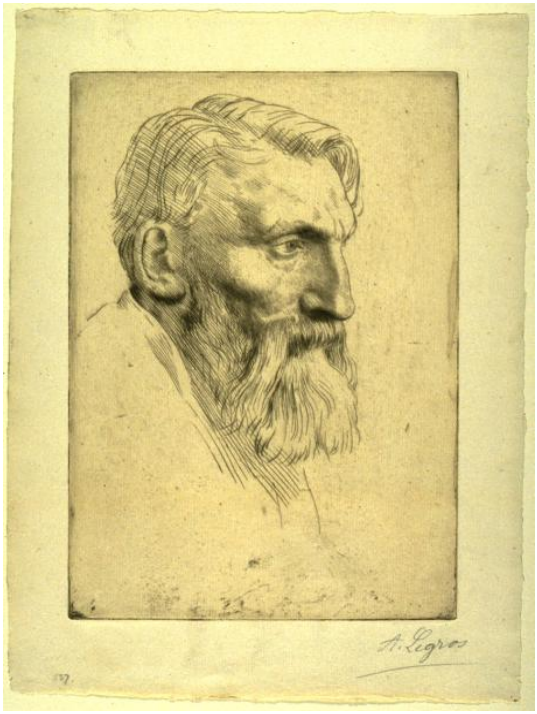
Project IST-511599
RODIN
"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D24

Internal Versions of Plug-in Tools

Editor: Michael Butler, University of Southampton

Public Document

28 February 2007

http://rodin.cs.ncl.ac.uk/

Contributors:

Colin Snook (University of Southampton)
Abdolbaghi Rezazadeh (University of Southampton)
Michael Leuschel (University of Düsseldorf)
Jens Bendisposto (University of Düsseldorf)
Olivier Ligot (University of Düsseldorf)
Apostolos Niaouris (University of Newcastle)
Thierry Lecomte (Clearsy)

Table of Contents

1 Introduction 4
2 U2B Plug-in 5
2.1 Overview of plug-in functionality 7
2.2 Overview of plug-in integration 14
2.3 Conclusions 17
2.4 References 17
3 B2Rodin Plug-in 18
3.5 Overview of plug-in functionality 18
3.6 Overview of plug-in integration 20
4 Brama Plug-in 22
4.1 Overview of plug-in functionality 22
4.2 Overview of plug-in integration 26
5 Mobility Plug-in 26
5.3 Overview of plug-in functionality 27
5.4 Overview of plug-in integration 29
6 ProB Plug-ins 31
6.1 RODIN ProB Visual Animation Plug-in 32
6.2 RODIN ProB Disprover Plug-in 32
Appendix A The ProB Plug-In for Eclipse and Rodin
Appendix B Debugging Event-B Models using the ProB Disprover Plug-in

1 Introduction

This deliverable consists of prototype versions of plug-in tools that are intended to be internal to the RODIN project. The deliverable consists of the software along with this overview paper report. In the previous plug-in deliverables from WP4 (D11 and D16) we described a mixture of requirements on tools and tools that existed in stand-alone form. Since D16 was produced, the prototype RODIN platform became available. This allowed us to concentrate our effort on producing plug-ins that integrated with the RODIN platform. We have focused our effort on integrating a small number of plug-ins on the platform over the last 12 months.

Several of the plug-ins can be downloaded as follows:

U2B	http://www.ecs.soton.ac.uk/~cfs/downloads/ac.soton.umlb-site/
B2RODIN	http://www.b4free.com/b2rodin
Brama	http://www.brama.fr/index_en.html
ProB	http://www.stups.uni-duesseldorf.de/ProB/update/prototype/

A prototype of the Mobility Checker plug-in is packaged with this deliverable. A plug-in is installed in the RODIN Eclipse platform as follows:

- Choose 'Help/Software Updates/Find and Install
- Select 'Search for new features to install
- Click on the 'New Remote Site...' button
- In the dialog box, fill the following fields :
    - Plug-in name in the 'Name' field
    - URL, e.g., 'http://www.b4free.com/b2rodin' in the 'URL' field
- then click on OK
- Click on finish to install the plug-in.

Prototypes of a model-based testing tool and a translator from Event-B to the Bluespec hardware description language are still under development. It is important to note that the main use of RODIN tools in a case study involves using the core consistency checking and proof provided by the platform. The plug-ins are used for the additional functionality they provide as appropriate. Each case study will use a mixture of the core functionality and the additional plug-in functionality.

The plug-ins provided as part of D24 are all integrated with the RODIN platform and are now being validated on the case studies of WP1. The planned use of the plug-ins in the case studies is summarised in the following table:

	CS1	CS2	CS3	CS4	CS5
UML-B	X	X	X	X	
B2RODIN		X		X	X
Brama		X			
Mobility Checker					X
ProB		X	X	X	X

CS1: Protocol engineering
CS2: Engine failure management
CS3: Mobile internet services
CS4: CDIS Air Traffic Control
CS5: Ambient campus

2 U2B Plug-in

The UML-B described in this report is a new graphical formal modelling notation that is based on UML and relies on Event-B and the RODIN Event-B verification tools. In previous work [SnBu06] we developed a specialisation of the UML called UML-B using the profiling extension mechanism included in UML. The profile and translator was capable of several alternative modelling styles, including an event style version of classical B. However, the degree of integration between the tools was poor and unidirectional. The new version of UML-B is implemented in Eclipse and is therefore platform independent and closely integrated with the RODIN Event-B tools. UML-B is now a plugin extension feature to the RODIN Event-B platform and U2B runs as an Eclipse builder so that Event-B is generated and analysed automatically as soon as the UML-B model is saved. Problems discovered by the verification tools will be fed back and displayed on the UML-B model diagrams (this feature is still under development).

Experience with the initial version of UML-B indicated that the richness and semantics of UML could be misleading for modellers. UML-B used a subset of UML features that were useful for translation into B. However, users were confused over which features they should use and often complained that a setting hadn't done anything. Another problem was that sometimes experienced UML users complained that the semantics of UML was not quite the same as that used by UML-B. For our initial attempt at the new RODIN UML-B we again used a profile. In UML 2.0, the concept of profiles had been strengthened and was supported in Eclipse by the UML2 project. We used properties attached to our profile stereotypes to define all the features we needed even where UML contains a similar concept. Only the bare diagram elements from UML were used. The method was tested using Rational Software Architect [RSA] which is based on Eclipse and supports profiles implemented in UML2. This method was an improvement on the previous UML-B. The specialisation was clearly demarked from the basic UML notation leading to less confusion. Stereotypes were automatically applied and profile features were entered in a separate view pane. However, there was a strong feeling that the profile was an add-on and not an integral part of the notation. There was still the problem that

the main notation contained a lot of unused redundant modelling concepts. Apart from the concern about usability and elegance of the modelling notation, there was a great deal of redundant tooling supporting the unused UML and a dependence on the Eclipse UML2 project providing appropriate facilities to support our aims. The profile extension mechanism is intended to be used when a relatively small adaptation of UML is required. When the specialisation is more extensive, as in our case, a new metamodel should be defined. The UML is defined by a metamodel that is described using MOF (a small subset of UML for describing modelling notations). The advantage of defining UML-B via an independent metamodel is that it can be designed to the requirements rather than as an adaptation of something more general. Hence UML-B is now a UML-like formal modelling language rather than a specialisation of the UML.

