of a theory). Theories are deployed to the current workspace (i.e., workspace scope), and the user can use any defined extensions in any project within the workspace. The Rule-based Prover was originally devised to provide a usable mechanism for user-defined rewrite rules through theories. Theories were, then, deemed a natural choice for defining mathematical extensions as well as proof rules to reason about such extensions. In essence, the Theory plug-in provides a systematic platform for defining and validating extensions through a familiar technique: proof obligations.

Support for using polymorphic theorems in proofs was added in version 1.1.

### 2.3.3. Choices / Decisions

The Theory plug-in contributes a theory construct to the Rodin database. Theories were used in the Rule-based Prover (before it was discontinued) as a placeholder for rewrite rules. Given the usability advantages of the theory component, it was decided to use it to define mathematical extensions (new operators and new datatypes). Another advantage of using the theory construct is the possibility of using proof obligations to ensure that the soundness of the formalism is not compromised. Proof obligations are generated to validate any properties of new operators (e.g., associativity). With regards to prover extensions, it was decided that the Theory plug-in inherits

the capabilities to define and validate rewrite rules from the Rule-based Prover. Furthermore, support for a simple yet powerful subset of inference rules is added, and polymorphic theorems can be defined within the same setting. Proof obligations are, again, used as a filter against potentially unsound proof rules.

### 2.3.4. Available Documentation

Pre-studies (states of the art, proposals, discussions):

- *Proposals for Mathematical Extensions for Event-B* [15]
- *Mathematical Extension in Event-B through the Rodin Theory Component* [16]
- *Generic Parser's Design Alternatives* [17]
- *Theoretical Description of Structured Types* [18]

Technical details (specifications):

- *Mathematical_Extensions wiki page* [19]
- *Constrained Dynamic Lexer wiki page* [20]
- *Constrained Dynamic Parser wiki page* [21]
- *Theory plug-in wiki page* [22]
- *Records Extension Documentation on wiki* [23]

User's guides:

- *Theory Plug-in User Manual* [24]

### 2.3.5. Status

Work on the Theory plug-in includes:

- Bug fixes.
- Usability improvements.
- Exploring other potential ways of defining operators and types (e.g., axiomatic definitions).

## 2.4. Decomposition

### 2.4.1. Overview

Decomposition can advantageously be used to decrease the complexity and increase the modularity of large systems, especially after several refinements. Main benefits are the distribution of proof obligations over the sub-components which are expected to be easier to be discharged and the further refinement of independent sub-components in parallel introducing team development of a model which is attractive for the industry. Shared variable and shared event decomposition are supported in the same tool: the former seems to be suitable when designing concurrent programs while the latter seems to be particularly suitable for message-passing distributed programs. The tool was initially developed in ETH Zurich. Further development of the tool was a collaboration between ETH Zurich, Southampton and SYSTEREL. After some user feedback, the tool was improved in terms of usability and performance. The ongoing work aims for a more automated tool that can propagate changes in the sub-components and minimise the user intervention as much as possible while maintaining or enhancing the performance.

### 2.4.2. Motivations

The *top-down* style is one of the most used in modelling in Event-B. It allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement process when dealing with many events and state variables. The main purpose of the model decomposition is precisely to address such difficulty by separating a large model into smaller components. Two methods have been identified for the Event-B decomposition: shared variable (proposed by Abrial) and shared event (proposed by Butler). We developed a plug-in in the Rodin platform that supports these two decomposition methods for Event-B. Because decomposition is monotonic, the generated sub-components can be further refined independently. Therefore we can expand the team development options: several developers share parts of the same model and work independently in parallel. Moreover the decomposition also partitions the proof obligations which are expected to be easier to discharge in sub-components.

### 2.4.3. Choices / Decisions

The tasks performed on the decomposition plug-in were focused on consolidation.

### 2.4.4. Available Documentation

Two entry pages are available on the wiki:

- *The Decomposition Plug-in User Guide,* [25]
- *The event model decomposition wiki page.* [26]

### 2.4.5. Status

The decomposition plug-in is available in version 1.2.2 and works on Rodin 2.4 and 2.5. It is available from the main Rodin update site.

## 2.5. Team-Based Development

### 2.5.1. Overview

The Team-Based Development plug-in enables Rodin models to be stored in a version repository such as SVN. During the final year of DEPLOY, the development concerned mostly consolidation and interface enhancement.

### 2.5.2. Motivations

To achieve the storage of Rodin models in SVN, the Team-Based Development plug-in allows maintaining a synchronised copy of the model resources in an EMF format. EMF comparison tools can then be used to examine differences between versions.

### 2.5.3. Choices / Decisions

The EMF default XMI format was used to store models in a form that can be accessed independently of the Rodin database. Since an EMF framework for Event-B already existed (but relied on serialisation into the Rodin database), it was easy to provide an option to serialise into the XMI format. The EMF compare was customised to provide a more user friendly comparison.

### 2.5.4. Available Documentation

See the Team-based Development Documentation on wiki [27].

### 2.5.5. Status

The Team-Based Development plug-in is available on the main Rodin update site. Currently 3-way comparison is not supported. (3-way comparison is needed if 2 people check out and change the model so that there are 2 working copies as well as a repository baseline).

## 2.6. Modularisation

### 2.6.1. Overview

Modularisation offers an intuitive yet rigorous mechanism for structuring large developments. Its main distinguishing feature is the use of *interface* components - a special form of an abstract machine defining callable operations and external variables. To decompose a design using modularisation a designer needs to identify a self-contained part of a design and captures it in an independent *module*. A module is an Event-B development starting with an interface as its top-level abstraction. Modules may be included into machines and offer *operations* that may be used to define actions of an including machine. Variables of an interface may not be modified directly and thus the invariant property of an interface holds at all times. An essential part of a decomposition step is identifying a part of an abstract state that is removed and said to be realised by one of the included modules. Modularisation works well for sequential and concurrent systems.

### 2.6.2. Motivations

The major motivation for the initial design and the continuing development of the modularisation extension is the desire to deal with large-scale specification using Event-B notation and refinement-based development. Generally, users find the modularisation approach fairly intuitive and flexible.

### 2.6.3. Choices / Decisions

Modularisation is not a simple extension and it changes both the syntax and semantics (the set of proof obligations) of Event-B. It is possible to encode all of the modularisation concepts directly in the Event-B notation. However, it was deemed very important to offer high-level structuring concepts - interfaces and operations - that engineers may relate to. Although an operation call in modularisation is merely a metaphore, the syntactic resemblances of an procedure call in a programming language significantly improves model readability. A similar idea motivated the introduction of an interface component - sensible syntax and efficient set of proof obligations are

paramount.

### 2.6.4. Available Documentation

See the Modularisation plug-in [28] wiki page.

### 2.6.5. Status

The modularisation plug-in is available for Platform versions 1.6 - 2.4. Starting from Platform version 2.3, modularisation is compatible with the ProB animator and model checker. The modularisation is not compatible with Camille text editor.

## 2.7. UML-B

### 2.7.1. Overview

The UML-B plug-in supports modelling in a UML-like diagrammatic notation with conversion to Event-B for verification. UML-B supports class and state machine diagrams as well as a project structure diagram (showing machines and contexts). UML-B continues to be supported but currently is not undergoing new development. Some enhancements were made last year in order to improve the usability of state machines, however, most new development concentrates on the new Event-B diagrammatic extensions to Event-B (such as the Event-B Statemachines plug-in).

The Event-B Statemachines plug-in is a new tool, based on the UML-B state machine diagrams, which allows to integrate state machines into normal Event-B machines. It provides a graphical diagram editor, an Event-B generator and, as an optional plug-in, diagram animator for ProB.

### 2.7.2. Motivations

The Event-B Statemachines plug-in has been introduced as a result of the necessity to integrate higher level constructs into established Event-B modelling process. From the experience of working with the UML-B tool it became apparent that a tighter integration is required between Event-B models under development and high level extensions such as state machines. In particular, this integration should be flexible enough to make it easy for an user to add new constructs at any point of Event-B development and work with them through refinement, which is a key feature of the Event-B language and Rodin tool.

