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

DEPLOY Deliverable D23

D9.2 Model Construction Tools and Analysis Tools II

Editor: *Carine Pascal (Systerel)*
Internal reviewers: *Thierry Lecomte (ClearSy), Michael Leuschel (Düsseldorf University)*
Contributors: *Nicolas Beauger (Systerel), Jens Bendisposto (Düsseldorf University), Michael Butler (Southampton University), Andreas Fürst (ETH Zurich), Alexei Iliasov (Newcastle University), Michael Jastram (Düsseldorf University), Issam Maamria (Southampton University), Daniel Plagge (Düsseldorf University), Renato Silva (Southampton University), Laurent Voisin (Systerel)*

Public Document

29 January 2010

http://www.deploy-project.eu

Contents

1 Introduction

The deliverable D23 of the DEPLOY project is split into two parts:

- The Rodin core platform and plug-ins (i.e. the DEPLOY tools).
- This document.

The Rodin platform is available from the SourceForge site ([1]). The tool documentation is provided within the Event-B wiki ([2]).

This document gives a description of the work that was carried on during the second year of the DEPLOY project (Feb 2009-Jan 2010), in the course of the WP9 *Tooling research and development* work package, and brings new perspectives for the coming year.

In particular, the WP9 partners have strovin to meet the following objectives:

- Improved scalability of the Rodin platform to support industrial deployments, through GUI enhancements (smart completion, renaming, text editing, etc), decomposition support and design-pattern management.
- Prover integrity and performance, to enhance the confidence in provers and to enlarge their proving capabilities. To this aim, the existing provers have been improved and a new rule-based prover plug-in has been implemented.
- Model animation and testing, to validate Event-B models. More precisely, the ProB or AnimB plug-ins allow a domain expert to detect errors in a model and ensure the presence of desired functionalities. Moreover, it is very important for many industrial applications to be able to completely hide the underlying formal specification.
- Model checking (ProB), to enable users to find sequences of events that prevent safety properties or proof obligations to be fulfilled.
- UML integration. UML-B provides a diagrammatic, formal modelling notation based on UML.
- Code generation, to enable complete support for development, from high-level Event-B models down to executable implementations. An initial definition of language support for code generation has been put forward.

This document covers the following items: general platform maintenance, UML-B improvements, ProB improvements, text editor plug-in, decomposition plug-in, initial definition of language support for code generation, improvements to existing provers, rule-based prover, pattern plug-in, flow plug-in and modularisation plug-in.

For each of these newly implemented features or improvements, the document is structured as follows:

- Overview. The involved partners are identified and an overview of the contribution is given.
- Motivations. The motivations for each tool extension and improvement are expressed.
- Choices / decisions. The decisions (e.g. design decisions) are justified.
- Available documentation. Some pointers to the Event-B wiki or related publications are listed.
- Planning. A timeline and the current status (as of 29 Jan 2010) is given.

References

[1] http://sourceforge.net/project/showfiles.php?group_id=108850&package_id=181714
[2] http://wiki.event-b.org

2 General Platform Maintenance

2.1. Overview

The purpose of the platform corrective and evolutive maintenance is to address bugs and feature requests reported either by mail or through the appropriate trackers on SourceForge.

The noticeable new features in the main platform for the past year are listed below:

- Mathematical Language V2 (releases 1.0 and upper)

  The new version of the mathematical language is supported.

  See Event-B Mathematical Language [1].
- Theorems everywhere (releases 1.0 and upper)

  It is possible to mix theorems and regular predicates in axioms, invariants and guards.
- Auto-completion (releases 1.0 and upper)

  When entering a predicate or expression in the Event-B machine / context editor, it is possible to type `C-Space` to see a list of possible identifiers that could be entered at the cursor position.
- Entering mathematical symbols (releases 1.1 and upper)

  The Rodin platform provides many more ways to enter mathematical symbols:

  - either type the ASCII shortcut (as in previous releases),

  - or type the LaTeX command (as defined in style `bsymb`),

  - or click in the *Symbol Table* view which displays the symbols graphically,

  - or directly enter the Unicode value of the symbol (for advanced users).

  See Rodin Keyboard [2].

See the Release Notes [3] and the SourceForge [3] databases (bugs and feature requests) for details about the previous and upcoming releases of the Rodin platform.

2.2. Motivations

The main evolutions of the Rodin platform are driven by the description of work for the DEPLOY project and the requirements expressed by industrial WP1 to WP4 partners or by advanced users during the lifecycle of the project.

Beyond that, any user registered on SourceForge may record any encountered bug on the Rodin platform or request a new feature, using the dedicated trackers [3]. Depending on the category, the bug / feature is assigned to the WP9 partner who is in charge of processing it:

Category	Partner
AnimB	Christophe METAYER
B2LaTeX	University of Southampton
Decomposition	Systerel
Event-B core	Systerel
Event-B interface	Systerel

Event-B POG	Systerel
Event-B provers	Systerel
Event-B static checker	Systerel
PRO-B	Dusseldorf
Renaming	University of Southampton
Requirements	Dusseldorf
Rodin platform	Systerel
Text editor	Dusseldorf
U2B	Southampton

The priorities are discussed during the WP9 meetings (bi-weekly management conference call, WP9 face-to-face meetings during DEPLOY workshops).

2.3. Choices / Decisions

The WP9 partners have agreed on a release policy (see the Rodin Platform Releases [3] wiki page). In particular:

- A new version of the Rodin platform is released every 3 months.
- The code is frozen during the 2 weeks preceding each release.
- The Eclipse versioning policy is enforced (See Version Numbering [4]).
- A wiki page is dedicated to each release.

The main advantages, for both developers and end-users, are summarized below:

- Information. The wiki page dedicated to each release provides instant information on the new features and improvements, which may be discussed if necessary.
- Validation. The period of code freeze is more especially devoted to bug fixes, and the frequency of the stable releases is ensured.
- Integration. A synchronization between the optional plug-ins and other plug-ins is now possible.

2.4. Available Documentation

The following pages give useful information about the Rodin platform releases:

- Release notes.

  See Rodin Platform Releases [5].

  More details are provided in the notes distributed with each release (eg. [6]).

- Bugs.

  See [7].

- Feature requests.

  See [8].

2.5. Planning

The Rodin Platform Releases [3] wiki page lists in particular the upcoming releases and give the scheduled release dates.

Special efforts will be made on the following topics, which are requested by all users in an industrial context:

- Mathematical Extensions.

  Currently, the operators and basic predicates of the Event-B mathematical language supported by the Rodin platform are fixed. The purpose is to extend the platform to support user-defined data types and associated operators, including inductive data types. Users will then be able to define operators of polymorphic type as well as parameterised predicate definitions.

- Team-based Development.

  The purpose is to perform simultaneous developments.

  The Decomposition plug-in [9] gives an answer to this requirement by allowing to cut a model in sub-models which may be handled independently. In the same manner, the EMF Compare Editor [10] enables the comparison of machines and contexts: it is a first step to be able to use the Rodin platform in a team environment by putting a code repository (e.g., Subversion) underneath it.

  In order to understand the problem properly, some usage scenarios for team-based development [11] and for merging proofs [12] have already been written. Moreover, a page has been initiated to remember the main requirements (see Teamwork Requirements [13]). These pages provide a basis for brainstorming and further developments on the topic.

- Documentation.

  The purpose is to continuously increase and improve available documentation on the Wiki. It may contain requirements, pre-studies (states of the art, proposals, discussions), technical details (specifications), teaching materials (tutorials), user's guides, etc. The intended audience may be developers or end-users.

References

[1] http://wiki.event-b.org/index.php/Event-B_Mathematical_Language
[2] http://wiki.event-b.org/index.php/Rodin_Keyboard
[3] http://wiki.event-b.org/index.php/D23_General_Platform_Maintenance#Available_Documentation
[4] http://wiki.eclipse.org/index.php/Version_Numbering
[5] http://wiki.event-b.org/index.php/Rodin_Platform_Releases
[6] http://sourceforge.net/project/shownotes.php?release_id=693928
[7] http://sourceforge.net/tracker/?atid=651669&group_id=108850
[8] http://sourceforge.net/tracker/?group_id=108850&atid=651672
[9] http://wiki.event-b.org/index.php/D23_Decomposition
[10] http://wiki.event-b.org/index.php/EMF_Compare_Editor_installation
[11] http://wiki.event-b.org/index.php/Scenarios_for_Team-based_Development
[12] http://wiki.event-b.org/index.php/Scenarios_for_Merging_Proofs
[13] http://wiki.event-b.org/index.php/Teamwork_Requirements

3 UML-B Improvements

3.1. Overview

This part of the deliverable describes improvements to the UML-B plug-in feature, which is the responsibility of University of Southampton.