The new UML-B provides a top-level Package diagram for showing the structure of, and relationships between, components (machines and contexts) in a project. Contexts are described in a context diagram (similar to a class diagram but has only constant data) and Machines are specified in a Class diagram. Statemachines can be attached to classes. Statemachines can also be attached to states providing a hierarchical nesting of statemachines. For textual constraints and actions we use a notation, μB (micro B) that borrows from the Event-B notation. μB has the following differences from Event-B: An object-oriented style dot notation is used to show ownership of entities (attributes, operations) by classes.

To give a flavour of UML-B, consider the specification of the telephone book in Fig. 2.1. The classes, NAME and NUMB represent people and telephone numbers respectively. The association role, pbook, represents the link from each name to its corresponding telephone number. Multiplicities on this association ensure that each name has exactly one number and each number is associated with, at most, one name. The properties view shows μB conditions and actions for the add event. The add event of class NAME has the constructor property set (not shown in Fig. 2.1) which means that it adds a new name to the class. It non-deterministically selects a numb, which must be an instance of the class, NUMB, but not already used in a link of the association pbook (see μB guard), and uses this as the link for the new instance (see μB action). The remove event (which is a destructor of class NAME) has no μB action; its only action is the implicit removal of self from the class NAME. This specification is equivalent to the Event-B model shown in Fig. 2.2 and indeed the U2B tool automatically produces the Event-B model of Fig. 2.2 from the UML-B version in Fig. 2.1.

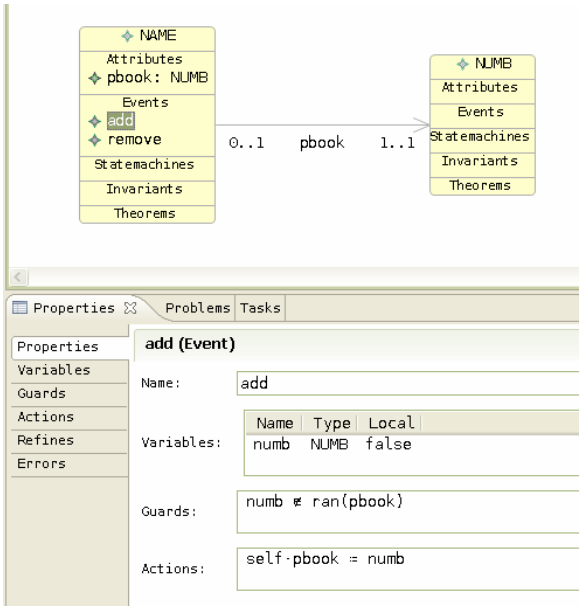


Fig. 2.1. UML-B Specification of a phone book

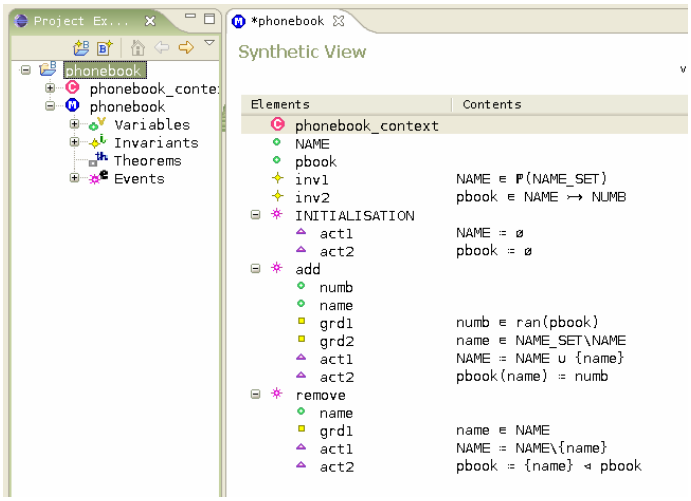


Fig. 2.2. Event-B specification of a phone book

2.1 Overview of plug-in functionality

The UML-B modelling environment consists of a project creation wizard that creates and initialises a UML-B project folder and provides and initial empty UML-B model. The UML-B builders are associated with the project so that they run automatically whenever resources (files) are saved in the project. Four interlinked diagram types (package, context, class and statemachine) are provided. The top-level package diagram is opened with an empty canvas by the wizard. This canvas represents the UML-B project. Other diagram types are linked and opened via model elements as they are drawn on the various canvases.

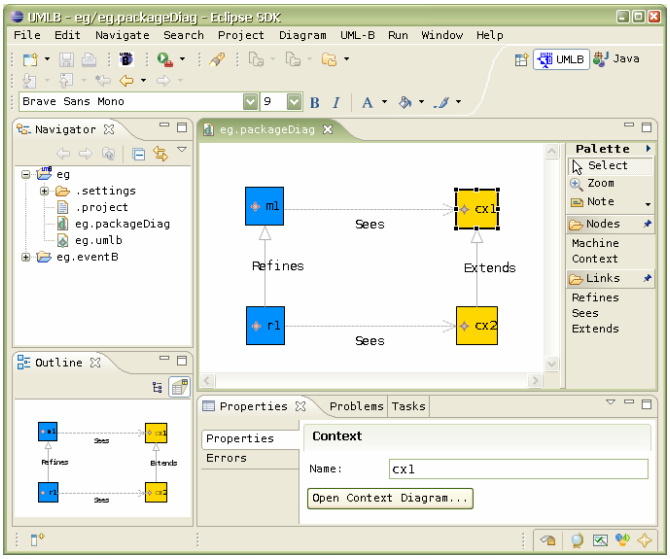


Fig. 2.3. Example package diagram

Package Diagrams

Package Diagrams are used to describe the relationships between top level components (machines and contexts) of a UML-B project. The diagram shows the machines in a UML-B project and the refinement relationships between them, the contexts and the extension relationships between them and which contexts are seen by each machine. Fig. 2.3 shows an example of these relationships between two machines (blue) and two contexts (yellow). Notice the properties view at the bottom of the perspective. This is where the property details of model elements are configured and where error messages will be reported. In this case context cx1 is selected in the drawing and the properties view contains a button to open the context diagram for cx1.

Context Diagrams

The Context diagram is used to define the static (constant) part of a model. This reflects the use of contexts in Event-B. ClassTypes are used to define given sets of instances and then to define constant attributes that are based on that set (i.e. lifted). For example, Fig. 2.4 shows a ClassType PERSON that has an attribute, id. The properties for the attribute provide control over the cardinality features of the attribute. In this case, we wish all instances of PERSON to have exactly one id and for that person's id to be unique. Hence, we have set the functional, total and injective properties. (Functional and total are default values for attributes). Fig. 2.4 also shows the Event-B context that has been produced from this UML-B context by the U2B tool. In the Event-B version of the context (as shown in the upper right hand view pane), sets are denoted with a purple star icon, constants with a yellow circle and axioms with a green star.