### 2.7.3. Choices / Decisions

For the Event-B Statemachines plug-in the following key decisions were made:

- The UML-B state machines example was taken as a concept.
- Well-established Eclipse development frameworks — EMF and GMF — were chosen for implementation of the new plug-in and simplified (from the original UML-B state machines) EMF metamodel and diagram have been implemented.
- The integration idea between Event-B and state machines was based on EMF extension mechanism and serialisation principle: a state machine was designed as an extension to EMF Event-B metamodel that would be serialised as a string to an attribute in Rodin database, thus making the details of it transparent to Rodin.

- For the translation of state machines to Event-B the QVT framework has been selected, considering it as a well-supported framework, used in other Eclipse projects such as GMF, and more declarative nature of it compared to pure Java, which would improve maintainability.

As a result of work on Event-B Statemachines plug-in a set of additional plug-ins has been developed that forms a framework to support developer effort in implementing other similar tools and high level extensions for Event-B. These plug-ins include generic serialised persistence and navigator support for new EMF extensions, generic diagram metamodel and navigator actions, generic refinement and Event-B generator modules for new extensions.

### 2.7.4. Available Documentation

See the Event-B State-machines Documentation on wiki [29].

### 2.7.5. Status

The Event-B State machines plug-in is available on the main Rodin update site.

## 2.8. ProR

### 2.8.1. Overview

ProR is a replacement of the original requirements plug-in, which got discontinued in 2010. It is based on the OMG ReqIF standard[30], which provides interoperability with industry tools. It evolved into the Eclipse Foundation project "Requirements Modeling Framework" (RMF[31]), resulting in significant visibility. ProR is independent from Rodin. Integration is achieved with a separate plug-in that provides support for traceability and model integration.

### 2.8.2. Motivations

While the original requirements plug-in for Rodin was useful as a prototype, a number of shortcomings lead to a new development. In particular, the original plug-in was a traceability tool with externally managed requirements. With ProR, requirements are authored and edited within Eclipse. The original plug-in supported only a limited number of attributes and flat (unstructured) requirements. ProR supports all data structures that the ReqIF standard[30] supports. Further, ReqIF-support for industry tools like Rational DOORS, MKS or IRqA is expected in the near future, while the original plug-in required a custom adaptor for each data format.

ProR is developed independently from Rodin. Dependencies to Rodin exist only in the Rodin integration plug-in. This significantly decreases the maintenance effort for the integration plugin, while increasing the visibility of ProR in the Open Source community. The move of ProR from the University of Düsseldorf to the Eclipse Foundation increases visibility even further. The Rodin integration plug-in is maintained as an independent project at GitHub.

### 2.8.3. Choices / Decisions

The following key decisions were made when developing ProR:

- **New development, rather than continuing the original plug-in** - the architecture of ProR differs significantly from that of the original plug-in (as explained earlier). In addition, new technologies like EMF promised a cleaner, more powerful framework for an implementation.
- **ReqIF as the underlying data model** - the ReqIF standard[30] is gaining traction and promises interoperability with industry tools. In addition, a digital version of the data model was available for free and could serve as the foundation for the model code.
- **The Eclipse Modeling Framework** (EMF) was identified as a technology that would allow a quick and clean foundation for an implementation. Further, the Rodin EMF-plug-in represents a convenient interface for integrating ProR and ProB. Last, the digital data model from the OMG could be imported directly into EMF and used for generating the model code.
- **Keeping ProR independent from Rodin** - There is significant interest in ReqIF right now, but this interest is unrelated to formal methods. By providing an implementation that is independent from Rodin, we have a much larger target group of potential users and developers. By carefully designing extension points, we can still provide a powerful Rodin integration.
- **Eclipse Foundation Project** - we were actively establishing an open source community around ProR. By recruiting engaged partners early on, development progressed faster than anticipated. By becoming an Eclipse Foundation project [31], we exceeded our goals in this respect.

### 2.8.4. Available Documentation

There are two major sources of information about ProR on the internet:

- ProR at the Eclipse Foundation [31]
- ProR Documentation for end users and plugin developers [32]

### 2.8.5. Status

ProR took on a life on its own as part of the Requirements Modeling Framework[31]. It is currently in the incubation stage of an Eclipse project. There are currently five committers in total, with two from the Rodin project, namely Michael Jastram (Project Lead) and Lukas Ladenberger.

The Rodin integration supports:

- Creating traces between model elements and requirements,
- Highlighting of model elements in the requirements text,
- Marking of invalidated traces, where either the requirement or model element had changed.

The Rodin integration is hosted at the University of Düsseldorf.

## 2.9. BMotion Studio

### 2.9.1. Overview

BMotion Studio is a visual editor which enables the developer of a formal model to set-up easily a domain specific visualisation for discussing it with the domain expert. BMotion Studio comes with a graphical editor that allows to create a visualisation within the modeling environment. Also, it does not require to use a different notation for gluing the state and its visualisation. BMotion Studio is based on the ProB animator and is integrated into Rodin. However, BMotion Studio is independent from Rodin. Integration is achieved with a separate plug-in.

- BMotion Studio has been quite successful, and besides a number of bug fixes and some performance profiling and tuning, the useability of the tool was improved.
- One of our students made experiments towards visualizing industry standards with BMotion Studio. The first experiments were quite successful.
- First experiments towards visualizing mathematical assertions found in formal specifications using Venn Diagrams/Euler Diagrams/Constraint Diagrams were made.

### 2.9.2. Motivations

The communication between a developer and a domain expert (or manager) is very important for successful deployment of formal methods. On the one hand it is crucial for the developer to get feedback from the domain expert for further development. On the other hand the domain expert needs to check whether his expectations are met. 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. To overcome this problem, BMotion Studio comes with a graphical editor that allows to create a visualisation with static images and drag and drop within the modelling environment, not requiring additional skills.

An often stated limitation in using formal methods is the difficulty in understanding the formal notation. To overcome this problem and to support the user we made first experiments towards visualizing mathematical assertions found in formal specifications using Venn Diagrams/Euler Diagrams/Constraint Diagrams.

### 2.9.3. Choices / Status

The following key decisions were made when developing BMotion Studio:

- **Keeping BMotion Studio user-friendly** - The user should be able to create a visualization not requiring additional skills in programming languages.
- **ProB as animator for providing state information** - With the ProB animator, we have a powerful tool for interacting with the model.
- **Provide extensibility for specific domains** - By carefully designing extension points, we can provide a powerful integration for specific domains.
- **Keeping BMotion Studio independent from Rodin** - By providing an implementation that is independent from Rodin, we have a much larger target group of potential users and developers.

### 2.9.4. Available Documentation

- BMotion Studio Documentation for end users and plugin developers.[33]
- Context sensitive help is under work.

### 2.9.5. Status

The tool is available as a part of the ProB animator and is ready for use for visualizing Event-B models within the Rodin tool. Of course, we are working on new features.

## 2.10. Mode/FT Views

### 2.10.1. Overview

The Mode/FT Views plug-in is a modelling environment for constructing modal and fault tolerance features in a diagrammatic form and formally linking them to Event-B models. The consistency conditions between the modal/FT views and Event-B models are ensured by additional proof obligations. The views form a refinement chain of system modal and fault tolerant behaviour which contribute to the main Event-B development. The views reserve a place for tracing modal and FT requirements.

### 2.10.2. Motivations

There are two major motivations for creating the Mode/FT Views plug-in:

- An overview of the requirements documents within DEPLOY indicated that systems are often described in terms of operational modes and configurations. This led to a work on formal definition of modal systems.
- Fault-tolerance is the crucial part of the behaviour of dependable critical systems that needs to benefit from formal modelling as functionality does. The requirements documents for the pilot studies in DEPLOY contain a high number of requirements related to fault handling and fault tolerant behaviour. A significant part of them are also described by using recoveries and degraded modes.