A new plug-in feature has been developed to provide animation of UML-B state-machine diagrams. This feature was developed by University of Southampton.

The longer term development of UML-B relies on an EMF representation of Event-B. The development of a new EMF Event-B plugin-in feature is also described in this section. This feature was initially developed by University of Southampton, Heinrich-Heine University, Düsseldorf and University of Newcastle. It is now mostly maintained and developed by University of Southampton.

3.2. Motivations

3.2.1. UML-B Support for State machine Refinement

The current version of the UML-B tool has been improved to support the refinement of state-machines. At the last deliverable, refinement of classes was supported and state-machine refinement was beginning to be investigated. The investigation has experimented several notation and methodological alternatives. The design has now been finalised and an implementation has been achieved. State-machines can be refined by adding nested state-machines inside states. Some of the transitions in the nested state-machine do not represent new events but contribute to the refinements of existing transitions in the parent state-machine. A concept of *transition elaboration* has been invented to represent this relationship.

3.2.2. UML-B General Improvements

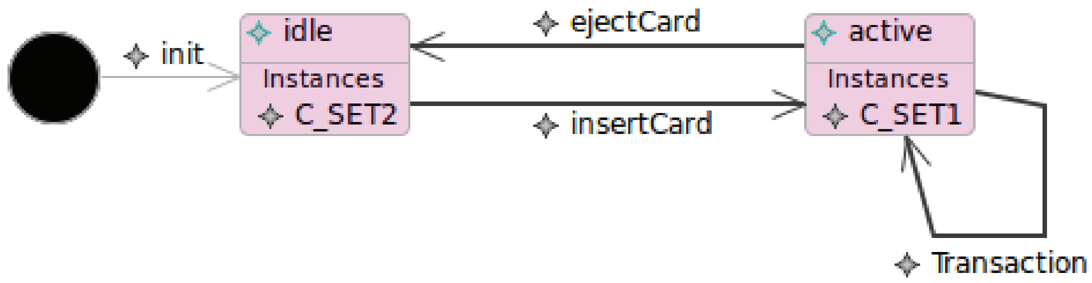
Many other minor improvements have been made to the UML-B tool including:

- collapsing empty compartments on diagrams,
- improved navigation between diagrams,
- improved properties views,
- ability to order classes and class-types in the output translation,
- support for theorems everywhere (i.e. invariants and axioms can now be designated as theorems).

3.2.3. UML-B State-machine Animation

This feature was developed in response to a requirement from Siemens Transportation. Several state-machines can be selected (representing refinements and hierarchical nesting) for simultaneous animation. The animation relies on Pro-B animation of the corresponding Event-B models (that have been automatically generated by UML-B). The animated diagrams show the currently active states and the enabled transitions. Events can be 'fired' by clicking on the enabled transition. Where the state-machine belongs to a class, instances of the class can be seen moving from state to state.

The screenshot below shows a simple statemachine being animated. Two class instances are currently active; one models an ATM that is not available (e.g. in maintenance) and the other is in the process of validating a card. Transitions that are enabled for one or other of the class instances are emboldened. (The instance to be used when a transition is fired is selected when the transitions is clicked upon).



A larger screenshot of refined and nested statemachines being animated in parallel is shown here:

Statemachine Animation Large Screenshot [1]

3.2.4. EMF Framework for Event-B

An EMF (Eclipse Modelling Framework) based representation of Event-B was developed and made available as a plug-in feature for Rodin. This enables Event-B machines and contexts to be loaded into EMF based tools. Serialisation (i.e. loading and saving) is performed via the Rodin API. This feature can be viewed as an *enabling technology*. Hence motivation derives from several sources including:

- A Text editor was requested by several industrial and academic partners - A fully-featured EMF based text editor (**Camille**) has been developed by Düsseldorf and is now available.
- Team-working facilities are required by all industrial partners (particularly Bosch and SSF) - EMF Compare/merge tools are now under investigation to support a **teamworking** repository plug-in feature.
- **UML-B integration** - since UML-B is based on EMF, the development of an EMF representation of Event-B enables UML-B concepts to be added as extensions.

3.3. Choices / Decisions

3.3.1. UML-B Support for State machine Refinement

The methods and modelling notations for refinement in UML-B were developed by experimentation using a case study of an ATM. The use of hierarchical nested state-machines (which were already available in UML-B) as a technique for adding detail in refinement was quickly adapted by making changes to the meta-model and translation. This technique was found to be suitable. Some further experimentation was needed in order to understand the need to link transitions of the nested state-machines with those in their parent. A concept of *elaboration* was introduced, whereby an elaborating transition contributes guards and actions to the event produced from the elaborated parent transition. Transition splitting (analogous to event splitting in Event-B refinements) is a natural consequence of refinement of states. An idea to *bundle* the split transitions in the parent state-machine so that the correspondence with the abstract refined state-machine is more obvious has not been pursued for now since it would add complication to the tooling.

3.3.2. UML-B State-machine Animation

Initially, we attempted to model the animation state information as an extension to the UML-B meta-model. We discovered technological difficulties in extending EMF models in this way. Therefore, we adopted an alternative solution using an independent meta-model of animation diagrams. These replicate parts of the structure of UML-B but add meta-properties to model the animation. When a model is to be animated, an animation model is constructed programmatically to match the UML-B model. Thereafter, the animation runs independently of UML-B. This has the additional benefit that the diagram can be simplified and tailored to better suit animation. For example, removing the editing palette.

3.3.3. EMF Framework for Event-B

The structure of the EMF meta-model for Event-B was studied in great detail. Various options for sub-packaging the model were tried but it has been found that it is more convenient for users to keep a simple package structure. Currently this consists of three packages; a *core* package containing abstract basis and project level meta-model, a *machine* package and a *context* package. A flexible abstract basis has been derived through experimentation. The abstract basis consists of an inheritance hierarchy of abstract meta-classes which provide great flexibility for writing code that deals with the meta-model in as generic a manner as possible. A driving factor in the design was to support both project level tools and component level tools. The latter should be able to manipulate a single machine or context without loading referenced components. This was achieved customising the EMF proxies (used in references) so that they are calculated lazily (when a request to resolve is received).

3.4. Available Documentation

UML-B Refinement is described in a paper which was presented at the FM2009 conference in Eindhoven. It is available here:

Language and Tool Support for Class and State Machine Refinement in UML-B [2]

A tutorial on how to refine state-machines is available on the wiki:

Refinement of Statemachines [3]

State-machine animation is described on the wiki here:

UML-B - Statemachine Animation [4]

It is also available as a short paper here:

Animation of UML-B State-machines [5]

The EMF Framework for Event-B is described on the wiki here:

EMF framework for Event-B [6]

It is also available as a short paper here:

An EMF Framework for Event-B [7]

3.5. Planning

UML-B integration:

- Develop extensibility mechanisms for EMF Event-B framework via experimentation with structured data (records) plug-in.
- Re-engineer UML-B context diagrams as a diagrammatic view of records.
- Re-engineer UML-B package diagram based on EMF Event-B framework.

References

[1] http://wiki.event-b.org/index.php/Statemachine_Animation_Large_Screenshot
[2] http://eprints.ecs.soton.ac.uk/18268
[3] http://wiki.event-b.org/index.php/Refinement_of_Statemachines
[4] http://wiki.event-b.org/index.php/UML-B_-_Statemachine_Animation
[5] http://eprints.ecs.soton.ac.uk/18261
[6] http://wiki.event-b.org/index.php/EMF_framework_for_Event-B
[7] http://eprints.ecs.soton.ac.uk/18262

4 ProB Improvements

4.1. Overview

This part of the deliverable describes improvements of the ProB animation and model checking plug-in.

The improvements and development of ProB were mainly carried out by University of Düsseldorf, with some support by the University of Southampton. Furthermore, the work was driven by requirements of Siemens and SAP; some tool development was also undertaken by SAP.

New features:

- Multi-level animation and validation.
- B-Motion Studio.
- Disprover Support.
- First steps towards test-case generation.

Improvements:

- Scalability improvements driven by Siemens and SAP applications.
- Using proof information to improve model checking.

Other works:

- First steps towards validation of ProB for usage by Siemens in SIL-4 chain.
- Evaluation against SAT/SMT/BDD-based approaches.