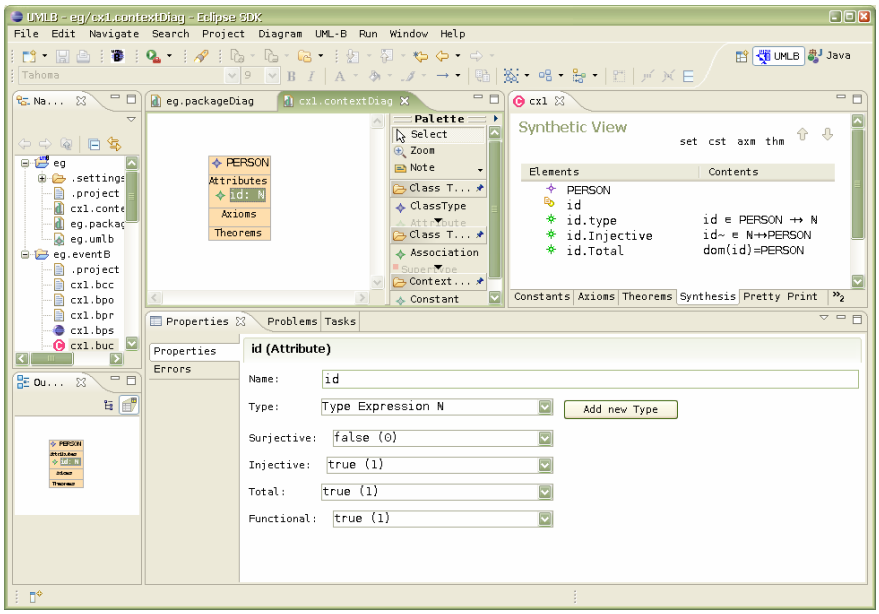


Fig. 2.4. Example Context diagram showing properties view of an attribute and Event-B translation.

Fig. 2.5 shows further features of the context diagram. An association, accounts, provides a constant in exactly the same way that the attribute, id, did. The only difference is that associations are shown graphically as a connection between two ClassTypes and do not default to functional and total. Hence, UML-B simplifies the treatment of associations compared to UML since UML-B associations are always uni-directional and contained by the source whereas UML associations are uncontained independent elements that are referenced by the two roles belonging to the two classes involved. This simplification makes translation into Event-B easier by removing the full flexibility of UML associations. However, since the UML approach would require multiple redundant variables which would be very undesirable when proving their consistency, the loss of flexibility is desirable.

The ClassType, PERSON has the ClassType CUSTOMER as its superset. Hence it is translated into a constant which is a subset of the given set, CUSTOMER. The ClassType, ACCOUNT has its instances property set to accounts. The instances property provides a means to model a ClassType from a set of instances defined elsewhere within the context. In this example ACCOUNT is actually the set of link mappings in the constant association, accounts. Hence, ACCOUNT corresponds to UML association classes. The mechanism is more flexible than association classes since any expression resulting in a set can be used, including predefined types and their derivatives such as $\mathbb{N}$, $\mathbb{P}(\mathbb{N})$ etc. and expressions using other features within the model, e.g. BANKxBANK. The Constant, interestRate, is independent of any ClassType providing a mechanism for defining constants that are not lifted by, or dependent on, any ClassType.

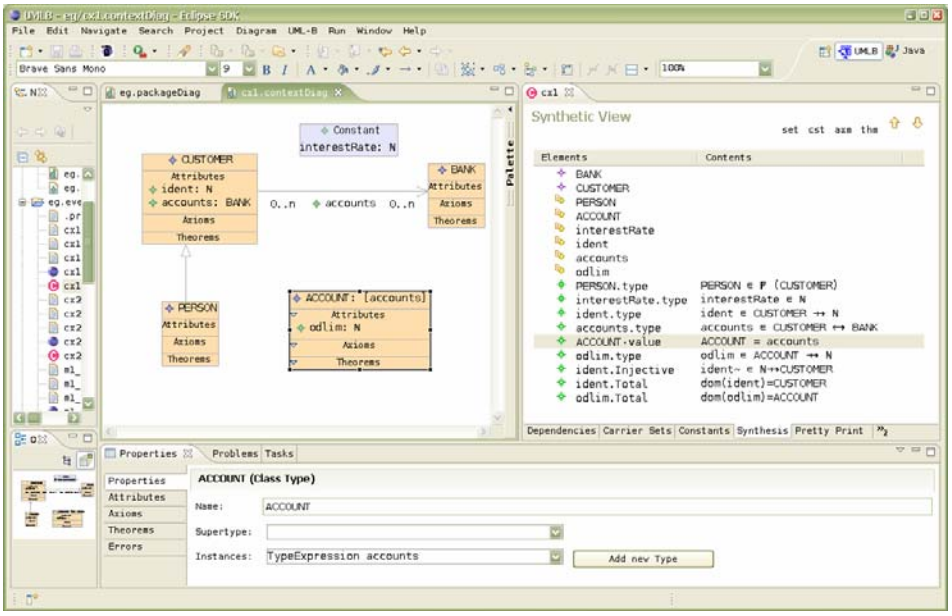


Fig. 2.5. Example Context diagram showing ClassTypes and their Event-B translation.

Class Diagrams

The class diagram is used to describe the behavioural part of a model. Classes represent subsets of the ClassTypes that were introduced in the context. This subtype relationship is explicitly defined in the class' properties. The class' associations and attributes are similar to those in the context but represent variables instead of constants. Actions and constraints (i.e. guards, invariants and theorems) are expressed in the μB notation. In μB, the owning instance of a class attribute is specified using the dot notation. For example i.x refers to the value of the variable x belonging to instance, i. When an expression is attached to a class, the owning instance for the current contextual instance is referenced using the reserved word, *self*. Note that the value represented by an expression i.x depends on the cardinality of the variable x. If it is a function, a single value is represented; if not, a set of values (corresponding to the relational image) is represented. If no instance is specified the expression gives the complete class–wide value of the feature (i.e. the complete relation rather than the value for a single instance).

An example of a class diagram is shown in Fig. 2.6. The bank class has an association, accounts, with the account class which will be translated into a variable, accounts, of type bank $\leftrightarrow$ account and initialised to $\varnothing$. Additional invariants giving the functional nature of the inverse relation and coverage of the range, reflect the 1..1 cardinality at the source end of the association. The attribute, balance, of class, account, defaults to a total function. A class invariant specifies that the account's balance must be greater than its overdraft limit, odlim. This invariant is written in the μB notation.

Fig. 2.6. Example Class diagram and its Event-B translation

The correspondence between an association's multiplicity constraints (introduced in Fig. 2.5 but also applicable to associations between classes) and the constraints on the resulting Event-B relationship is clear from the drawing tool. The multiplicity properties are described using the usual mathematical terminology (functional, total, injective, surjective) with the UML style multiplicity also shown and annotated automatically on the diagram. Table 1 shows the full correspondences.

Classes may contain events that modify their attributes. An example was shown in the overview given in section 3 (Fig. 2.1). Such events implicitly utilise a local variable that non-deterministically selects a valid current instance of the class. This instance is referred to via the reserved word self when referencing the attributes of the class. Constructors and destructors add or remove these implicit instances from the current instances of the class. We have also found it useful to model 'fixed' classes where instances cannot be added or removed. Many systems (e.g. embedded systems) have this, sometimes complex, static configuration. The use of generic UML-B models to verify and validate complex static configurations is investigated in [SJP05].

Even in systems that are essentially object-oriented, there are often singular features which do not require lifting to a class of objects. UML-B provides machine level features (events, statemachines, invariants and theorems) to be placed on the class diagram canvas without containment within a class. These features are translated directly to Event-B without any of the class instance additions described elsewhere. A complete non-object-oriented model can be constructed in this way. Although there would be little benefit

from the class diagram, the package diagram overview of the project is still useful and an orthogonal and hierarchical statemachine representation is available.