The plug-in provides an environment for specifying modal and fault tolerant behaviours which are often interrelated. By having a refinement chain of system-level modal diagrams, the development benefits from additional modelling constraints and improved requirements traceability.

### 2.10.3. Choices / Decisions

The following key decisions were made when developing Mode/FT Views:

- **The Eclipse Graphical Modelling Framework (GMF)** was used as a platform for building a user-friendly modelling environment.
- **Proof obligations for the views are injected into the standard PO repository of the models** - This ensures that all the tools related to theorem proving can be used in the same way as they are used for Event-B proof obligations.

### 2.10.4. Available Documentation

A Mode/FT Views documentation is available for users.[34]
Papers have been published:

- *Structuring Specifications with Modes* [35].
- *Modal Systems: Specification, Refinement and Realisation* [36].
- *On Fault Tolerance Reuse during Refinement* [37].

### 2.10.5. Status

The Mode/FT Views is a plug-in for the Rodin platform. The tool is available from its update site [34]

## 2.11. Shared references

[1] http://wiki.event-b.org/index.php/D32_General_Platform_Maintenance#Available_Documentation
[2] http://wiki.event-b.org/index.php/Category:D32_Deliverable
[3] https://sourceforge.net/tracker/?func=detail&atid=651672&aid=3092835&group_id=108850
[4] http://wiki.event-b.org/index.php/User_Documentation_Overhaul
[5] http://handbook.event-b.org/
[6] http://wiki.event-b.org/index.php/Rodin_Platform_Releases
[7] https://sourceforge.net/tracker/?group_id=108850&atid=651669
[8] https://sourceforge.net/tracker/?group_id=108850&atid=651672
[9] http://sourceforge.net/projects/rodin-b-sharp/files/Core_Rodin_Platform/2.5/
[10] http://wiki.event-b.org/index.php/Rodin_Platform_2.5_Release_Notes
[11] http://wiki.event-b.org/index.php/Rodin_Platform_Releases#Current_plug-ins
[12] http://wiki.event-b.org/index.php/Rodin_Platform_Releases#Known_plug-in_incompatibilities
[13] http://wiki.event-b.org/index.php/Rodin_Platform_2.4_External_Plug-ins
[14] http://wiki.event-b.org/index.php/Rodin_Platform_2.5_External_Plug-ins
[15] http://deploy-eprints.ecs.soton.ac.uk/216/
[16] http://deploy-eprints.ecs.soton.ac.uk/251/
[17] http://wiki.event-b.org/index.php/Constrained_Dynamic_Parser#Design_Alternatives
[18] http://wiki.event-b.org/index.php/Structured_Types
[19] http://wiki.event-b.org/index.php/Mathematical_Extensions
[20] http://wiki.event-b.org/index.php/Constrained_Dynamic_Lexer
[21] http://wiki.event-b.org/index.php/Constrained_Dynamic_Parser
[22] http://wiki.event-b.org/index.php/Theory_Plug-in
[23] http://wiki.event-b.org/index.php/Records_Extension
[24] http://wiki.event-b.org/images/Theory_UM.pdf
[25] http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide
[26] http://wiki.event-b.org/index.php/Event_Model_Decomposition
[27] http://wiki.event-b.org/index.php/Team-based_development
[28] http://wiki.event-b.org/index.php/Modularisation_Plug-in
[29] http://wiki.event-b.org/index.php/Event-B_Statemachines
[30] http://www.omg.org/spec/ReqIF/
[31] http://eclipse.org/rmf
[32] http://pror.org
[33] http://www.stups.uni-duesseldorf.de/BMotionStudio
[34] http://wiki.event-b.org/index.php/Mode/FT_Views
[35] http://deploy-eprints.ecs.soton.ac.uk/105/
[36] http://deploy-eprints.ecs.soton.ac.uk/153/
[37] http://deploy-eprints.ecs.soton.ac.uk/253/

# 3 Scalability

## 3.1. Overview

Three main domains concerned the scalability task during the last year of the project:
**Improved Performance**. According to the refocus mentioned in the General Platform Maintenance chapter, much of the reworking efforts performed on the core platform basically aimed to overcome Rodin scalability weaknesses, and the continuous need of seamless proving experience. The whole performance was enhanced by core implementation and user interface refactorings. Improvements made to the proving experience will be detailed in a separate chapter.
**Design pattern management / Generic Instantiation**. Formal methods are applicable to various domains for constructing models of complex systems. However, often they lack some systematic methodological approaches, in particular in reusing existing models, for helping the development process. The objective in introducing design patterns within formal methods in general, and in Event-B in particular, is to overcome this limitation.
**Editors**. Along DEPLOY, edition became a central concern. Indeed, many usability issues appeard the models involved in pilots became "industrial" sized:

- The legacy structured editor based on a form edition specific architecture reached its limits in editing such complex models. Industrial partners found this issue significant regarding the adoption of the Rodin platform in industry and providing a new structured editor correcting this issue became an important task during this last year of the DEPLOY project.
- Camille textual editor had to tackle challenges related to resources mapping and management for projects of industrial size mentioned above or even the design of extension capabilities. Extension capabilities became a major concern since the grammar could be extended with theories.

## 3.2. Motivations

### 3.2.1. Improved Performance

Many issues concerning stability and performance were related to the core code of the Rodin platform, causing crashes, loss of data, corruption in models. Other issues were related to the UI causing platform hanging, and sometimes leading to its freezing which required sometimes to kill the Rodin process, thus also leading to potential loss of data and corruption in models. Addressing these issues before the end of DEPLOY was mandatory.

### 3.2.2. Design Pattern Management / Generic Instantiation

The idea of 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 on how to solve a problem which can be used in many different situations. The idea is to have some predefined solutions, and incorporate them into the development with some modification and/or instantiation. With the design pattern tool we provide a semi-automatical 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.

### 3.2.3. Editors

This section concerns the actual editors that user work with to view and modify models.

**Editor of the core platform**

Two major kinds of failures were faced when developing industrial sized models:

- Failures occurring on Windows platforms, the mostly used system, when reaching the operating system limit number of graphical elements in memory allowed to be displayed at once. This situation used to crash the Rodin platform, and it appeared necessary to control the number of graphical elements needed to display the model elements. Architectures sparing these graphical elements were thus favoured.
- Failures due to the high consumption of memory by the heavy graphical elements used by the forms of legacy editor.

  Moreover, it appeared important to visualize some elements that are not part of the current level of modelling, but are linked to it, for example, the actions in an abstract event, or its guards. Such elements are called the "inherited" elements, or "implicit children". The legacy structured editor being directly interfaced with the underlying Event-B models in database, it was difficult and tricky to modify it in order to display inherited elements. This was one more argument to go for a new editor based on a intermediary representation of the models.

  Another motivation to go for another editor was to get a more modest and ergonomic way to visualize and edit the models. The models being presented through styled text pretty printing and the edition tightly derived from a textual edition.

**Camille**

Camille was developed as an alternative editor in 2009. In contrast to the existing editor, this was a pure textual editor that supported standard features like copy and paste, auto-completion, color highlighting and many more. In addition, it performed better than the existing structural editor.

While successful, Camille offered only limited support for models with extended elements contributed by plug-ins. Also, the underlying EMF Compare framework was problematic due to bugs and performance issues.

## 3.3. Choices / Decisions

### 3.3.1. Improved Performance

SYSTEREL lead a two phase investigation to have a better idea of the work to be done. Each phase being followed by some refactoring of the code. Out of the early investigation, a cause and effect relationship has been found between performance loss and the various reported bugs, such as "platform hanging" bugs. Indeed, it appeared that solving the performance issues sometimes solved induced bugs as well, making the scalability improvement tasks encompass the maintenance goals.
Later, a deeper investigation was performed where profiling and code review were the two techniques used to tackle the issues left. The profiling strategy allowed to get a better localisation of the performance loss in both UI and core code while the code review helped to understand the intrinsic misuses or drawbacks of particular components and/or architectures. The proving UI was refactored in order to use a light textual representation.