4.2. Motivations

4.2.1. Multi-level Animation and Validation

Thus far ProB only allowed single-level animation, i.e. the animator would animate a single refinement level in isolation. This meant that ProB was not able to detect a large class of potential errors in the model:

- A broken gluing invariant.
- An invalid witness.
- Violation of guard strengthening.
- Violation of variant decrease (resp. decrease or stability) for convergent (resp. anticipated) events.

The new validation algorithm now can animate a range of refinements together. The user can decide which levels are to be animated together. As such, all of the above errors can now be detected by ProB. User experience is also improved, as he or she can inspect also the abstract variables. The new algorithm has been successfully applied to various case studies, and thus far up to 14 levels have been animated concurrently without problem.

4.2.2. B-Motion Studio

It is often very important to be able to show a formal model to a domain expert or manager, not versed in formal methods. For example, only a domain expert will be able to detect certain mistakes in the formal model. To enable to easily and quickly build graphical visualisations of Rodin models, we have developed B-Motion Studio. B-Motion Studio comes with a graphical editor to arrange graphical components and link them with the formal model. No new programming language has to be learned: the linking is described in B itself. To run a graphical visualisation, the ProB animator is used.

4.2.3. Test-Case Generation

During deployment in the SAP workpackage it became clear that test-case generation from the Event-B models is required for success. In this task, we have developed a first algorithm for test-case generation, which ensures complete transition coverage of a high-level model, and translates the test-cases into traces of a refined model, so that the tests can be run on the "real" system. Optimisations, to reduce the length and number of test cases, as well as to minimise race conditions, have been implemented.

4.2.4. Scalability, Application by Siemens and Validation

We tackled a case study in WP2, which centres on the San Juan metro system installed by Siemens. The control software was developed and formally proven with B. However, the development contains certain assumptions about the actual rail network topology which have to be validated separately in order to ensure safe operation. For this task, Siemens has developed custom proof rules for AtelierB. AtelierB, however, was unable to deal with about 80 properties of the deployment (running out of memory). These properties thus had to be validated by hand at great expense (and they need to be revalidated whenever the rail network infrastructure changes).

The motivation then was to try and use ProB for this task. This required a considerable amount of work on improving the scalability of the ProB kernel, to be able to deal with large sets and relations. The ProB parser and type checker also had to be re-developed to be able to deal with large industrial specifications.

The case study was a success: ProB was able to validate all of the about 300 properties of the San Juan deployment, detecting exactly the same faults automatically in around 17 minutes that were manually uncovered in about one man-month.

This leads to the next task: the issue of validating ProB, so that it can be integrated into the SIL4 development chain at Siemens.

4.2.5. Proof Directed Model Checking

In order to improve the performance and scalability of animation and model checking with ProB, we want to make use of the proof information available in an integrated platform such as Rodin. In particular, we can use the information about which POs have already been discharged to simplify the task of the model checker and to guide the order in which it evaluates states.

Our first implementation has shown significant reduction of the model checking effort for industrial applications. It uses proof information obtained from Rodin to remove invariants that are proven to hold. Actually, if unproven(e) is the unproven part of the invariant for event e, and a state is reachable via events e and f, we only need to check the intersection of unproven(e) and unproven(f) (see Jens Bendisposto, Michael Leuschel. Proof Assisted Model Checking for B. ICFEM 2009. [1]).

4.2.6. SAT/SMT/Kodkod

In this subtask we investigate alternate approaches to validate high-level B models using techniques and tools based on BDDs, SAT-solving and SMT-solving. The overall motivation is to improve the scalability of the animator and model checker.

When searching for variable values that satisfy a certain predicate, experience showed that ProB performs best when some values are already known, as in finding state transitions where all values of the previous state are known. When looking for constants that satisfy the axioms or when trying to find counterexamples for proof obligations (see next Disprover paragraph), SAT resp. SMT solving techniques look very promising to support ProB's constraint solving approach.

4.2.7. Disprover

In order to help the user, we wanted to make it possible to apply ProB to individual proof obligations. In some cases, this enables proving a sequent by exhaustive case analysis. Also, if ProB finds a counterexample, the user gets important feedback: the proof obligation cannot be discharged, along with a reason why.

4.3. Choices / Decisions

One important choice for ProB is to use the same constraint solving kernel and interpreter for B, Event-B and Z. This decision, allows us to maintain a tool capable of animating and model checking these three formalisms. It also gives us a much wider range of test cases. For example, most of the regression tests of ProB come in the form of B machines (Rodin archives are much "cumbersome" as a basis of regression tests: they need to be upgraded and imported into workspaces). This choice also enabled us to achieve the successful deployment of ProB in WP2, where the use of classical B was mandatory.

Concerning the inclusion of SMT, SAT, BDD techniques into ProB, no decision has been taken yet. We are still investigating the possibilities.

Concerning B-Motion Studio, we decided not to use Flash (we had an earlier prototype using Flash). We wanted a tool that can be easily used and installed into Rodin. We also wanted a tool that can be used without having to learn a new programming language (Action Script).

4.4. Available Documentation

4.4.1. User Manual

ProB User Manual [2]

4.4.2. Published Papers

Below is a list of published papers, along with an abstract.

4.4.2.1. Improved Kernel to deal with large sets and relations

In this part we describe the successful application of the ProB validation tool on an industrial case study. The case study centres on the San Juan metro system installed by Siemens. The control software was developed and formally proven with B. However, the development contains certain assumptions about the actual rail network topology which have to be validated separately in order to ensure safe operation. For this task, Siemens has developed custom proof rules for AtelierB. AtelierB, however, was unable to deal with about 80 properties of the deployment (running out of memory). These properties thus had to be validated by hand at great expense (and they need to be revalidated whenever the rail network infrastructure changes). In this paper we show how we were able to use ProB to validate all of the about 300 properties of the San Juan deployment, detecting exactly the same faults automatically in around 17 minutes that were manually uncovered in about one man-month. This achievement required the extension of the ProB kernel for large sets as well as an improved constraint propagation phase. We also outline some of the effort and features that were required in moving from a tool capable of dealing with medium-sized examples towards a tool able to deal with actual industrial specifications. Notably, a new parser and type checker had to be developed. We also touch upon the issue of validating ProB, so that it can be integrated into the SIL4 development chain at Siemens

Michael Leuschel, Jérôme Falampin, Fabian Fritz, Daniel Plagge. Automated Property Verification for Large Scale B Models, FM'2009. [3]

4.4.2.2. Multi-Level Animation and Validation

We provide a detailed description of refinement in Event-B, both as a contribution in itself and as a foundation for the approach to simultaneous animation of multiple levels of refinement that we propose. We present an algorithm for simultaneous multi-level animation of refinement, and show how it can be used to detect a variety of errors that occur frequently when using refinement. The algorithm has been implemented in ProB and we applied it to several case studies, showing that multi-level animation is tractable also on larger models.

Stefan Hallerstede, Michael Leuschel, Daniel Plagge. Refinement-Animation for Event-B --- Towards a Method of Validation. ABZ'2010 [4]

See also Stefan Hallerstede, Michael Leuschel. How to Explain Mistakes. TFM'09. [5]

4.4.2.3. Test Case Generation

Choreography models describe the communication protocols between services. Testing of service choreographies is an important task for the quality assurance of service-based systems as used e.g. in the context of service-oriented architectures (SOA). The formal modelling of service choreographies enables a model-based integration testing (MBIT) approach. We present MBIT methods for our service choreography modeling approach called Message Choreography Models (MCM). For the model-based testing of service choreographies, MCMs are translated into Event-B models and used as input for our test generator which uses the model checker ProB.

Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, Ina Schieferdecker. Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. TESTCOM/FATES 2009. [6]

4.4.2.4. Proof-Directed Model Checking

With the aid of the ProB plug-in, the Rodin Platform provides an integrated environment for editing, proving, animating and model checking Event-B models. This is of considerable benefit to the modeller, as it allows him to switch between the various tools to validate, debug and improve his or her models. The crucial idea of this paper is that the integrated platform also provides benefits to the tool developer, i.e. it allows easy access to information from other tools. Indeed, there has been considerable interest in combining model checking, proving and testing. In previous work we have already shown how a model checker can be used to complement the Event-B proving environment, by acting as a disprover. In this paper we show how the prover can help to improve the efficiency of the animator and model checker.

Jens Bendisposto, Michael Leuschel. Proof Assisted Model Checking for B. ICFEM 2009. [1]

4.4.2.5. Debugging Tricky Proof Obligations with the ProB Disprover

While a large number of proof obligations can be discharged automatically by tools such as the Rodin platform, a considerable number still have to be proven interactively. In this paper, we describe a disprover plug-in for Rodin that utilizes ProB to automatically find counterexamples for a given problematic proof obligation. In case the disprover finds a counterexample, the user can directly investigate the source of the problem, as pinpointed by the counterexample. We also discuss under which circumstances our plug-in can be used as a prover, i.e. when the absence of a counterexample actually is a proof of the proof obligation.