Table 1. UML-B association multiplicities and their Event-B translation

UML-B association properties					**Description of Event-B representation**
multiplicity	**surjective**	**injective**	**total**	**functional**	*(Ai and Bi are the instances sets of the classes)*
0..n → 0..n					relation
1..n → 0..n	*				Relation covering Bi
0..1 → 0..n		*			Relation and inverse is function
1..1 → 0..n	*	*			Relation covering Bi and inverse is function
0..n → 1..n			*		Relation covering Ai
1..n → 1..n	*		*		Relation covering Ai and Bi
0..1 → 1..n		*	*		Relation covering Ai and inverse is function
1..1 → 1..n	*	*	*		Relation covering Ai and Bi and inverse is function
0..n → 0..1				*	partial function to Bi
1..n → 0..1	*			*	partial surjection to Bi
0..1 → 0..1		*		*	partial injection to Bi
1..1 → 0..1	*	*		*	partial bijection to Bi
0..n → 1..1			*	*	total function to Bi
1..n → 1..1	*		*	*	total surjection to Bi
0..1 → 1..1		*	*	*	total injection to Bi
1..1 → 1..1	*	*	*	*	total bijection to Bi

Statemachine Diagrams

Statemachines attached to classes represent a variable of the class that partitions the behaviour of the class in some way. For example, the statemachine, bal_state, of class, account, partitions the behaviour of the account class into two states, black and red (Fig. 2.7). The transitions of a statemachine represent events with the additional behaviour associated with the change of state implied by the transition. That is, the event can only occur when the instance is at its source state and, when it fires, changes the state of the instance to the target state. In the previous version of UML-B, the transitions represented branches of a select statement within an event and all the transitions with a similar name were collated into a single event. With Event-B this is no longer possible since all the selection constructs have been removed. Hence each transition represents a separate event. As with events, event variables can be added to the transition to provide a non-deterministically chosen value to be used in the transition's guards and actions.

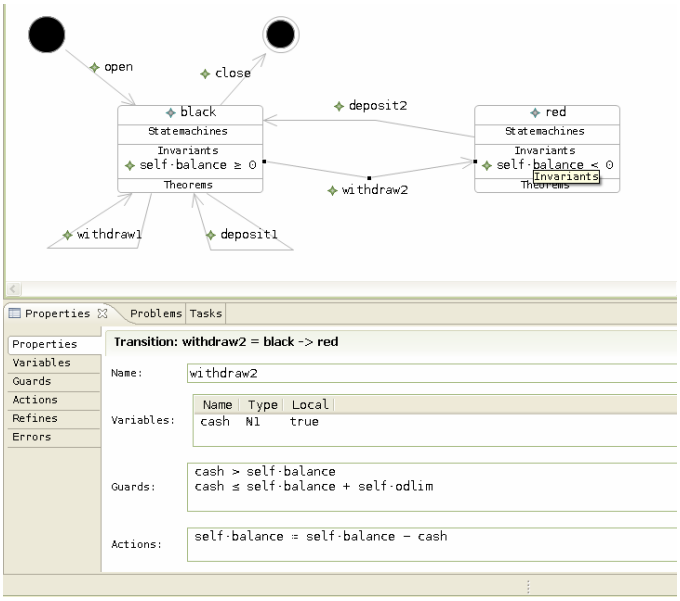


Fig. 2.7. Example Statemachine diagram showing ancillary properties of a transition

In order to define the type of the state variable, bal_state, the translation needs a given set that consists of the two states, black and red. This is defined in the implicit context for m1 as shown in Fig. 2.8.

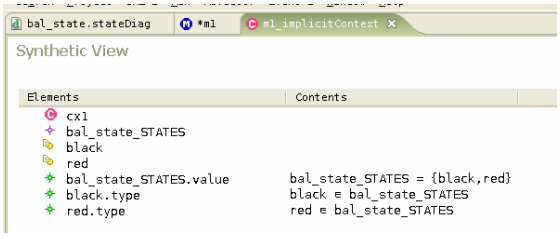


Fig. 2.8. Translation of example statemachine into Event-B (data parts)

Invariants may be attached to the states as shown in Fig. 2.7. During translation, these invariants are universally quantified over the class instances and constrained by an antecedent as shown in Fig. 2.9. This provides an efficient mechanism for linking the meaning of the states to other class variables.

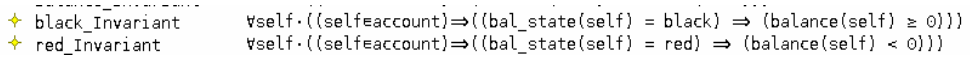


Fig. 2.9. Translation of example statemachine into Event-B (state invariants)

The translation of a transition (for example, withdraw2 is shown in Fig. 2.10) is similar to class events except that a guard for the starting state (source.state) and an action to move to the target state (target.state) are added.

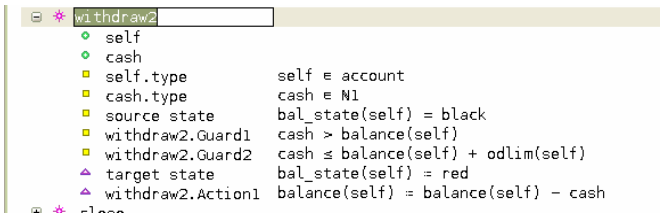


Fig. 2.10. Translation of example statemachine into Event-B (transition)

The transition from the starting state defines a constructor for the class. Hence the translation (Fig. 2.11) selects an unused instance and adds it to the set of current instances and initialises all the class variables for that instance. Similarly, the transition to a final state is a destructor and removes the instance from the current instances and from the domain of all the class variables.

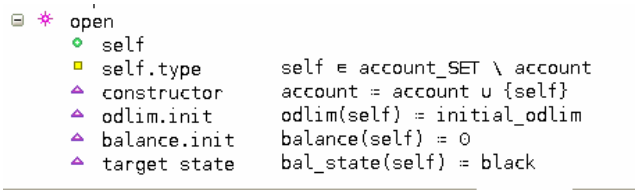


Fig. 2.11. Translation of example statemachine into Event-B (constructor)

An alternative, semantically equivalent, translation of statemachines is provided and can be selected per statemachine by setting a property switch in the diagram. In this alternative translation a variable is provided for each state which represents the instances currently in that state. The choice of translation is influenced by the model. For example, the alternative translation is useful when the transitions are guarded by the number of other instances in particular states, since it is then convenient to refer to the cardinality of a state. Fig. 2.12 shows the translation of the same example statemachine given in Fig. 2.7 but with the alternative translation selected. The states, red and black are represented as subsets of class instances and the event is guarded by self ∈ black. Two actions, leave_source and target_state are required to move self from black to red.