### 3.3.2. Design Pattern Management / Generic Instantiation

In our notion of design patterns, a pattern is just a development in Event-B including an abstract model called specification and a refinement. During development there might be the possibility of applying a pattern that was already developed to the current development. The steps of using the tool are as follows.

**Matching** The developer the 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 the which variable in the current development corresponds to each 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 events in the development.

**Renaming** Once the matching is done, the developer has the possibility to adapt 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 there are name clashes, meaning the pattern refinement includes variable or events having the same name as existing elements in the development.

After adaptation of the pattern refinement the tool generates automatically 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 being a refinement is guaranteed by the generation process of the tool.

Design Patterns in Event-B are nothing else than ordinary Event-B models and are not restricted with respect to their size. Due to the fact that pattern and development have to be matched manually by the developer, the size of a pattern effects the usability. Furthermore, if the pattern is too specific either the pattern cannot be used in most situations or the developer is forced to tune its development such that a further appliance of the pattern is possible. A pattern-driven development could thus lead to models including needless elements that would have been avoided when developing without patterns.

### 3.3.3. Editors

**Rodin Editor**

The Rodin Editor has been built on the basis of enhancements brought to the Proving UI. In fact, by using the same kind of textual component, every model element is presented as text, as would a pretty print do. The edition is made possible through the use of a unique additional graphical component that overlays the presentation of the element (to be edited). It lighten the presentation, but also side steps the use of a significant amount of greedy graphical elements. Only two graphical elements are needed for presentation and edition: the model viewer and its overlay editor!
The first public version (0.5.0) of the Rodin Editor has been released on the 13/07/2011 as a plug-in of the Rodin platform. This decision has been made in order to let the plug-in incubation for the time it is being tested and stated that no regression is introduced. The Rodin Editor since its version 0.6.1 is part of the Rodin core platform.

**Camille**

Camille was quite successful, but due to the limited suitability for editing models using plug-ins, an architectural overhaul has been planned. The course of action was not obvious, however, as there are a number of possible architectures with varying advantages and disadvantages.

Therefore, Ingo Weigelt at the University of Düsseldorf created a technical report[1] that analyzes the needs of the users and suggests a number of possible solutions. In the FP7 project ADVANCE, a solution will be decided upon and implemented.

## 3.4. Available Documentation

Links for Improved Performance

- There is a wiki page concerning the Rodin Performance[2].

Links for Design Pattern Management / Generic Instantiation

- A paper concerning the generic instantiation is available[3]
- An article about Pattern Management is published on the wiki[4].
- The article about Generic Instantiation on the wiki[5]

Links for Edition

Two documentation pages have been created for the Rodin Editor:

- A general page describing the editor and how it works [6].
- A user guide page [7].

Moreover, a special category has been created on both SourceForge feature request[8] and bug[9] trackers.

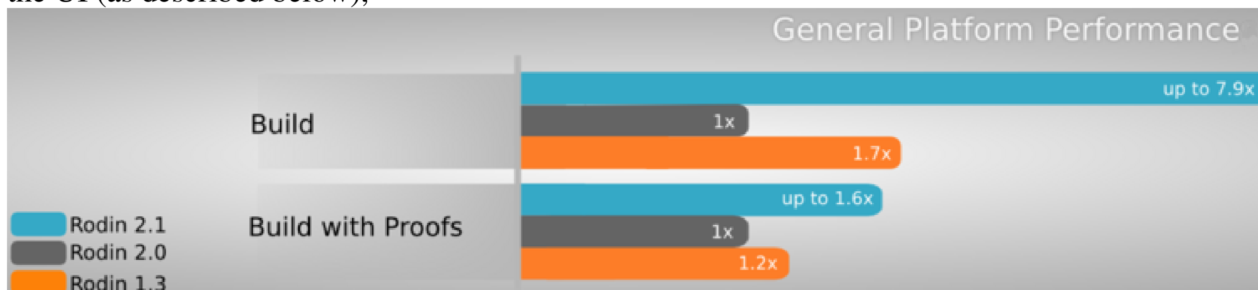
Camille Documentation

- As the Camille text editor is based on the standard Eclipse editor, that documentation applies fully as well[10].
- Additional information is available in the Text Editor[11] section on the wiki.
- The Rodin Handbook[12] provides some basic usage information in a few places.
- The technical report[1] describes analysis of user's needs and possible solutions.

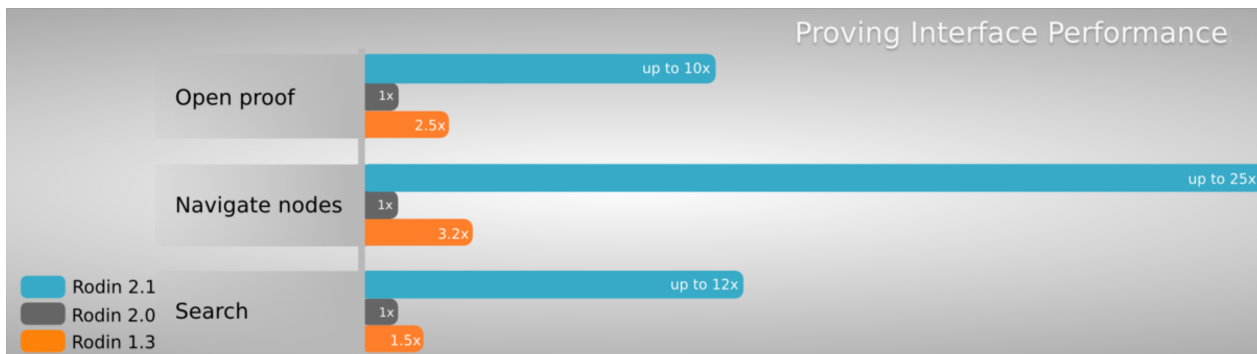
## 3.5. Status

### 3.5.1. Improved Performance

The refactoring made on both core code and UI code allowed to gain up to 25 times speed-up on the UI (as described below),




and almost a 2 times speed-up in the core code, making the platform usable in an industrial sized context.




### 3.5.2. Design Pattern Management / Generic Instantiation

The Generic Instantiation plug-in is available in version 0.2.1 for Rodin 2.4 and above from the main Rodin Update Site in the *Composition and Decomposition* category.

### 3.5.3. Editors

**Rodin Editor**

The Rodin Editor is part of the Rodin platform since Rodin 2.4 released by end of January 2012.

**Camille**

Camille has been quite successful, and besides a number of bug fixes and some performance profiling and tuning, the code base has not changed much.

## References

[1] http://cobra.cs.uni-duesseldorf.de/w/Special:Publication/Weigelt2012
[2] http://wiki.event-b.org/index.php/Rodin_Performances
[3] Silva, Renato and Butler, Michael (2009) "Supporting reuse of Event-B developments through generic instantiation". In, International Conference on Formal Engineering Methods (ICFEM 09), Rio de Janeiro, Brazil, 09 - 12 Dec 2009. 19pp. (Submitted) http://eprints.soton.ac.uk/68737/
[4] http://wiki.event-b.org/index.php/Pattern
[5] http://wiki.event-b.org/index.php/Generic_Instantiation
[6] http://wiki.event-b.org/index.php/Rodin_Editor
[7] http://wiki.event-b.org/index.php/Rodin_Editor_User_Guide
[8] https://sourceforge.net/tracker/?group_id=108850&atid=651672
[9] https://sourceforge.net/tracker/?group_id=108850&atid=651669
[10] http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.user/concepts/concepts-6.htm Eclipse
[11] http://wiki.event-b.org/index.php/Text_Editor
[12] http://handbook.event-b.org/

# 4 Prover Enhancement

## 4.1. Overview

Two tasks concerned the prover performance from the core platform: the addition of rewriting and inference rules, and the addition of a mechanism to allow the customization and the parametrization or combination of tactics. While the addition of rewriting and inference rules has always been a regular task to enhance the Rodin integrated prover during DEPLOY lifetime, a new way to manage tactics has been implemented. In fact, the user is now able to define various types of tactics called 'profiles' which could be customized and parameterized tactics to discharge some specific proof obligations.
The ProB plug-in allows to find and visualize counter examples to the invariant and deadlock preservation proof obligations. ProB has also been used for finding count examples to proof rules of the industrial partner Siemens.
The SMT Solvers plug-in allowing to use the SMT solvers within Rodin is an effective alternative to the Atelier-B provers, particularly when reasoning on linear arithmetic.
The Isabelle/HOL integration plug-in allows to extend the Rodin proving capabilities through the definition of tactic written in Isabelle/HOL and thus, provides help to automatically discharge proof obligations, but also allows the verification of the new extensions of the Rodin sequent prover written in java through the translation of their corresponding rules.