Olivier Ligot, Jens Bendisposto, Michael Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. AFADL 2007. [7]

4.4.2.6. Inspection of Alternate Approaches

ProB is a model checker for high-level B and Event-B models based on constraint-solving. In this paper we investigate alternate approaches for validating high-level B models using techniques and tools based on using BDDs, SAT-solving and SMT-solving. In particular, we examine whether ProB can be complemented or even supplanted by using one of the tools BDDBDDB, Kodkod or SAL.

Daniel Plagge, Michael Leuschel, Ilya Lopatkin, Alexander Romanovsk. SAL, Kodkod, and BDDs for Validation of B Models. Lessons and Outlook. AFM'09. [8]

4.4.2.7. Validation of ProB

Symmetry reduction is a model checking technique that can help to alleviate the problem of state space explosion, by preventing redundant state space exploration. In previous work, we have developed three effective approaches to symmetry reduction for B that have been implemented into the ProB model checker, and we have proved the soundness of our state symmetries. However, it is also important to show that our techniques are sound with respect to standard model checking, at the algorithmic level. In this paper, we present a retrospective B development that addresses this issue through a series of B refinements. This work also demonstrates the valuable insights into a system that can be gained through formal modelling.

Edd Turner, Michael Butler, Michael Leuschel. A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking. ABZ'2010. [9]

4.4.2.8. B-Motion Studio

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

Lukas Ladenberger, Jens Bendisposto, Michael Leuschel. Visualising Event-B models with B-Motion Studio. FMICS'2009. [10]

4.5. Planning

In future, it is planned to work on the following topics:

4.5.3. Model-based Testing

- Directed model checking to achieve coverage (DEPLOY extension; flow graphs).
- Integrate algorithm into Rodin.
- Make algorithm more generic.
- Top-down multi-level animation.
- Move from prototype to real product.

4.5.4. B-Motion Studio

- Experiment with existing Flash animation and B model of ClearSy.
- Improve usability, more widgets.

4.5.5. Validation of ProB

- Test coverage analysis for Prolog code.
- Validation document to be delivered to Siemens.

4.5.6. Scalability

- More experiments with SAT, SMT, BDD techniques.
- Integration of Kodkod into ProB to solve complicated predicates over first order relations and simple sets.
- Adaption of ProB for the upcoming mathematical extensions. Indeed, for the moment the Rodin user is often required to model basic datatypes (records, sequences,...) or operators (transitive closure) herself. This is a big challenge to the animator, which does not know that the constants and variables of the machine (e.g. injective functions) are "simply" meant to model quite basic datatypes. With the introduction of mathematical extensions for records, transitive closure, ... this hurdle will be overcome.

4.5.7. Usability

- Feedback errors found by ProB into the PO view (as red icons).
- Improve disprover, detect when it is a decision procedure.
- Allow the user to easier inspect elements of the animated model: the cause of errors, why events are not enabled, etc.
- Further improvements to the GUI: 2-D Viewer, better multi-level animation view.

References

[1] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=253
[2] http://asap0.cs.uni-duesseldorf.de/trac/prob/wiki/
[3] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=248
[4] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=256
[5] http://www.springerlink.com/content/282p2316x7165588/
[6] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=252
[7] http://www.stups.uni-duesseldorf.de/publications_detail.php?id=219
[8] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=249
[9] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=257
[10] http://www.stups.uni-duesseldorf.de/~leuschel/publication_detail.php?id=258

5 Text Editor Plug-In

5.1. Overview

This part of the deliverable describes the Camille text editor plug-in.

5.2. Motivations

A number of frameworks for text editors are available, but EMF (Eclipse Modeling Framework Project [1]) was quickly identified as the target candidate, in combination with TEF (Textual Editing Framework), which is based on EMF. The framework seamlessly integrates into Eclipse. It is already proven within Rodin, as it is the foundation of UML-B. It is extensible - contributors of other plug-ins will be able to extend the text editor as well. And last, a proof-of-concept prototype had been put together very quickly.

Rodin has done away with a textual representation of the formal models. Indeed, events, theorems, axioms, etc. are stored in the Rodin database, and there is no classical text file to edit the models. The models are directly manipulated by a structural form-based editor. During the lifetime of the Rodin and DEPLOY projects it became increasingly apparent that the current structural editor was not able to cope with some industrial needs. Functions that are taken for granted in an editor - unstructured copy & paste or printing, just to name a few - were missing or only partially functional.

As an Event-B model consists of text, users were requesting a text editor, which would allow them to apply their already present text editing skills. As text editing is a well-understood problem, a variety of frameworks were available to implement one. These frameworks typically provide a wide range of standard features "for free", and typically provide extension points for extendability with additional features. The development team selected the Eclipse Modeling Framework (EMF), for the reasons outlined below.

The wide adoption of the text editor confirms that good decisions have been made. Besides standard text editing features as cut & paste, unlimited undo and redo, line numbering and many more, many Rodin-specific features had been implemented. The text editor supports syntax and semantic highlighting, code completion, templates, an outline view, quick navigation and many more.

5.3. Choices / Decisions

One important design consideration was to be able to re-use the Rodin formula syntax and parser. This was deemed vital for keeping up with future evolutions of the platform, e.g. the mathematical extensions will introduce new operators. As such, the grammar had to be carefully designed to be able to parse the structure of an Event-B model independently of the content of the predicates, expressions and actions. In other words, the structural parser detects the structure of the model and sends the formula content to the Rodin parser.

There is a very important limitation of the editor; while it can cope with changes of the mathematical language, it is not able to automatically deal with changes to the model’s structure. If we need to add information to the structure of a model itself, such as information on decomposition or flows it is necessary to modify the parser’s grammar and recompile it. There is to our best knowledge only one tool for the Eclipse framework that allows modification of the grammar during runtime. In theory, the features of TEF [2] perfectly fit our needs and indeed Alexei Iliasov developed in 2008 a prototype based on TEF. However there were a number of things that convinced us to not use it. First of all, the framework is released under the GNU Public License which is legally not compatible with the Eclipse Public License used in Rodin. Second, the parser RunCC used in TEF has some issues related to handling parse errors. Because the project did not make progress for three years it is very unlikely that these problems will be fixed by the creators of the parser. Extending RunCC could solve this problem but it does not seem to be feasible without major effort and resources. The grammar used to generate Camille's parser is close to EBNF, so we think that it is not too difficult for plug-in developers to contribute to the grammar but one has to be careful to not restrict the current language, i.e. we can only add optional syntax elements to the grammar, otherwise we break older models. However, this does not mean that we cannot have new mandatory elements, it only means that it has to be checked by the underlying model instead of the parser.

We also considered the BE4 Framework [3] as well as Eclipse XText [4]; an early version of Camille was indeed based on BE4 and it still contains some of its basic concepts. The main reason to not use BE4 was that editors for EMF models are less strongly based on files than editors for other languages. The EMF framework has its own notification system for changes in models. These change notifications may be caused by file changes. In addition, changes to the model coming from other plug-ins may trigger notifications too. This means that a text editor needs to react, i.e. to run its compiling respectively updating processes according to these changes. Therefore a build process which is based on files as BE4 offers is rather unsuitable. XText was also considered and is a reasonable alternative to our approach. At the time we started developing Camille, the XText framework was not released and it was not clear if and when certain features are included. Also the API was not stable at that time. In contrast to our solution, XText has the advantage that it creates the parser from the EMF model.

The most challenging technical part was the synchronisation with the Rodin database, particularly in the presence of other tools concurrently manipulating the same model. To achieve this, we created a new abstraction of the Rodin database as an EMF (Eclipse Modelling Framework Project [1]) data model. It allows us to work with Event-B models independently of the persistence strategy. In addition we can use EMF standard technologies for manipulating, comparing and merging of Event-B models.

The fit between the database model and the text representation posed a number of challenges. Users must be able to edit both in the text editor and the structural editor. However, the text editor provides much more freedom in formatting than the structural editor. The current implementation keeps the formatting of the user intact, unless changes through the structural editor are made. Also, comments cannot be placed anywhere, which is unintuitive in a text environment. This is due to the fact that the database allows comments only in certain places. Some constraints on permitted characters in labels and identifiers had to be made. In all these instances, meaningful error messages guide the user, and the editor attempts to be as unobtrusive as possible.