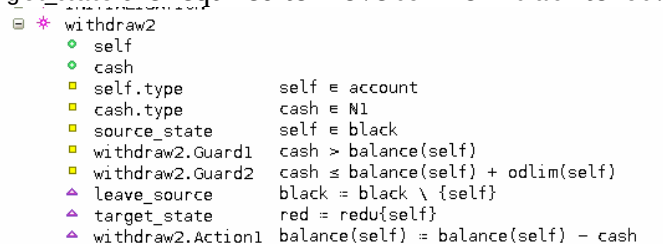


Fig. 2.12. Translation of example statemachine into Event-B transition using alternative translation

2.2 Overview of plug-in integration

The abstract syntax of the structure of the UML-B language is given by a metamodel using UML class diagrams with OCL constraints attached to some model elements. The UML-B metamodel uses a small subset of UML's class diagram features that is equivalent to the OMG's Metamodel Object Facility (MOF). Extensive use of generalisation ensures that commonalities in UML-B model elements are defined. The metamodel is a precise description of the abstract syntax of the UML-B language and is

used to automatically generate repository and editing utility code. Fig. 2.13 shows part of the UML-B metamodel to illustrate the modelling style. Italicised classes represent abstract model elements, instances of which can only exist via one of their subtypes. A base class, *UML-Belement*, provides a name and error marking scheme to all model elements. A subtype of this base class, *UML-BconstrainedElement* provides a base for elements that own constraints (axioms or invariants) and theorems. Note that the metamodel does not define the syntax of predicates, merely representing them as a string attribute of the **UML-BPredicate** class. One subtype of *UML-BconstrainedElement* is *UML-Bconstruct* which is further subtyped into **UML-BMachine** and **UML-BContext**. These reflect the main modelling components of Event-B. Fig. 2.13 also shows that **UML-BMachine** can contain **UML-BClass** and **UML-BContext** can contain **UML-BClassType**. Fig. 2.13 omits many features such as statemachines, variables and events that are contained within the metamodel.

In some cases, where constructing a fully constraining graphical model is not possible, OCL constraints are added to the model. An example is the metamodel for states and transitions where a state that has the initial attribute set is not allowed to have incoming transitions. It may have been possible to subclass states in some way so that initial states were prevented from having incoming transitions. However, there is a similar need to prevent final states from having outgoing transitions and it was felt that a graphical depiction of this situation would complicate the model. OCL constraints are either implemented within the graphical modelling tool to prevent invalid models being created or are used in a pre-translation validation stage to ensure that the model is well-formed.

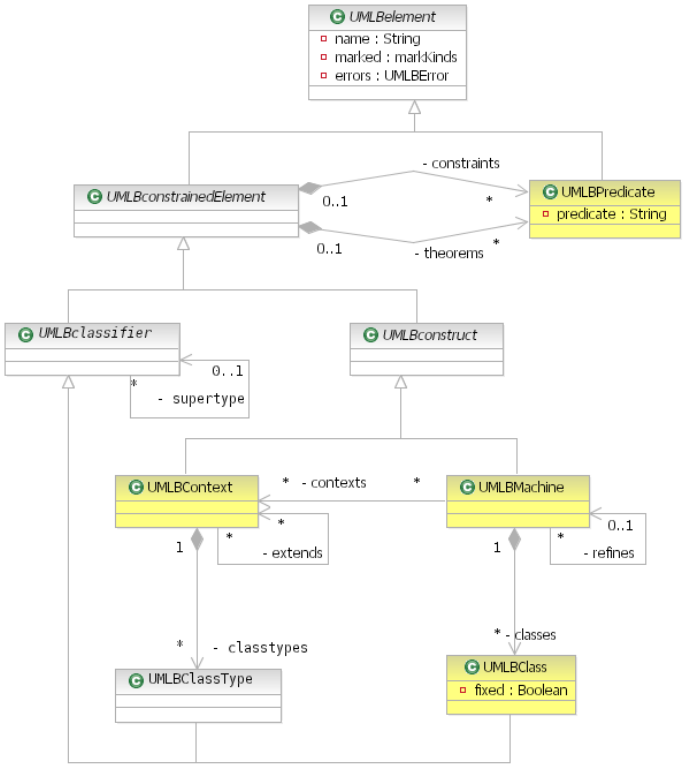


Fig. 2.13. The UML-B metamodel (part of).

The EMF (Eclipse Modelling Framework) [EMF] is an Eclipse project that automatically generates code for a model repository, model editor and API utilities based on an object model. The EMF generated code provides utilities to programmatically create and manipulate instances of the metamodel with serialisation provided in XMI. The GMF (Graphical Modelling Framework) [GMF] is another Eclipse project that, after a fair amount of configuration, will automatically generate code for a graphical modelling tool based on an EMF model.

The UML-B metamodel (i.e. the full version of Fig. 2.13) was imported into EMF in order to generate the Eclipse plugins necessary to support the UML-B modelling language. The GMF was then use to generate the UML-B graphical modelling tool. Drawings created using the UML-B modelling tool are saved as serialised UML-B model files. An Eclipse 'builder' responds to changes to such model files and translates them into a RODIN Event-B project. In order to do this it uses the API of the RODIN database to create a RODIN Event-B project containing machines and contexts and add Event-B elements to these machines and contexts.

When these constructs are saved by the U2B program, the RODIN verification tools (also Eclipse builders) automatically verify the Event-B model and report any errors. A final stage, which is not yet complete, is to listen for these errors and annotate the UML-B diagrams so that a user can work entirely in the UML-B environment and benefit from the powerful static verification and prover technology provided by RODIN Event-B. It is still expected that there will be proof obligations where the prover requires human

assistance to discharge. This requires the modeller to switch perspective to the Event-B prover environment and to work in the Event-B notation. However, one of the primary goals of Event-B is to achieve better rates of automatic proof so that these instances are reduced.

2.3 Conclusions

The advantages of moving away from a UML extension (profile) to a completely new metamodel are that the language is more suitable and elegant for expressing Event-B modelling concepts. The full expressivity of UML is largely redundant and can confuse users. A profile would rely heavily on well-formed ness constraints, whereas these are mostly built into the UML-B metamodel. UML-B still retains sufficient commonality with UML for the main goals of approachability to be attained by industrial users.

UML-B provides a fully integrated graphical front end for the Event-B modelling language. UML-B has similarities with UML that make it more approachable for users that are used to using UML. Since UML-B automates the production of many lines of textual B, models are quicker to produce and hence exploration of a problem domain is more attractive. This assists novices in finding useful abstractions for their models. We have found that the efficiency of UML-B in quickly generating large amounts of textual formal B and its ability to divide and contextualise small μB predicates and expressions assists novices who would otherwise, rightly or wrongly, be deterred from writing formal specifications. Furthermore, the new event oriented UML-B with its strong integration with the RODIN Event-B tools is gaining acceptance as a useful visual aid even with experienced formal methods users.

2.4 References

[EMF] The Eclipse Modelling Framework Project, http://www.Eclipse.org/modeling/emf/?project=emf

[GMF] The Eclipse Graphical Modelling Framework, http://www.Eclipse.org/gmf/

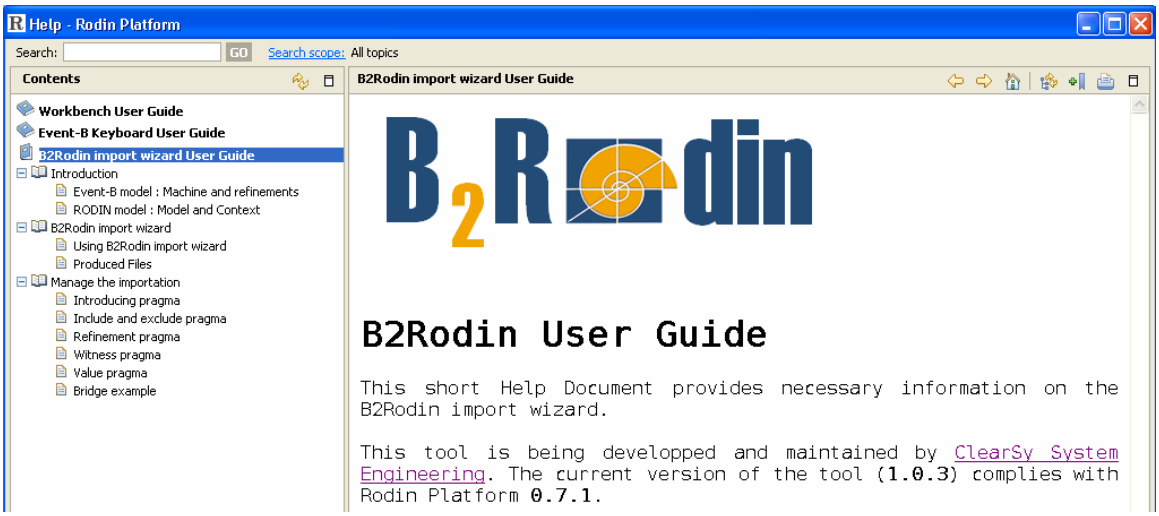
[SnBu06] C. Snook and M. Butler, UML-B: Formal modeling and design aided by UML, *ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15 , Issue 1* (January 2006) pp. 92 – 122, 2006

[SPJ05] C.Snook, M. Poppleton and I. Johnson, *Rigorous engineering of product-line requirements: a case study in failure management*, submitted for publication.