## 4.2. Motivations

### 4.2.1. New rewriting and inference rules

In an Event-B development, more than 60% of the time is spent on proofs. It has been a continuous aim to increase the number of automatically discharged proof obligations (POs) by improving the capabilities of the integrated sequent prover through the addition of rewriting and inference rules. Naturally all of the industrial partners in DEPLOY are keen for the proofs to be as automatic as possible. These rules were provided through tactics, or existing or newly created. These tactics were automatic, or manual, or sometimes both. Providing new proving rules, even if it sometimes does not increase directly the number of automatically discharged POs aims to help the user to interactively discharge them and spare his time.

### 4.2.2. Advanced Preferences for Auto-tactics

The proportion of automatically discharged proof obligations heavily depends on Auto-Tactic configuration. Sometimes, the automatic prover fails because the tactics are applied in a 'wrong' order - 'wrong' for a given PO - even though all needed tactics are present. Early version of Rodin provided preferences for automatic tactics that enabled to reorder them, but the ordering was lost at each change: one could not record a particular tactic order in order to reuse it later.

Another issue is to have more than one possibility to combine the tactics. Indeed, the only implicit combination of tactics available consisted in trying to apply them in turn for every open node of a proof. In the proving area, there exists a notion of *tactic combinators*, also called *tacticals*, that allow to combine tactics in various specific manners, thus providing a sort of tactic arithmetic.

The advanced preferences for auto-tactics solved these two issues.

### 4.2.3. Isabelle Plug-in

At the beginning of the DEPLOY project, the user had three choices when valid proof obligations are 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 at 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 it does not help in others, and it requires considerable experience and a deep understanding of Rodin's theorem provers.
3. Extending Rodin's theorem provers is quite effective in principle. However, prover extensions need be definned in Java 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 therefore questionable whether the model is indeed correct.

The integration of Isabelle/HOL[1] into Rodin is one solution to this problem. Isabelle/HOL is the instantiation of the generic theorem prover Isabelle[2] to higher-order logic[3][4][5]. The main advantage of Isabelle is to provide a high degree of extensibility whilst ensuring soundness. In a nutshell, every proof in Isabelle has to be approved by Isabelle's core, which has matured over several decades and is therefore most likely correct. In Isabelle, 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 link-ups to several competition winning automated theorem provers such as Z3 [6] and Vampire[7]. 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.

### 4.2.4. ProB Disprover

ProB can be used to find counterexamples to invariant preservation and deadlock preservation proof obligations. This feature relies on the constraint solving capabilities of the ProB kernel. It is available from the ProB plugin after loading an Event-B model.

Within the context of WP2 (Transportation) ProB has also been used to try and find counterexamples to individual proof rules in the Siemens proof rule database. For this, the ProB command line tool has been extended to load proof rule files (using a Prolog parser to avoid the JVM startup time) and then applies the ProB constraint solver to try and find counter examples to each proof rule.

### 4.2.5. SMT Solver Integration

The integration of SMT solvers into the Rodin platform is motivated by two main reasons. On the one hand, the enhancement of its proving capability, especially in the field of arithmetics. On the other hand, the ability of extracting some useful informations from the proofs produced by these solvers, such as unsatisfiable cores, in order to significantly decrease the time necessary to prove a modified model.

### 4.2.6. Incorporating SAT Solvers via Kodkod

The motivation behind incorporating the relational logic solver "Kodkod" into ProB was to evaluate how other technologies can complement ProB's constraint solving capabilities. A tight integration into ProB makes it available as well for model checking, animation as well as for the disproving capabilities mentioned above. Our integration allows predicates from Event-B (and B, Z and TLA+ as well) to be solved using a mixture of SAT-solving and ProB's own constraint-solving capabilities developed using constraint logic programming: the first-order parts which can be dealt with by Kodkod and the remaining parts solved by the existing ProB kernel. We have conducted a first empirical evaluation and analyzed the respective merits of SAT-solving and classical constraint solving. We also compare to using SMT solvers via recently available translators for Event-B.

Our experiments have shown that the translation can be highly beneficial for certain kinds of constraints, and as such opens up new ways to analyze and validate formal specifications in Event-B. However, the experiments have also shown that the constraint logic programming approach of ProB can be superior in a considerable number of scenarios; the translation to Kodkod and down to SAT is not (yet) the panacea. The same can be said of the existing translations from B to SMT. As such, we believe that much more research required to reap the best of both worlds (SAT/SMT and constraint programming). An interesting side-effect of our work is that the ProB toolset now provides a double-chain (relying on technology developed independently and using different programming languages and paradigms) of validation for first-order predicates, which should prove relevant in high safety integrity level contexts.

## 4.3. Choices / Decisions

### 4.3.1. New rewriting and inference rules

During the last year of DEPLOY, rewriting (over 150) and inference rules were implemented into the built-in prover. The implementation was scheduled and prioritized according to the ratio importance/complexity of the rule.

### 4.3.2. Advanced Preferences for Auto-tactics

Since Rodin 2.1, one can create his own tactic profiles. A tactic profile allows to set and record a particular order of chosen basic tactics. Furthermore, a profile can be applied either globally (as before), or specifically for a given project.

Since Rodin 2.3, tacticals and parameterization have been added to the profiles, thus increasing the potential of such proving feature. A tactic profile may now be composed of tacticals, that combine any number of basic tactics and other profiles. The parameterization allows for example to set a custom timeout on external provers such as AtelierB P1.

### 4.3.3. Isabelle Plug-in

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. Most of the work that has been done concerned the study of Event-B theory and its translation to HOL. Unlike translations to other theorem provers (such as PP, newPP, and ML), the translation to HOL preserves provability: the translations of provable proof obligations are provable and the translations of unprovable proof

obligations are unprovable. (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 performance of the default configuration can be tweaked regarding some specific situations and 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[8] 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 could be drastically increased by declaring new inference and rewrite rules. This proceeding had an important advantage over directly entering manual proofs: by declaring new rules, we did not only make Isabelle's automated tactics prove the proof obligation under investigation, but also other proof obligations that we had not yet looked at.

### 4.3.4. ProB Disprover

Currently the constraint based invariant and deadlock checker is run from within the ProB plug-in and not from the prover interface on individual proof obligations. Indeed, for invariant preservation the disprover thus checks an event for all invariants, rather than being applied on individual proof obligations related to that event. However, ProB does know which invariants have already been proven to be preserved by that particular event. The rationale for running the invariant preservation checks in this way is that:

- the counterexample is a full valuation of the models variables and constants which can be displayed using the ProB interface,
- the invariants and guards truth values can be inspected in detail using the ProB interface, make the counter-example intelligible to the user,
- ProB can find out which of the axioms are theorems and ignore them. Indeed in the old ProB disprover approach all of the assumptions were fed, and ProB did not know which ones are relevant to find correct values for the constants and variables. In fact, some of the theorems turned out to be very complicated to check, and prevented ProB from finding counterexamples.

Similarly, the constraint-based deadlock checker is run from the ProB plug-in on a loaded model. As Rodin does not currently generate the deadlock freedom proof obligations, this was an obvious choice. Also, it enables the user to type in additional predicates to find "particularly interesting" counter examples (see ICLP'2011 article below[9]). Also, another advantage is that, again, the counter example can be inspected using the ProB interface.