5.4. Available Documentation

5.4.1. Online Documentation

- Text Editor Wiki Page [5]
- EBNF Syntax for the Textual Representation [6]

5.4.2. Papers

- A Semantics-Aware Text Editor for Event-B. Fabian Fritz. Master's Thesis. 2009 [7]
- Developing Camille, a Text Editor for Rodin. Jens Bendisposto, Fabian Fritz and Michael Leuschel. 2009. (to appear in WS-TBFM 2010)

5.5. Planning

The Text Editor needs to be extended to support new attributes (e.g. stemming from the upcoming decomposition plug-in) and the upcoming mathematical extensions.

Users have expressed the desire to be able to insert comments everywhere. It is unclear whether this can be achieved without a major refactoring of the Rodin Database.

References

[1] http://www.eclipse.org/modeling/emf/
[2] http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html
[3] http://www.stups.uni-duesseldorf.de/thesis_detail.php?id=12
[4] http://www.eclipse.org/Xtext/
[5] http://wiki.event-b.org/index.php/Text_Editor
[6] http://wiki.event-b.org/index.php/TextEditor_EBNF
[7] http://www.stups.uni-duesseldorf.de/thesis_detail.php?id=20

6 Decomposition Support

6.1. Overview

The Event-B model decomposition is a new feature in the Rodin platform.

Two methods have been identified in the DEPLOY project for model decomposition: the *shared variable* decomposition (or A-style decomposition), and the *shared event* decomposition (or B-style decomposition). They both answer to the same requirement, namely the possibility to decompose a model M into several independent sub-models $M_1, ..., M_n$.

Academic (ETH Zurich, University of Southampton) and industrial (Systerel) partners were involved in the specification and development of model decomposition. Systerel, which could have useful discussions with Jean-Raymond Abrial on the topic, was more especially responsible for the A-style decomposition. The University of Southampton, where Michael Butler is professor, was in charge of the B-style decomposition.

6.2. Motivations

One of the most important feature of the Event-B approach is the possibility to introduce additional details such as new events and data during refinement steps.

Therefore, the refinement process entails an increasing complexity of a model, where one has to deal with a growing number of events, state variables, and consequently proof obligations. This is well illustrated in the *Event build-up* slide of the Wright presentation during the Rodin Workshop 2009.

See Experiences with a Quite Big Event-B Model [1].

The purpose of the Event-B model decomposition is precisely to give a way to address such a difficulty, by cutting a large model M into smaller sub-models $M_1, ..., M_n$. The sub-models can then be refined separately and more comfortably than the whole. The constraint that shall be satisfied by the decomposition is that these refined models might be recomposed into a whole model MR in a way that guarantees that MR refines M .

The model decomposition leads to some interesting benefits:

- Design/architectural decision. It applies in particular when it is noticed that it is not necessary to consider the whole model for a given refinement step, because only a few events and variables are involved instead.
- Complexity management. In other words, it alleviates the complexity by splitting the proof obligations over the sub-models.
- Team development. More precisely, it gives a way for several developers to share the parts of a decomposed model, and to work independently and possibly in parallel on them.

Note that the possibility of team development is among the current priorities for all industrial partners. The model decomposition is a first answer to this issue.

6.3. Choices / Decisions

The main decision concerning the implementation of the Event-B model decomposition in the Rodin platform is to make available both decomposition styles (*shared variables* and *shared events*) through one single plug-in. These approaches are indeed complementary and the end-user may take advantage of the former or of the latter, depending on the model, e.g., the *shared variables* approach seems more suitable when modelling parallel system and the *shared events* approach seems more suitable when modelling message-passing distributed systems.

Choices, either related to the plug-in core or to the plug-in graphical user interface, have been made with the following constraints in mind:

- Planning. Some options, such as using the Graphical Modelling Framework for the decomposition visualization, or outsourcing the context decomposition, have not been explored (at least in the first instance), mainly because of time constraints (in the DEPLOY description of work, the decomposition support is planned for end of 2009).
- Easy-to-use (however not simplistic) tool. It applies on the one hand to the tool implementation (decomposition wizard, configuration file to replay the decomposition) and on the other hand to the tool documentation (the purpose of the user's guide is to provide useful information for beginners and for more advanced users, in particular through a *Tips and Tricks* section).
- Modularity and consistency. In particular, the developments have not been performed in the Event-B core. Instead the Eclipse extension mechanisms have been used to keep the plug-in independent (e.g., the static checker, the proof obligation generator and the editor have been extended).
- Performance. The decomposition tool should perform in reasonable time and memory, compared to other Rodin plug-ins.
- Recursivity. It must be possible to decompose a previously decomposed model.

Other technical decisions are justified in the specification wiki pages.

6.4. Available Documentation

The following wiki pages have been respectively written for developers and end-users to document the Event-B model decomposition:

- Event model decomposition specification.

    See Event Model Decomposition [2].

- Decomposition plug-in user's guide.

    See Decomposition Plug-in User Guide [3].

6.5. Planning

The decomposition plug-in has been available since release 1.2 of the platform (initial version).

A further version allowing to edit an existing decomposition configuration is planned with release 1.3 of the platform.

See Rodin Platform 1.2 Release Notes [4] and Decomposition Release History [5].

References

[1] http://wiki.event-b.org/index.php/Image:Steve_Wright_Quite_Big_Model_Presentation.pdf
[2] http://wiki.event-b.org/index.php/Event_Model_Decomposition
[3] http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide
[4] http://wiki.event-b.org/index.php/Rodin_Platform_1.2_Release_Notes
[5] http://wiki.event-b.org/index.php/Decomposition_Release_History

7 Initial Definition of Language Support for Code Generation

7.1. Overview and Motivation

Code generation is an important part of the formal engineering tool chain that will enable complete support for development from high-level models down to executable implementations. Work has commenced on the development of support for code generation from Event-B models. This is a new line of work for DEPLOY that was not identified in the original Description of Work for the project. During the first year of the project, as the Deployment Partners gained experience with deployment of formal modelling, it became clear that having support for generation of code from refined Event-B models would be an important factor in ensuring eventual deployment of the DEPLOY approach within their organisations. This is especially true for Bosch and Space Systems Finland (SSF). During the DEPLOY re-focus at Month 18, it was decided to introduce a code generation task into a revised workplan and devote resources to this task.

After receiving more detailed requirements from Bosch and SSF, it became clear we should focus our efforts on supporting the generation of code for typical real-time embedded control software. In essence, this involves programs structured as tasks running concurrently such as supported by the Ada tasking model. The individual tasks are sequential programs and tasks share state variables via some form of monitor mechanism. For real-time control, both periodic and aperiodic task should be supported; tasks should have priorities to ensure appropriate responsiveness of the control software.

For the DEPLOY pilots, it is regarded as sufficient to support construction of programs with a fixed number of tasks and a fixed number of shared variables – no dynamic creation of processes or objects is required. It is not our intention to develop a fully-fledged industrial-strength code generation framework within the lifetime of the DEPLOY project. Instead an aim is to develop a sufficient framework to act as a proof-of-concept to enable code generation for the Bosch and SSF pilots. A further aim is to gain practical experience with a prototype code generation framework that will serve as a basis for future R&D on a scalable code generation framework. The code generation work is being lead by Southampton with initial input from Newcastle.

7.2. Choices / Decisions

Three candidate approaches have been posited as follows:

- Enrich Event-B with explicit algorithmic structures for use in later refinement stages and use these explicit structures to guide code generation.
- Synthesise sequential and concurrent code from existing low-level Event-B models.
- Exploit the code generation facilities of Atelier-B for “classical” B.

Approach 3 will be important for STS in WP2 as it will enable integration of Rodin with the existing development process in STS with AtelierB and some effort will be devoted to this. However, this approach will not address the needs of all the deployments because of the overhead involved in having to use AtelierB alongside Rodin as well as the lack of support for concurrent code generation in AtelierB. Supporting Approach 2 has the attraction that developers can remain within the uniform language framework of Event-B. However, it is not clear that we can meet the real-time performance guarantees required for embedded systems with this approach within the lifetime of the DEPLOY project.

While we will pursue some exploration of this approach, we will focus most effort on Approach 1 as this will enable existing techniques for real-time programming to be incorporated in later stages of development. Language support for explicit sequential and concurrent programs will be defined to enable construction of code-oriented models at low levels of refinement. We refer to this form of Event-B as task-oriented. A code generation framework will be developed to enable automated generation of C and Ada subsets suitable for real-time embedded systems from code-oriented models. A refinement-based proof method for code-oriented models will be defined and incorporated into the Rodin toolset.