[RSA] Rational Software Architect, http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html

3 B2Rodin Plug-in

The **B2Rodin** tool allows reusing existing B models within the RODIN platform. Such event B models should comply with [EVT2B]. The **B2Rodin** tool is reachable at the update site http://www.b4free.com/b2rodin.



3.5 Overview of plug-in functionality

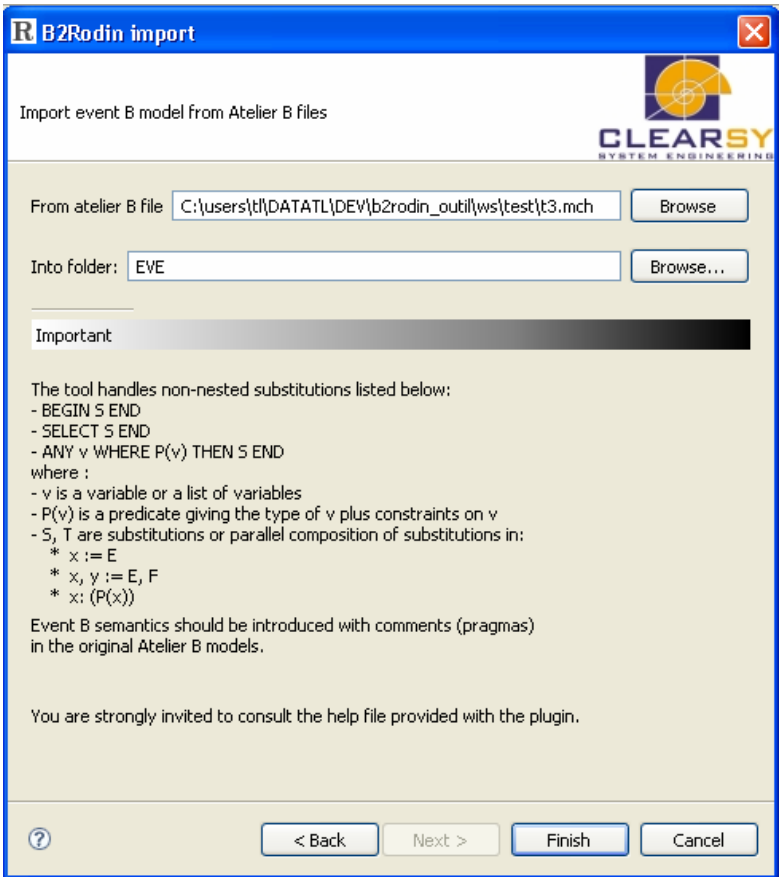
B2Rodin translates event-B based models, complying with [EVT2B] and parsable with Atelier B/ B Compiler, into RODIN models. This means that not all operations provided by the B language [BMREF] are authorized when writing an event-B model.
The B model to translate is composed of:

- an abstraction, contained in a machine file (*.mch),
- zero, one or multiple, successive refinements, contained in refinement files (*.ref).

Optionally, Writing an event-B model into *.sys files in order to get the « split » and « merge » feature is recommended. Then, from the *.sys files, another tool called **Evt2B** allows the translation into Atelier B compliant files (*.mch and *.ref).

The output of the B2Rodin tool is a set of:
- *.bum files, containing the models (original abstract machine and refinements);
- *.buc files, containing the contexts of the model (sets and constants)..

Prior to any translation, Atelier B compliant files need to be decorated with extra information, to introduce event B semantics: pragmas.



A pragma is a piece of information, inserted as comment in the source code, needed by **B2Rodin** to be able to link events and variables between a refinement and its abstraction. Several pragmas are available:

Initialisation pragma

The initialization pragma indicates which variable or formula of the refinement is able to remove the nondeterminism of an initialisation substitution. The variable or the formula is called a *witness.*

Variable pragma

The variable pragma indicates which variable or formula of the refinement is able to remove the nondeterminism of a substitution or an ANY. The variable or the formula is called a *witness.*

Refinement pragma

The refinement pragma indicates which event of the abstraction an event of the refinement refines. The pragma is written into a commentary before the concerned event of the refinement in the *.ref file.

Include/Exclude pragma

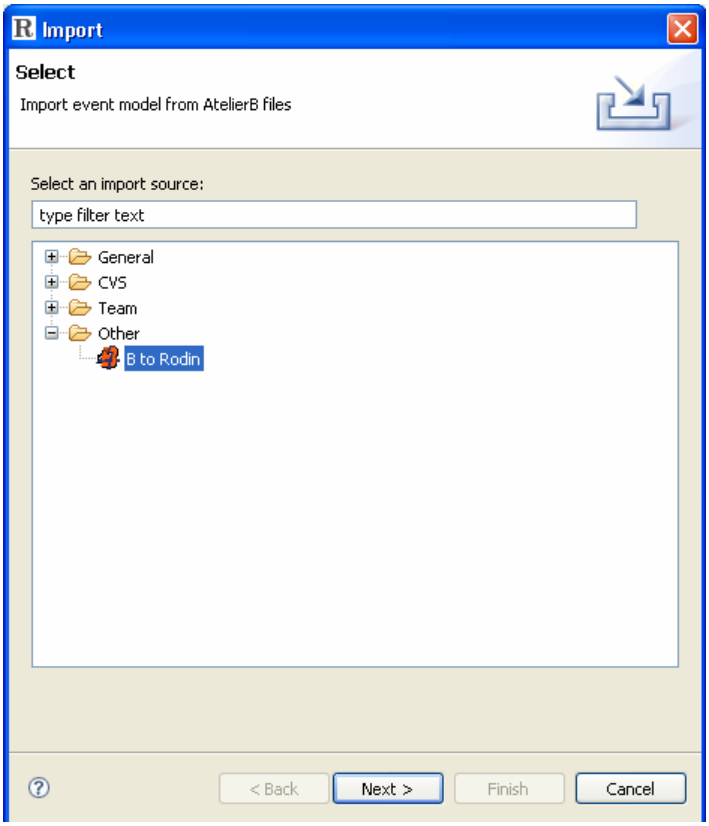
Given a refinement and all its upper abstractions, the *include* pragma indicates which events will be translated into the *.bum files. Given all the upper abstractions of a refinement, the *exclude* pragma indicates which events will be removed from the *.bum refinement associated file and all further refinements. They must be written at the beginning of the OPERATION clause. Once an event is included, it is implicitly copied in the lower refinement unless it is excluded. It's not possible to exclude an event that has not been included in an upper refinement. It's not possible to include an event that has been excluded in an upper refinement.