### 4.3.5. SMT Solver Integration

The translation of Event-B language into the SMT-LIB language is the main issue of this integration. Two approaches were developed for this. The more efficient one is based on the translation capabilities of the integrated predicate prover of the Rodin platform (PP). It is completed by translating membership using an uninterpreted predicate symbol, refined with an axiom of the set theory. Technically, the plug-in classes uses the XProver API along with the ISimpleSequent framework, and extend the existing XProverCall, XProverInput, XProverReasoner2, AbstractLazilyConstrTactic and ITacticParametizer classes which make it easy to integrate new tactics, reasoners and external prover calls.

### 4.3.6. Incorporating SAT Solvers via Kodkod

Before starting our translation to Kodkod, we had experimented with several other alternate approaches to solve constraints in ProB. bddbddb [10] offers the user a Datalog-like language that aims to support program analysis. It uses BDDs to represent relations and compute queries on these relations. In particular, one has to represent a state of the model as a bit-vector and events have to be implemented as relations between two of those bit-vectors. These relations have to be constructed by creating BDDs directly with the underlying BDD library (JavaBDD) and storing them into a file. Soon after starting experimenting with bddbddb [10] it became apparent that due to the lack of more abstract data types than bit vectors, the complexity of a direct translation from B to bddbddb [10] was too high, even for small models, and this avenue was abandoned.

SAL is a model-checking framework combining a range of tools for reasoning about systems. The SAL tool suite includes a state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers. Some first results were encouraging for a small subset of the Event-B language, but the gap between B and SAL turned out to be too big in general and no realistic way was found to handle important B operators.

Kodkod has the advantage that is provides good support for relations and sets which play an essential role in Event-B's mathematical notation.

## 4.4. Available Documentation

Links for New rewriting and inference rules: Two lists are available on the wiki

- The list of all the defined rewrite rules[11]
- The list of all the defined inference rules[12]

Links for Advanced Preferences for Auto-tactics:

- A page concerning tactic profiles is available in the user manual[13]

Links for Isabelle Plug-in:

- A wiki page is dedicated to the Isabelle Plug-in [14].
- Some of the theoretical foundations are described in [15]
- More information will become available in [16]

Links for ProB Disprover:

- A paper on constraint-based deadlock checking had been published.[9]

Links for ProB Kodkod Integration:

- A technical report has been published in the validation using ProB and Kodkod [17]

Links for SMT Solver Integration:

- A wiki page is dedicated to the SMT Solver integration plug-in[18].

## 4.5. Status

### 4.5.1. New rewriting and inference rules

By the end of the DEPLOY project:

- there are 477 rewrite rules expressed, 423 of them were implemented. (80 were defined at the beginning of DEPLOY, 32 implemented)
- there are 115 inference rules expressed, 98 of them were implemented. (54 were defined at the beginning of DEPLOY, 51 implemented)

### 4.5.2. Advanced Preferences for Auto-tactics

Advanced Preferences for Auto-tactics are functional in Rodin 2.3. This release provides a first set of tacticals and parameterization options.

Further releases may offer additional tacticals and options, according to user feedback.

### 4.5.3. Isabelle Plug-in

During the DEPLOY project, a way of customising Rodin's theorem provers on the user level has been developed, namely the theory plug-in (see General Platform Maintenance section). 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 improves over the current theory plug-in in the following ways:

- The theory plug-in is yet unable to express certain rules involving numerals, enumerated sets, binders, and Boolean connectives whereas 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.

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

### 4.5.4. ProB Disprover

The ProB constraint-based invariant and deadlock checkers are available in the current release of ProB for Rodin. The deadlock checker has been applied successfully on a very large industrial model within WP1 (automotive). There is of course always the scope to improve the capabilities of the tool by improving the constraint solving capabilities. In particular, further simplification of existential quantifiers will be important for the deadlock checker to be applied effectively on certain models.

The Siemens proof rule database validation work is still ongoing. This work has already identified several errors in the Siemens rule database. However, a current issue is the addition of explicit well-definedness conditions into the proof rules, as well as the fact that some proof rules use substations inside predicates.

### 4.5.5. SMT Solver Integration

Since the version 0.8 of the SMT Solvers integration plug-in, released in February 2012, the SMT-Solver veriT is fully integrated and distributed within the plug-in. Therefore, it is not necessary (but still possible) to install another copy of the solver. An Auto-Tactic with SMT profile is available. Thus, it is not necessary (but still possible) to create a new tactic profile to use SMT solvers tactics. However, it can be valuable to create such new profile to parameterize the SMT tactic. The integration was successfully tested with the following solvers:

- Alt-Ergo 0.93
- Cvc3 2011-11-21
- veriT (included in the plug-in)
- z3 3.2
- Cvc4 2011-12-11 (but few theories are available for now)

This integration shown good results in the field of linear integer arithmetic with uninterpreted sort and function symbols, and good support of basic set theory. The full set theory still needs to be better supported. All the results showed the SMT-Solvers Plug-in was a good alternative to the Atelier-B provers.

### 4.5.6. Incorporating SAT Solvers via Kodkod

The Kodkod integration is available in the latest nightly build version of ProB (1.3.5-beta7). The scope of the translations is being extended, the work has been partially carried out within the ADVANCE project and will be continued within that project.

## References

[1] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, volume 2283 of Lecture Notes in Computer Science. Springer, 2002.
[2] L. C. Paulson. The foundation of a generic theorem prover. Journal of Automated Reasoning, 5(3):363-397, 1989.
[3] P. B. Andrews. An Introduction to Mathematical Logic and Type Theory. Springer,2002.
[4] A. Church. A formulation of the simple theory of types. Journal of Symbolic Logic,5(2):56-68, 1940.
[5] M. J. C. Gordon and T. F. Melham. Introduction to HOL. Cambridge University Press, 1993.
[6] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In TACAS, volume 4963 of Lecture Notes in Computer Science, pages 337-340. Springer, 2008.
[7] A. Riazanov and A. Voronkov. The design and implementation of Vampire. AI Communications, 15(2-3):91-110, 2002.
[8] K. Varpaaniemi. Bepicolombo models v6.4 http://deploy-eprints.ecs.soton.ac.uk/244
[9] Constraint-Based Deadlock Checking of High-Level Specifications. In Proceedings ICLP'2011, Cambridge University Press, 2011. http://www.stups.uni-duesseldorf.de/w/Special:Publication/HaLe2011
[10] http://bddbddb.sourceforge.net/
[11] http://wiki.event-b.org/index.php/All_Rewrite_Rules
[12] http://wiki.event-b.org/index.php/Inference_Rules
[13] http://handbook.event-b.org/current/html/preferences.html
[14] http://wiki.event-b.org/index.php/Isabelle_for_Rodin

[15] http://www.springerlink.com/content/b7286q3k52180v68 Matthias Schmalz. Term Rewriting in Logics of Partial Functions. ICFEM 2011
[16] Formalizing the Logic of Event-B. Partial Functions, Definitional Extensions, and Automated Theorem Proving. Submitted as PhD thesis to ETH Zurich.
[17] http://www.stups.uni-duesseldorf.de/w/Special:Publication/PlaggeLeuschel_Kodkod2012 Validating B,Z and TLA+ using ProB and Kodkod. Technical Report, 2012.
[18] http://wiki.event-b.org/index.php/SMT_Solvers_Plug-in

# 5 Code Generation

## 5.1. Overview

The code generation plug-in allows the generation of code for typical real-time embedded control software from refined Event-B models. Such a feature will be an important factor in ensuring eventual deployment of the DEPLOY approach within industrial organisations.