An important requirement is that the code generation framework be amenable to extension and tailoring for specific needs. Rather than hard-wiring the code generation rules into the tool, we will aim to support a declarative rule-based approach to defining translation from task-oriented Event-B to a target implementation language. The translation should be tailorable by modifying or extending the declarative translation rules. We will explore the use of a model-based transformation framework such as ATL or ETL for this purpose. Rather than fixing on a specific set of implementation-level data types, we will exploit the mathematical extension facility being developed for Rodin to enable a flexible approach to defining implementation-level data structures. For example, arrays could be supported by defining a new theory of arrays. This theory would define array declarations and operators along with an appropriate set of proof rules for arrays to enable reasoning. Appropriate declarative rules would then be defined to translate array declarations and expressions to appropriate code in the target language.

7.3. Tasking Language

The Event-B tasking language will support the following sequential and concurrent algorithmic constructs:

Behaviour	Algorithmic Construct
Sequence	;
Branching	if
Iteration	while
Task/Thread/Process	task
Shared variables	machine

A task-oriented model will be definable using the following abstract syntax

Task :=

task Name

variables Variables

invariants Invariants

begin TaskBody **end**

TaskBody :=

Event

| TaskBody ; TaskBody

| **if** Guard→ TaskBody [] … [] Guard→ TaskBody **fi**

| **do** Guard→ TaskBody [] … [] Guard→ TaskBody **od**

The sequence, branching and iteration constructs correspond to their imperative counterparts. The task construct provides a means to specify the actions of interleaving, concurrent executions. A task may be implemented by an Ada task, or a thread in C. A task, in isolation, is a sequential program with clearly identified atomic steps, and each step corresponding to an atomic event.

The standard machine construct provides the mechanism for sharing data between the executing tasks. The interaction between a task and a machine can be represented synchronized event composition as currently supported by the synchronised-event plug-in. The generated implementation will need to ensure that a task has mutually exclusive access to the variables represented in the machine. This is provided by a machine's atomic events. A machine representing shared variables may be implemented by an Ada protected object, or in C by an explicit Mutex variable and appropriate lock acquisition.

To facilitate real-time programming constructs, we introduce the notion of task type, task period, and task priority.

Task :=

task Name

tasktype **periodic**(p) | **triggered** | **repeating** | **oneshot**
priority n

…

A tasktype is used to indicate the scheduling requirements for the required activity. It is typically the case in real-time systems that the required activity involves some repeating behaviour, this can be a continuously repeating loop, or one that repeats at a predetermined time interval. We introduce repeating and periodic tasktypes. The periodic task is parameterised by a time value. Activity can also be initiated by externally generated interrupts, which may arise from some user action or from the hardware itself. To accommodate this we introduce the triggered tasktype. The last type of activity that we consider is the one-shot task; where some activity is performed and not repeated. To facilitate this we add the oneshot tasktype. In some cases it may be useful to delay a task for a period of time; we introduce a delay to the TaskBody to facilitate this. It may be decided that, during a program's execution, some of the activities should take precedence over some others. We introduce priority which allows a developer to assign a numerical value indicating the precedence for scheduling. We adopt the convention that higher priority activities have a greater value, and this corresponds to the Ada convention.

7.4. Methodological Support

As previously outlined, the atomic steps of a sequential task will be syntactically explicit. This is in order to facilitate a direct mapping of task steps with events of an abstract machine. The development approach that we plan to support fits with the general refinement approach of Event-B in that explicit tasking will be introduced as part of a refinement step. A common approach to specifying a problem is to represent some desired outcome as a single abstract atomic event. Common patterns of refinement of such abstract events found in existing Event-B developments are as follows:

1. Choice refinement: Event E is refined to E1 [] E2 []...[] En, where each Ei refines E
2. Sequential refinement: Event E is refined to E1 ; E2 ; … En, where some Ei refines E and the other Ej refine skip.
3. Loop refinement: Event E is refined to E1 ; do E2 od ; E3, where E3 refines E and E1 and E2 refine skip.

These patterns motivate a simpler subset of the tasking language as follows

TaskBody :=

Event

| TaskBody ; TaskBody

| **if** Event [] … [] Event **fi**

| **do** Event **endwith** Event **od**

Initially we will support refinement to this subset by providing special structure-introduction refinement rules in the manner of the refinement calculus. We will also develop support for a generalisation of this set of rules in which the abstract model consists of a group of atomic events, with each event representing a different possible outcome. An example use of such a group would be to have an event to represent the normal behaviour of some system feature and a separate event to represent the error behaviour of the feature.

The above rules prevent the immediate introduction of nested tasking structures such as nested loops. Following the approach found in the refinement calculus and common practice in Event-B refinements, such nested structures may be introduced through further refinement steps. For example, to introduce a nested loop, the outer loop is first introduced and the inner loop is introduced in a subsequent refinement step. We will explore generalising the refinement support for tasks to support nesting through refinement along these lines.

7.5. Timescales

We refer to the tasking language outlined above as Version V1. Our planned timescales for further work on code generation are as follows:

- June 2010: demonstrator tool for language V1.
- June 2010: initial support for user defined datatypes (mathematical extensions).
- From June to October 2010: experimentation with and assessment of the demonstrator tool on the WP1 and WP3 pilots leading to feedback on V1 and the tool.
- Jan 2011: algorithmic language definition V2.
- June 2011: prototype tool for V2.
- From June to October 2011: experimentation with and assessment of the V2 tool on the WP1 and WP3 pilots.

8 Improvements to Existing Provers

8.1. Overview

Proving is at the core of the Rodin methodology. Therefore, it was no surprise that both industrial and academic users reported a lot of feedback on this topic. This feedback was provided either by mail or though the SourceForge trackers.

Based on this feedback, several actions were taken to improve the existing tooling for both automated and interactive proving. Some consisted in improving existing tools, while others needed a full design and development of new tools (especially for visualizing and managing proofs). Finally, an extension of the proving framework API as been realised to allow for the development of new plug-ins (such as the rule based prover).

Systerel has been in charge of existing prover improvements, with support from ETH Zurich. The evolution of the API has been designed in close collaboration with University of Southampton.

8.2. Motivations

The motivations for improvements to existing provers can be summarized as follows:

- Reducing proving time and effort.

  New proof rules, both manual and automatic (e.g.., One point rule [1], arithmetic rules), have been added to discharge more proof obligations more easily.

- Reflecting corrections in prover implementations.

  The reasoners [2] are versioned. A reasoner implementation may indeed evolve in time (bug fixes, modifications of the behaviour, etc), even after the old implementation has been used to prove a model. This may lead to potential issues when trying to reuse or replay proofs serialized by the old reasoner implementation. Such problems are solved through the reasoner versioning mechanism.

- Reducing proof storage space.

  Big proof files are difficult to handle, on the one hand by the Rodin platform (slow access), and on the other hand by users (project sharing).

  The proof purging [3] and proof simplifying [4] mechanisms have been implemented to address this issue.

- Facilitating manual proof review or reuse.

  The proof skeleton view [5] allows to quickly browse through a proof. : Moreover, the provided copy/paste feature makes it possible to reuse a stored proof into a new proof.

- Improving prover API.

  The tactic provider API [6] has been made more flexible to facilitate tactic contributions.

8.3. Choices / Decisions

The proving API one year ago asked that available proof commands would be completely determined statically (at application startup). This decision had to be revised to allow for dynamically contributed proof commands. This change was required not only for regular rules where several options could be considered based on available hypotheses, but also for plugging in the rule based prover, developed by University of Southampton. This API extension was fully designed and discussed through the Rodin wiki.

See New Tactic Providers [6].

Following to the detection of incorrect proof rules implemented in the tool, a complete review of all proof rules and their implementation has been carried out. Moreover, a review procedure has been defined to lower the risk that such glitches happen again in the future. Also, the decision to develop a rule based prover (where rules must be formally proved before being used) will provide greater confidence in the correctness of the prover.

In the current setting, proof files can grow very large (in the order of tens of megabytes). This is partly caused by the usage of the Rodin database mechanism for storing proofs in XML files. At the last DEPLOY workshop (October 2009), several other options have been discussed to reduce this memory footprint. This issue is beging further investigated.

8.4. Available Documentation

The following pages give useful information about prover improvements:

- Prover Rules

  See Inference Rules [7].

  See All Rewrite Rules [8].
- Proof Skeleton View

  See Proof Skeleton View [5].
- Proof Purger

  See Proof Purger Interface [3].
- Prover API evolution

  See New Tactic Providers [6].
- Versioned Reasoners

  See Versioned Reasoners [2].