The generation process is initiated with the model we would like to transform and the level of refinement we would like to start from. A refinement column can be partly or completely translated, the abstract part of the model being always part of the translated files. For each *.mch or *.ref file, a bum file is generated. If needed, a *_ctx.buc file (associated context) is generated. If a refinement introduces no new set or constant, the associated bum model just uses the previous context model declared. On the contrary, the bum model adds its own partial context and sees the context of its abstraction. The generation process overwrites existing *.bum and *.buc files, if any.

3.6 Overview of plug-in integration

B2Rodin is a plug-in, contributing to import wizard and to help categories. The tool is organized in several layers:

- the core, developed in C++ as an extension of the B Compiler / Decompiler;
- the finalizer, developed in xsl, for applying normalizing rules;
- the plug-in interface, developed in Java, connecting the core to the various Eclipse and RODIN services;
- the help services, written in xml and html.



This plug-in is completely integrated within the Eclipse Platform and may be updated online with the update site feature.

References

[EVT2B] MATISSE: The event B reference manual, http://www.atelierb.societe.com/ressources/evt2b/eventb_reference_manual.pdf

[BMREF] B Language Reference Manual 1.8.5 , http://www.atelierb.societe.com/ressources/manrefb.185.fr.pdf

4 Brama Plug-in

Brama is a tool for animating event B models, with two objectives:

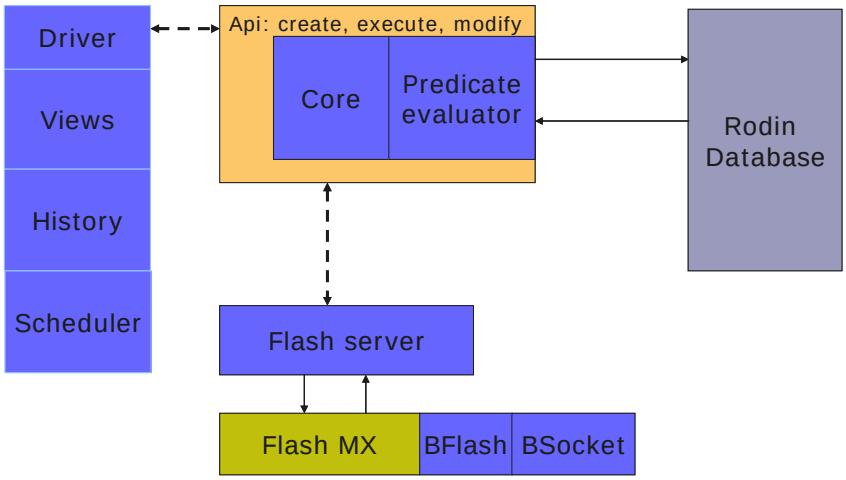
- enable the debugging of a multi refinement level model to get convinced that the model behaves as expected,
- show a B model in a way that it becomes understandable by a non specialist, thus able to be validated by third party.

A dedicated website has been set up and is reachable at
http://www.brama.fr/index_en.html.

4.1 Overview of plug-in functionality

Brama is composed of several parts:

- *Animation Engine*,
- Communication Manager between the Animation Engine and Flash MX,
- Graphical part, based on Flash.



The *Animation Engine* uses unchecked models (*.bum and *.buc) from the RODIN database. From such a model, the *Animation Engine* creates its own set of objects, independent from the RODIN database, by using the predicate evaluator. The *Animation Engine* doesn't listen to RODIN database modifications and behave independently. When an animation is initiated, a picture of the model is taken and used for the rest of the animation. Further modification of the underlying model has no effect on the animation, until it is stopped and restarted. The *Animation Engine* is multi-threaded and can be commanded via a set of commands.

The Communication Manager is responsible for sending and receiving information/commands from/to Animation Engine/Flash-based graphical part. This communication part is socket-based.
Messages from Animation Engine to Graphical Part are:

- event fired,
- new variable value.

Messages from Graphical Part are:

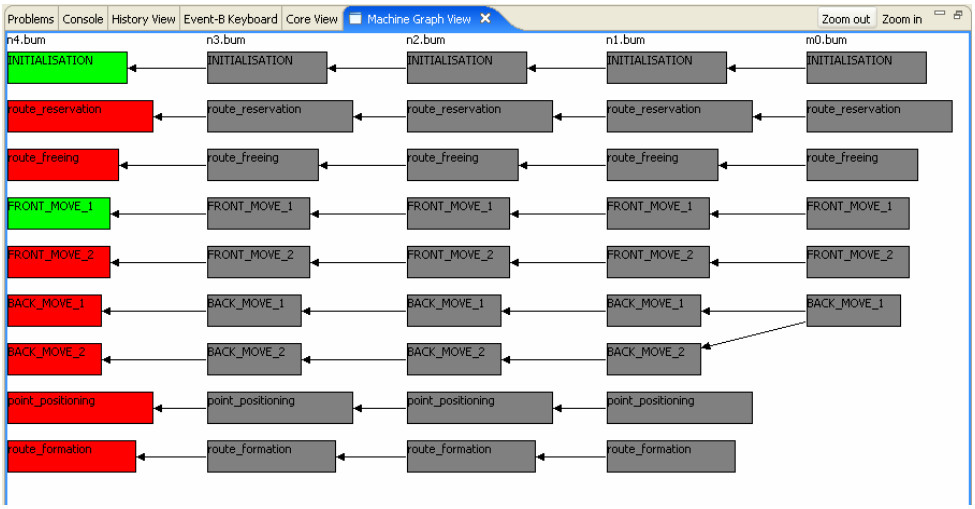
- event played,
- new variable value displayed,
- user interaction.

The graphical part is a Flash-based animation, connected to the Communication Manager and exchanging XML flows. It is developed using FlashMX. Animations are set up independently then connected to the underlying model, by specifying specific behavior upon reception of commands. The decoding routines, transforming XML flow in Flash commands, are common to all Flash animations using the Animation Engine.

The tool provides several services which are exemplified on the following pictures:

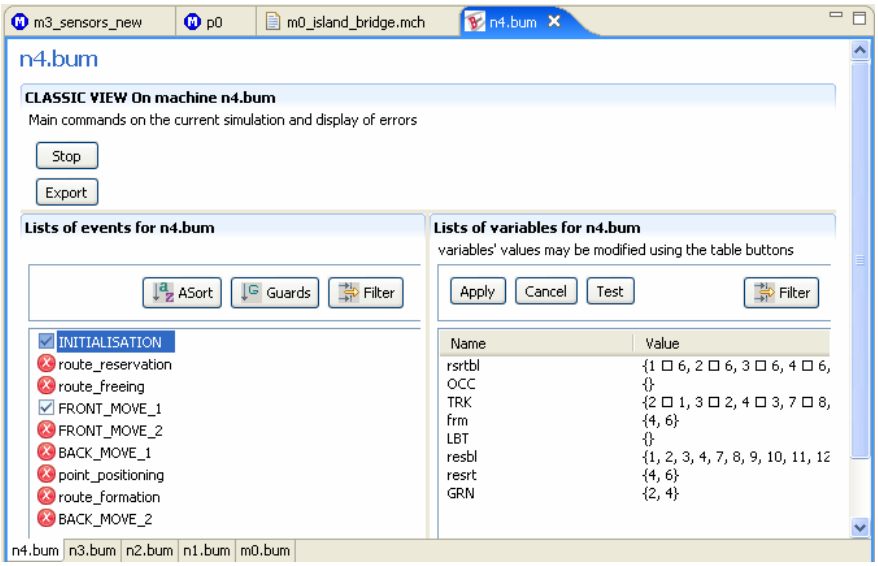
- Display the structure of a RODIN model

  While animated, the several refinement levels are showed in columns. The most abstract model is display on the right, the successive refinements are ordered from right to left. Boxes represent events and arrows represent refinement links. Green boxes are put for events that can fired, red boxes for events that can't be fired.