## 5.2. Motivations

DEPLOY industrial partners are interested in the formal development of multi-tasking, embedded control systems. During the project, work has been undertaken to investigate automatic generation, from Event-B models, for these type of systems [1][2][3][4][5]. Initially, we chose to translate to the Ada programming language[6], and use it as a basis for the abstractions used in our approach.
We described our previous code generation feature as a demonstrator tool; chiefly a tool designed as a proof of concept, used by us to validate the approach. In this sense, the tool as it stands now, is the first prototype intended to be used by developers, and thus motivated numerous evolutions and new features such as:

- allowing a more seamless edition of tasking Event-B,
- supporting extensibility,
- supporting other target languages than just Ada,
- supporting programming concepts reification using abstract translators.

## 5.3. Choices / Decisions

Considering the demonstrator as a baseline, we can list the new features as follows:

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

These evolutions will be detailed hereafter. Moreover, the code generators have been completely re-written. The translators are now implemented using Java only. In our previous work we attempted to make use of the latest model-to-model transformation technology, available in the

Epsilon tool set[7], but we decided to revert to Java since Epsilon lacked the debugging and productivity features of the Eclipse Java editor.

### 5.3.1. Tasking Event-B and its edition

A text-based task body editor was added, to minimize the amount of editing required with the EMF tree editor. The task body editor is associated with a parser-builder; after the text is entered in the editor the EMF representation is generated (by clicking a button) that is, assuming parsing is successful. If the parser detects an error, information about the parse error is displayed in an adjoining text box. When specifying events in the task body, there is no longer a need to specify two events involved in a synchronization. The code generator automatically finds the corresponding event of a synchronization, based on the event name, and using the composed machine component [8]. Composed machines are used to store event 'synchronizations', and are generated automatically during the decomposition process. This reduces the amount of typing in the TaskBody editor, since we no longer need to specify both local and remote (synchronizing) events. The new feature also overcomes the 'problem' that we previously experienced, with duplicate event names in a development, and event selection, when specifying the task body. The EMF tree editor in Rose is now only used minimally; to add annotations for Tasking, Shared and Environ Machines; typing annotations, and parameter direction information; and sensing/actuating annotations, where necessary. Further work is under way to integrate the code generation feature with the new Rodin editor.

### 5.3.2. Allowing Extensibility

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

### 5.3.3. Targeting new Languages

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

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

### 5.3.4. Using Abstract Translators

To test our approach, we have built translators for C, Ada and Java using the same underlying abstract translators. Additionally we consider the case where a new language may be required that has only modest differences to an existing language. A good example of this is to consider the case where a different library may want to be used from one used in an existing translation. For instance in C, concurrency can be achieved through different mechanisms such as OpenMP[9] or Pthreads[10]. In this case it may be that all but the mechanism for handling a subroutine call are the same, meaning that the majority of the translation can occur using common translators, with separate translators for the different methods of handling a subroutine call. To allow for this we allow the developer to assign a core and specialisation language to each translator they build. In cases where a translator for the specialisation language does not exist, the translator will automatically defer to the default core language translator, if one exists. This means that default translators for a particular core language can be written for the majority of the translation, with specialisations being provided where differences occur. The core and specialisation of the language is also reflected in the theory translator, meaning that language theories are only required for the core languages, rather than for each individual specialization.

## 5.4. Available Documentation

We have updated the documentation[11], including the Tasking Event-B Overview[12], and Tutorial [13] materials.

## 5.5. Status

We released a new version of the code generator on 22-03-2012. The changes were made to the methodology, user interface, and tooling. The first version of the code generator supported translation to Ada, and the current version also has limited support for C.

## References

[1] http://eprints.soton.ac.uk/270824/"Edmunds, Andrew and Butler, Michael (2010) Tool Support for Event-B Code Generation. In, WS-TBFM2010"
[2] http://eprints.soton.ac.uk/272006/"Edmunds, Andrew and Butler, Michael (2011) Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In, PLACES 2011, Saarbrucken, Germany"
[3] http://eprints.soton.ac.uk/272771/"Edmunds, Andrew, Rezazadeh, Abdolbaghi and Butler, Michael (2011) From Event-B Models to Code: Sensing, Actuating, and the Environment. At SBMF2011, Sao Paulo, Brazil, 28 - 26 Sep 2011."
[4] http://eprints.soton.ac.uk/335400/"Edmunds, Andrew, Rezazadeh, Abdolbaghi and Butler, Michael (2012) Formal modelling for Ada implementations: tasking Event-B. In, Ada-Europe 2012: 17th International Conference on Reliable Software Technologies, Stockholm, SE, 11 - 15 Jun 2012. 14pp. (In Press)"
[5] http://eprints.soton.ac.uk/336226/"Edmunds, Andrew, Butler, Michael, Maamria, Issam, Silva, Renato and Lovell, Chris (2012) Event-B code generation: type extension with theories. In, ABZ 2012, Pisa, IT, 19 - 21 Jun 2012. 4pp. (In Press)"
[6] J. Barnes. Programming in Ada 2005. International Computer Science Series. Addison Wesley, 2006.
[7] http://www.eclipse.org/epsilon/
[8] http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B
[9] http://openmp.org/wp/
[10] https://computing.llnl.gov/tutorials/pthreads/
[11] http://wiki.event-b.org/index.php/Code_Generation_Activity
[12] http://wiki.event-b.org/index.php/Tasking_Event-B_Overview
[13] http://wiki.event-b.org/index.php/Code_Generation_Tutorial

# 6 Model-based testing

## 6.1. Overview

Model-based testing (MBT) is an approach from software engineering that uses formal models as basis for automatic generation of test cases. A test case is defined as a sequence of actions (or events, or triggers) together with corresponding test data that can be executed against a System Under Test (SUT). In DEPLOY, we investigated a version of MBT using Event-B models as test models from which test cases are generated. The main purpose was to provide the deployment partners with an MBT method together with a Rodin plug-in that allows generation of test cases satisfying different coverage criteria (e.g. covering of all events in a model or covering paths to a set of target global states). This includes the generation of appropriate test data that satisfy the guards of the single test steps.

The partners involved in MBT for Event-B were University of Pitesti and University of Düsseldorf on the academic side and SAP on the deployment partners side.

In the last year of DEPLOY, the work done on MBT proceeded according to the plan (summarized in the previous deliverable D32), i.e. generating test cases and test data using constraint-solving and evolutionary techniques. Moreover, new ideas involving a technique called "automata learning" were investigated. The proposed methods were implemented, a new Rodin plugin, the MBT plug-in [1], was publicly released, and experiments were performed using the Event-B models available in the DEPLOY model repositories.

## 6.2. Motivations

MBT is a very active research area using several types of test models and delivering promising results. Before our investigations, the Event-B approach and framework did not support MBT, even though the MBT prerequisites were satisfied. Moreover, deployment partners such as SAP, showed interest into MBT as a good candidate for adoption of formal methods in practice. A first attempt to MBT was based on the model-checking and state exploration capabilities of the ProB plugin, by exploring the explicit state of an Event-B model and outputting event sequences as test cases. However, the attempt worked only for small domains of the variables and faced the classical state space explosion problem for larger domains. This motivated the search for alternative solutions as presented below.

## 6.3. Choices / Decisions

Over the reporting period of the last year, the following three different solutions were defined, implemented, and tested:

- Using abstraction and constraint-solving: To avoid the state space explosion due to the large bounds of the variables, this approach ignores these values in the first step and uses the ProB model-checker only to generate abstract test cases satisfying the given coverage criteria. However, these paths may be infeasible in the concrete model due to the data constraints along the path. The solution is to represent the intermediate states of the path as existentially quantified variables. The whole path is thus represented as a single predicate consisting of the guards and before-after predicates of its events. ProB's improved constraint solver is then used

to validate the path feasibility and find appropriate test data satisfying the constraints.

- Using model-learning: Event-B models are essentially abstract state machines. However, their states are not given explicitly; instead they can be implicitly derived from the values of the model variables. Since the notion of state is at the heart of MBT, we provide a model-learning approach that uses the notion of cover automata to iteratively construct a subset of a state space together with an associated test suite. The iterative nature of the algorithm fits well with the notion of Event-B refinement. It can also be adapted to work with decomposed Event-B models.
- Using search-based techniques: We also investigated meta-heuristics search approaches, including evolutionary algorithms, in the attempt to tackle large variable domains and state spaces in the search of test data. We worked on two applications: first, generation of test data for a given sequence of events (test case) and second, a generalisation to generation of test suites satisfying different coverage criteria.