8.5. Planning

The above mentioned improvements were made available since release 1.1 of the platform:

Rodin Platform 1.1 Release Notes [9]

In the third year of DEPLOY, most effort in the proving area will be put into:

- Better management of well-definedness conditions.

- Improvement to the rule based prover.

- Bridging the gap with external SMT solvers.

- Supporting mathematical extensions in proofs.

References

[1] http://www.cs.cmu.edu/afs/cs/academic/class/15671-f95/www/handouts/proof/node1.html
[2] http://wiki.event-b.org/index.php/Versioned_Reasoners
[3] http://wiki.event-b.org/index.php/Proof_Purger_Interface
[4] http://wiki.event-b.org/index.php/Proof_Simplification
[5] http://wiki.event-b.org/index.php/Proof_Skeleton_View
[6] http://wiki.event-b.org/index.php/New_Tactic_Providers
[7] http://wiki.event-b.org/index.php/Inference_Rules
[8] http://wiki.event-b.org/index.php/All_Rewrite_Rules
[9] http://wiki.event-b.org/index.php/Rodin_Platform_1.1_Release_Notes

9 Rule-based Prover

9.1. Overview

The rule-based prover plug-in offers a uniform mechanism to define and validate proof rules which can then be used in proofs.

The rule-based prover plug-in has two important components:

- Theory construct, where rules are defined and validated by means of proof obligations. Defining a rule includes stating whether it should be applied automatically, interactively or both.
- Prover extension, which is responsible for checking what rules are applicable and applying them.

The plug-in supports the definition and validation of rewrite rules. It is expected that the plug-in will also support defining inference rules.

The University of Southampton was responsible for the development of the rule-based prover.

9.2. Motivations

Extensibility is a major concern for theorem provers. The Rodin proving infrastructure offers an extensible mechanism where proof rules can be added and external provers can be plugged-in. However, it has the following limitations:

- In order to add a new proof rule, it is required to implement a rule schema (i.e. a reasoner) and a wrapper tactic. Therefore, a certain level of competence with the Java programming language as well as knowledge of Rodin architecture is necessary.
- After a new rule is added, soundness of the prover augmented with the new rule has to be established. It is not clear how this can be achieved at the level of Java code.

The rule-based prover is an attempt to address the aforementioned limitations in a uniform and effective fashion. It is uniform because it offers the user (we shall call a theory developer) the possibility to develop and validate theories in a similar way to developing and validating models. It is also effective since it relieves the theory developer from writing Java code, and covers most of the rewrite rules available in [8]

The advantages of the rule-based prover include:

- The rule-based prover unifies the way automatic and interactive rules are defined since this is literally specified by two toggle buttons.
- The addition and validation of new proof rules brings a degree of meta-reasoning to Rodin, and removes the need for Java code when adding rules.
- The theory construct provide a platform where prover extensions and (in the future) language extensions can be specified.
- Carefully checked library of rules can be provided.

9.3. Choices/Decisions

The main decisions that had to be made regarding the rule-based prover include the following:

- Whether to use contexts as a vehicle to define rules.
- What kind of rules should the theory cover first.
- How the meta-variables within rules are recognised (automatic type inference or not).

The following key points summarise the different decisions/choices that have been made:

- Contexts describe the static properties of models, and they are used to parameterise machines. Adding the capability of defining proof rules within contexts would allow the co-existence of modelling elements and meta-logical elements with no clear relationship between the two. This may require significant changes to the core architecture, and will unnecessarily overload the functionality of contexts with elements not directly relevant to modelling. As such, a clear separation of modelling and meta-reasoning was adopted. This resulted in a theory construct that acts as a placeholder for prover extensions.
- Rodin has a collection of rewrite and inference rules. Most inference rules as found in [7] require predicate variables to be defined. This is not the case for most rewrite rules as found in [8].

  Since predicate variables were not available when the development started, it was decided to cover rewrite rules first.
- Rules are defined using metavariables each of which must have a type. To facilitate static and type checking, metavariables must be defined in a theory before they can be used. The definition includes an identifier name and a type.
- Deciding whether a rule should be applied automatically is not straightforward. Therefore, this is left to the theory developer. Each new rule can be tagged automatic (can be applied automatically), interactive (available in interactive proofs) or both.

9.4. Available Documentation

There is a dedicated wiki page covering the plug-in functionality:

- Rule-based Prover [1]

9.4.1. Papers & Technical Reports

- Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. On an Extensible Rule-based Prover for Event-B, ABZ'2010. [2]
- Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. On an Extensible Rule-based Prover for Event-B, Technical Report. [3]

9.5. Planning

The rule-based prover plug-in is available as an external plug-in for Rodin release 1.1 and above.

See Rodin Platform 1.1 Release_Notes [9] and Rodin Platform 1.2 Release_Notes [4].

References

[1] http://wiki.event-b.org/index.php/Rule-based_Prover_Plug-in
[2] http://eprints.ecs.soton.ac.uk/18269/
[3] http://eprints.ecs.soton.ac.uk/18273/

10 Pattern Plug-in

10.1. Overview

The pattern plug-in enables an Event-B developer to make advantage of former models and their refinements.

A refinement can also be seen as the solution to the problem encountered in the abstraction. One can make use of such a solution if the solved problem appears in the current development. Instead of solving the problem again we directly use the already known solution. Certainly, we have to show that our current problem (or at least part of it) is the same as the solved problem.

We refer to such a reusable model containing a certain solution to a problem as a pattern. Since these patterns are just regular models every model can be a pattern in principle. There is only a limit in terms of usability that correlates with the specificity of the model (solved problem).

As mentioned above, the problem at hand (or at least part of it) has to be similar to the pattern we want to use. To ensure this similarity the developer has to match the pattern with the problem at hand. After a successful matching of the models (problems) the refinement (solution) of the pattern can be incorporated into the problem at hand. This leads to a refinement of this model that is correct by construction. In other words, a new refinement of the problem at hand can be generated which includes the achievements of the pattern and is correct without proving any proof obligation.

10.2. Motivations

The idea of patterns and their usage is described in every detail in the Master Thesis "Design Patterns in Event-B and Their Tool Support“ [1]. This approach follows the earlier proposal of Jean-Raymond Abrial and Thai Son Hoang stated in the paper "Using Design Patterns in Formal Methods: An Event-B Approach" [2].

There are two main motivations to use patterns.

- Reusing solutions to problems:

  Usually there is more than one solution to a problem. But not every solution is of equal quality. There are solutions that are especially easy, elegant or short. For a lot of problems there is a best practice to solve it. Having a pattern consisting in this "best" solution one does not have to bother finding it ever again.

- Reusing proofs:

  As mentioned above, the pattern approach is able to generate a refinement that is correct by construction. This is possible because the construction of the refinement leads to the same proof obligations as in the pattern. Since they are proved in the pattern there is no need do the same proof steps in the current development again. Reusing proofs, especially manual discharged proof obligations, saves a lot of time for the developer. The drawback reflects in the effort to match the pattern with the problem before a refinement can be generated. But this effort can be minimised by a tool that does as many steps as possible automatically or at least supports the developer wherever user interaction is required.

Case studies showed also another motivation to have such a tool. Using the pattern approach without tool support, especially the generation of the refinement, is time consuming and error prone. In this last step in the whole procedure, basically two machines are merged by copying and pasting elements. This often leads to name clashes where a developer can easily loose track. Having a tool checking for possible name clashes in advance can avoid a lot of confusion.

10.3. Choices / Decisions

- To support the developer and guide him through the whole pattern process, we designed a plug-in providing a graphical wizard that consists in several pages, one for each major step.
- It was desired to have direct access to the pattern plug-in in form of an API in addition to the wizard. This enables other Rodin developers to use the pattern plug-in programmatically. For this, the first version of the plug-in was revised by having the generation apart from the pure graphical interface.
- In order to speed up the process, all required Event-B element of the involved machines are collected in a central data object. Certain elements are contained in more than one list dependent to their attributes (e.g. a matched and renamed event).
- The correctness of the matching, which is checking the syntactical equality of the matched elements (guards, actions) is left to the developer for the moment. The automation of this was postponed to a later version as it appears to us as a minor point.
- As the pattern plug-in is in a pre-release phase, there is an option to generate the proof obligation in order to control the generation.

10.4. Available Documentation

Besides the documents mentioned above focusing on the theory there also exists a wiki page that is more tool related.

> See Pattern [3] for a short overview of the idea of patterns in Event-B and stepwise instructions for both developers interested in using the wizard and those more thrilled by APIs.

10.5. Planning

The pattern tool is available as an external plug-in for Rodin release 1.1 and above.