Furthermore, the output of the above procedures was complemented by test suite optimisations (based on evolutionary techniques) according to different test coverage criteria. This was motivated by the fact that the test generators sometimes produces too many feasible tests. This may be useful for certain types of intensive testing (e.g. conformance testing), but in other scenarios it is more appropriate to generate smaller test suites for lighter coverage criteria (e.g. each event is executed at least once).

## 6.4. Available Documentation

MBT plug-in wiki page: http://wiki.event-b.org/index.php/MBT_plugin

Associated deliverables containing technical information on the MBT work:

- DEPLOY deliverable D44 - see chapter 10 for technical details on the constraint-solving approach
- DEPLOY deliverable D54 - technical details on the search-based and model-learning approaches
- DEPLOY deliverable D32 - see chapter 10 for backward reference

Papers:


- Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Test data generation for Event-B models using genetic algorithms. In Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS’11), volume 181 of CCIS, pages 76–90. Springer, 2011.
- Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Towards search-based testing for Event-B models. In Proc. of 4th Workshop on Search-Based Software Testing (SBST’11), pages 194–197. IEEE, 2011.
- Ionut Dinca. Multi-objective test suite optimization for Event-B models. In Proc. of Int. Conf. on Informatics Engineering and Information Science (ICIEIS’11), volume 251 of CCIS series (Communications in Computer and Information Science), pages 551–565. Springer, 2011.
- Florentin Ipate, Ionut Dinca, and Alin Stefanescu. Model learning and test generation using cover automata. Submitted to IEEE Transactions on Software Engineering, January 2012.
- Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. Learn and test for Event-B - a Rodin plugin. Accepted at ABZ'12, LNCS, Springer, 2012.
- Ionut Dinca, Florentin Ipate, and Alin Stefanescu. Learning and test generation for Event-B decomposition. Submitted to ISoLA'12, LNCS, Springer, 2012.

## 6.5. Status

We consider that the MBT work has reached a rather mature level at the end of the project, by exploring different techniques and applying them successfully to the available Event-B models from the DEPLOY repository. Moreover, SAP has experimented with one of the methods (based on abstraction and constraint solving) via a web service in their testing infrastructure. The dissemination of the results was continuous by presentations of the results at different conferences, workshops, or research seminars and as conference and journal papers.

### References

[1] http://wiki.event-b.org/index.php/MBT_plugin

# 7 Model Checking

## 7.1. Overview

We have implemented various improvements to the model checking and constraint-solving components of ProB, most in reaction to issues arising in the industrial case studies. These improvements are described below.

### 7.1.1. State Space Reduction, Compression and Hashing

Driven by a case study from the space sector (a protocol modeled by SSF), where memory consumption was an issue, we have investigated ways to reduce ProB's memory consumption. A first step was to implement a first version of state compression, whereby we simplify stored states so that they require less memory. This was achieved without compromising speed and is now always activated. Furthermore, if the preference `COMPRESSION` is set to true, then ProB will also detect common (sub-)expressions in states and store the common expressions only once. For example, when several states have the same same value for a given variable `x` then its value will only be stored just once. This is particularly useful when complicated variables only change infrequently.

Related to this aspect, the hashing of ProB states was improved on 64-bit architectures, which is also important in the context of detecting common subexpressions for sharing (common subexpressions are detected by hashing). We have also implemented a cryptographic SHA1 hashing function for Prolog terms, but it is not yet used in the production version of ProB.

Finally, the most useful symmetry reduction technique of ProB is the so-called "hash marker" method. Here, we have also improved the computation of the hash symmetry markers, both achieving a reduction in size and runtime.

### 7.1.2. Constraint-Based Checking

Improved constraint-based checking for deadlocks, invariants and event sequences (used in MBT). In addition, many improvements in constraint solving kernel were implemented.

### 7.1.3. New Features

**Experiments with Theory Plugin**

Animating models that use the Theory plug-in is currently not possible. However, we prepared ProB's translation from Event-B into its internal representation in order to support user defined operators from the Theory plug-in. The work on the feature is currently suspended until the Theory plug-in supports accessing the definitions.

**Improved detection of infinite functions, and improved support for them**

ProB now detects much better when comprehension sets or lambda abstractions are infinite. In this case the set or function is kept symbolic and evaluated on demand.

In that light, ProB now also keeps track when an infinite set had to be approximated by a finite one and emits a warning. For example, given a predicate `!x.(x:NATURAL => x+1:NATURAL1)`, ProB will check the quantified expression only for values of x from 0 to MAXINT. These warnings are not yet displayed in the Rodin version, but are visible in probcli

and ProB Tcl/Tk.

**Support of finite operator**

The finite operator was previously ignored, as ProB only supported finite sets anyway. Now, however, ProB can deal with certain infinite sets (in particular infinite functions such as `%.x(x:INTEGER|x+1)`), this could lead to wrong results. ProB now supports the finite operator, and emits a warning if it cannot determine whether a comprehension set or lambda function is finite.

### 7.1.4. Experiments

TLA+ and B share the common base of predicate logic, arithmetic and set theory. We have conducted a translation of the non-temporal part of TLA+ to B, which makes it possible to feed TLA+ specifications into existing tools for B. Part of this translation must include a type inference algorithm, in order to produce typed B specifications. There are many other tricky aspects, such as translating modules as well as let and if-then-else expressions. We also developed an integration of our translation into ProB. ProB thus now provides a complementary tool to the explicit state model checker TLC, with convenient animation and constraint solving for TLA+. More importantly for DEPLOY, we have conducted a series of case studies, highlighting the complementarity to TLC. In particular, we have highlighted the sometimes dramatic difference in performance when it comes to solving complicated constraints in TLA+. Also, the benchmarks have allowed us to pinpoint some areas where ProB could be further improved. This also led to some of the compression techniques mentioned earlier.

## 7.2. Motivations

The motivations for the state space compression arose from experiments in WP3 (space), and from applications of ProB outside of DEPLOY.

The need for constraint-based deadlock checking arose in the automotive work package, more precisely during the elaboration of the cruise control system. Here, proving turned out to be impractical and ProB was used to find deadlocks and guide the development of the model.

The work on infinite functions was motivated by models from the transportation section (WP2), where many models contained infinite functions encoding certain auxiliary computations.

## 7.3. Choices / Decisions

Aggressive compression can also induce a performance penalty. The new default mode was chosen such that there should be no performance penalty, with reduced memory usage. (Indeed, the time for compression is regained by reduced time to store and retrieve the states.)

A more aggressive setting can be forced by `-p COMPRESSION TRUE`. This will further reduce memory consumption, but may increase runtime (although quite often it does not).

## 7.4. Available Documentation

- A manual is available on using command line version of ProB [1]
- A paper has been published on constraint-based deadlock checking of high-level specifications[2]
- A report[3] on the translation from TLA+ to B and experiments with TLC has been published.

## 7.5. Status

All improvements, unless explicitly stated below, are accessible in the latest release of ProB. For example, the improved hashing and light-weight state compression is available in the current release of ProB. The SHA1 hash technique is not available in the current release. The research on further compression technique is ongoing and will be continued within the project ADVANCE[4]. The work on the Theory plug-in support is currently suspended until the Theory plug-in provides the features described earlier.

## References

[1] http://www.stups.uni-duesseldorf.de/ProB/index.php5/Using_the_Command-Line_Version_of_ProB
[2] http://www.stups.uni-duesseldorf.de/w/Special:Publication/HaLe2011 Constraint-Based Deadlock Checking of High-Level Specifications. In Proceedings ICLP'2011, Cambridge University Press, 2011.
[3] http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschelTLA2012
[4] http://www.advance-ict.eu/