> See Rodin Platform 1.1 Release Notes [9] and Rodin Platform 1.2 Release_Notes [4].

The current version of the pattern plug-in covers the following functionalities:

- Interactive guidance for matching the variables.
- Interactive guidance for matching the events and their parameters, guards and actions.
- Collecting the seen contexts in order to enable the user to match the carrier sets and constants.
- Checking for name clashes and proposing possible renaming.
- Detection of disappearing variables that have to be replaced.
- Detection of disappearing parameters that have to be replaced.
- Generation of the new machine file.

Desired functionalities that are missing in the current version:

- Automated syntactical check of the matched elements.
- Automated extraction of the glueing invariants to find the replacement for disappearing variables.
- Automated extraction of the witnesses to find the replacement for disappearing parameters.

The current version has been passed to interested partners for evaluation. The date for the missing functionalities being implemented in the plug-in will depend on the responses of the evaluators and their need of having those functionalities available.

References

[1] http://e-collection.ethbib.ethz.ch/view/eth:41612
[2] http://www.springerlink.com/content/d088h53531x7226j
[3] http://wiki.event-b.org/index.php/Pattern

11 Flow Plug-in

11.1. Overview

Event-B, being an event systems formalism, does not have a mechanism to explicitly define event ordering. Although event guards may express any desired event ordering, the ability to have a summary of possible event flows in a concise and compact form is useful for many tasks, for example, code generation and connecting with other formalisms. The flows plug-in addresses one aspect of event ordering: it allows a modeller to specify and prove that a given sequence of events does not contradict a given machine specification. More precisely, if we were to execute a machine step-by-step following our prescribed sequence of events we would not discover divergences and deadlocks not already present in the original machine. In other words, the constraint on event ordering must be such that the overall specification is an Event-B refinement of the original model. Importantly, this means that all the desired model properties proved before are preserved.

Sequential composition of events may be expressed in a number of ways:

- Event immediately follows another event; no other events may take place between the composed events.
- Event eventually follows an event; thus, although there is an interference from other events, it is guaranteed that the second is eventually enabled.
- Event may follow an event; this is the weakest form of connection when we only say that it may be the case that the second event follows the first event; it may happen, however, that some other event interferes and the second event is delayed or is even not enabled ever.

Although the last case may seem the least appealing, it is the one that forms the basis of the Flows plug-in. The primary reason to offer such a weak guarantee is proof effort required for stronger types of connectives.

11.2. Motivations

There are a number of reasons to consider an extension of Event-B with an event ordering mechanism:

- For some problems the information about event ordering is an essential part of requirements; it comes as a natural expectation to be able to adequately reproduce these in a model.
- Explicit control flow may help in proving properties related to event ordering.
- Sequential code generation requires some form of control flow information.
- Since event ordering could restrict the non-determinism in event selection, model checking is likely to be more efficient for a composition of a machine with event ordering information.
- A potential for a visual presentation based on control flow information.
- Bridging the gap between high-level workflow and architectural languages, and Event-B.

It is also hoped that the plug-in would improve readability of larger models: currently they are simply a long list of events with nothing except comments to provide any structuring clues.

11.3. Choices / Decisions

The primary functionality of the plug-in is the generation of additional proof obligations. Rodin model builder automatically invokes the static checker and the proof obligations generator of the plug-in and the proof obligations related to flow appear in the list of the model proof obligations.

One of the lessons learned with an initial plug-in prototype was that a CSP-like language notation is not the best way to express event ordering as not all users are familiar with process algebraic notations. It was decided to use graphical editor to allow a visual layout of flow diagrams. This, in our view, is a more intuitive way of specifying event ordering. To realise this, we have relied on GMF - an Eclipse library to manipulate EMF models using graphical editors.

11.4. Available Documentation

There is a wiki [1] page summarising proof obligation involved in proving machine/flow consistency.

11.5. Planning

The plug-in is available since the release 1.2 of the platform.

References

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

12 Modularisation Plug-in

12.1. Overview

The Modularisation plug-in realises a support to structure Event-B developments into modules. The objective is to achieve better structuring of models and proofs while also providing a facility for model reuse. It is expected that the structuring approach realised in the plug-in would complement the functionality A/B-style decomposition plug-in.

The module concept is very close to the notion of Event-B development (a refinement tree of Event-B machines). However, unlike a conventional development, a module is equipped with an interface. An interface defines the conditions on the way a module may be incorporated into another development (that is, another module). The plug-in follows an approach where an interface is characterised by a list of operations specifying the services provided by the module. An integration of a module into a main development is accomplished by referring operations from Event-B machine actions using an intuitive procedure call notation.

The plug-in was developed in Newcastle University in cooperation with Abo Academy and Space Systems Finland.

12.2. Motivations

There are several conceptual approaches to decomposition. To contrast our proposal, let us consider some of them.

One approach to decomposition is to identify a general theory that, once formally formulated, would contribute to the main development. For instance, a model realising a stack-based interpreter could be simplified by considering the stack concept in isolation, constructing a general theory of stacks and then reusing the results in the main development. Thus, an imported theory of stack contributes axioms and theorems assisting in reasoning about stacks.

Decomposition may also be achieved by splitting a system into a number of parts and then proceeding with independent development of each part. At some point, the model parts are recomposed to construct an overall final model. This decomposition style relies on the monotonicity of refinement in Event-B although some further constraints must be satisfied to ensure the validity of a recomposed model. A-style and B-style decompositions fit into this class.

Finally, decomposition may be realised by hierarchical structuring where some part of an overall system functionality is encapsulated in a self-contained modelling unit embedded into another unit. The distinctive characteristic of this style is that recomposition of model parts happens at the same point where model is decomposed.

The Modularisation plug-in realises the latter approach. The procedure call concept is used to accomplish single point composition/decomposition. There are a number of reasons to try to split a development into modules. Some of them are:

- Structuring large specifications: it is difficult to read and edit a large model; there is also a limit to the size of a model that the platform may handle comfortably and thus decomposition is an absolute necessity for large scale developments.

- Decomposing proof effort: splitting helps to split verification effort. It also helps to reuse proofs: it is not unusual to return back in refinement chain and partially redo abstract models. Normally, this would invalidate most proofs in the dependent components. Model structuring helps to localise the effect of such changes.
- Team development: large models may only be developed by a (often distributed) developers team.
- Model reuse: modules may be exchanged and reused in different projects. The notion of interface make it easier to integrate a module in a new context.
- Connection to library components.
- Code generation/legacy code.

12.3. Choices / Decisions

The primary objective in the tool design was to provide a simple to use tool that could be used by a non-expert modeller. Of course, close integration with the core platform functionality was paramount.

- We have decided that there is a need for a new type of Event-B component: interface. A decomposition based on explicit interface (rather than on an implicit one, such as in A-style decomposition) facilitates the reuse of modules and makes it easier to provide a rich management infrastructure.
- We have had to decide whether to make module integration more explicit and flexible or hide details under syntactic sugar and thus achieve better model readability. We have decided that model readability should take priority over everything else. However, while model representation becomes more compact, it does not make proofs easier.
- During the initial experiments we have identified a need for multiple module instantiation. This allows a modeller to use several copies of the same module using a qualifier prefix to distinguish objects imported from the modules.
- One crucial point was to realise modularisation support in such a way that structuring may be recursively applied within modules. Indeed, a module implementation (module body) is a machine and thus it is self-similar to a caller context that is a machine.
- For the current version, we have not implemented the generation of enabledness condition logically required for module implementation. This condition, in some form, should be present in the platform core.

12.4. Available Documentation

There is a dedicated wiki page covering the plug-in functionality. Also, we are working on further documentation and tutorial.

- Plug-in wiki [1]
- Plug-in tutorial [2]
- Installation guide [3]

Two small-scale examples are available:

- [4] - A model of queue based on two ticket machine module instantiations (very basic).
- [5] - Two doors sluice controller specification that is decomposed into a number of independent developments (few first steps only).

12.5. Planning

The plug-in is available since the release 1.1 of the platform. See the Modularisation Plug-in Release Notes [6].

References

[1] http://wiki.event-b.org/index.php/Modularisation_Plug-in
[2] http://wiki.event-b.org/index.php/Modularisation_Plug-in_Tutorial
[3] http://wiki.event-b.org/index.php/Modularisation_Plug-in_Installation_Instructions
[4] http://iliasov.org/modplugin/ticketmachine.zip
[5] http://iliasov.org/modplugin/doors.zip
[6] http://wiki.event-b.org/index.php/Modularisation_Plug-in_Release_Notes