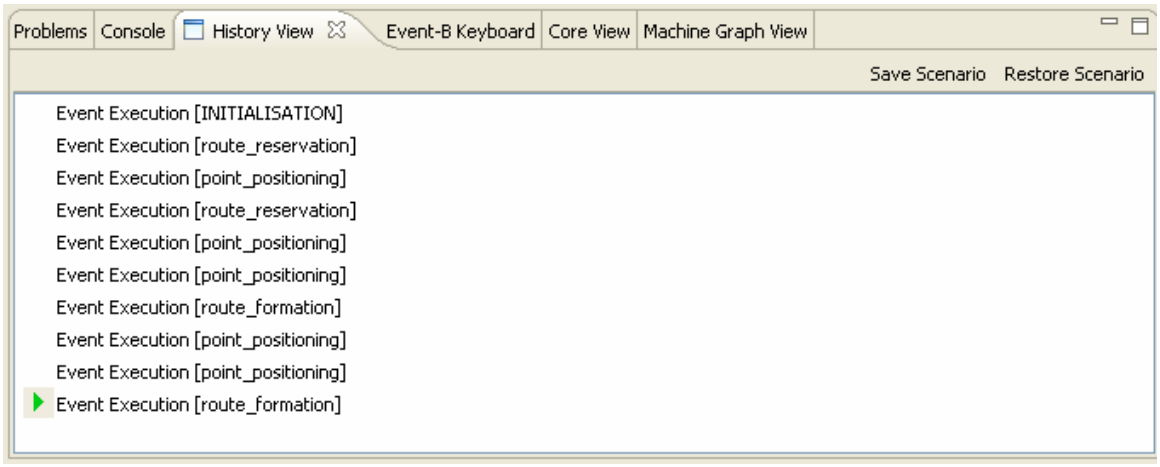
- Debug a RODIN model

  This view allows to:
  - fire events (the ones associated with a check box). For example, in the following picture, INITIALISATION and FRONT_MOVE_1 can be triggered. In case of non-deterministic choices substitution, the formula evaluator tries to find a correct valuation for variables. In case of failure, the user is asked to enter a correct value. Main requirement is to keep user interactions as low as possible;
  - sort events according to their name (ascendant, descendant), to their guard (open or closed first);
  - display values of variables;
  - check new variable value against the invariant;
  - apply new values to the model;
  - export a model, including scenarios (sequence of events, explicit valuations from user) as a stand alone animation.



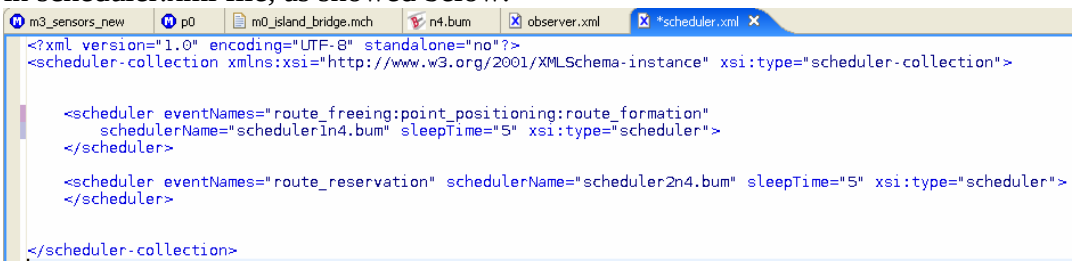
- Manage history:

  This view allows to:
  - trace event execution,
  - backtrack execution,
  - save scenario,
  - restore scenario.



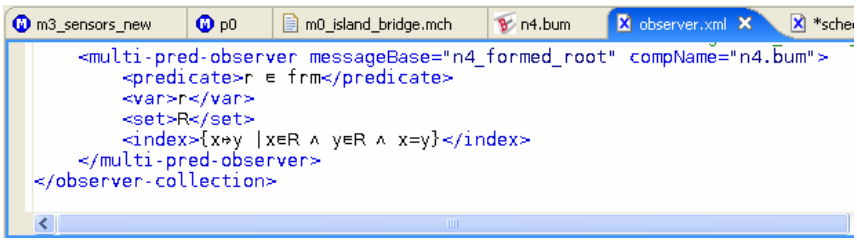
- Schedule events:

  In some cases (reflex behavior for example), we would like to have events fired automatically without user interaction. For that, schedulers may be explicitly defined, indicating which event(s) to trigger automatically and what delay (in ms) to apply once such an event is enabled. Those schedulers are declared and defined in scheduler.xml file, as showed below.



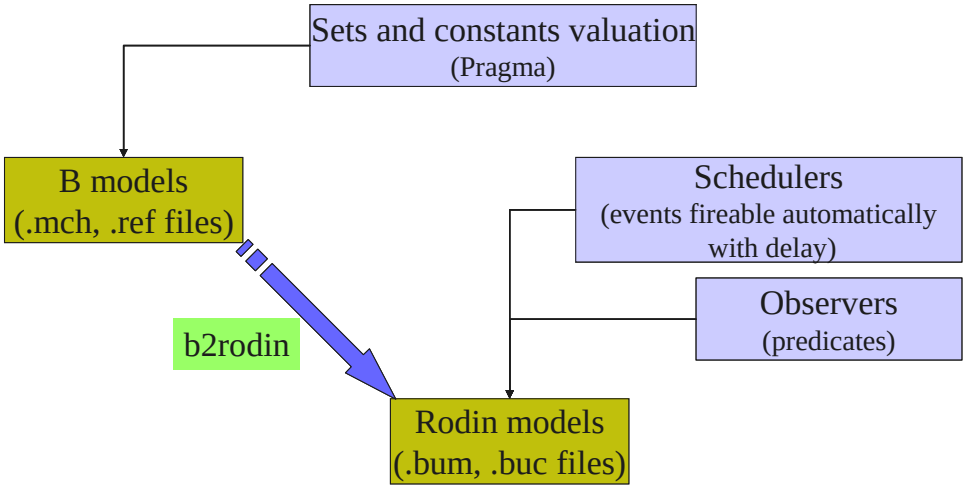
- Observe predicates:

  Animation may require extra information/computation. Predicate evaluation capabilities have been added. Predicates are defined in the observer.xml file and messages are sent to the Communication Manager when a predicate valuation changes.



4.2 Overview of plug-in integration

Integration in the RODIN platform requires having access to bum and buc files (unchecked models) in read-only mode. Brama can't modify the RODIN model. Brama is working on a stable RODIN model: further modification of a model is not taken into account in the animation when initiated.
The Brama tool is also connected to B2Rodin, according to the following schema, as B2Rodin provides sets and constants valuation required by Brama.



Scheduler.xml and observer.xml files are independent from the RODIN platform.

5 Mobility Plug-in

As planned in the original project proposal, the mobility plug-in was to be developed and primarily evaluated in the context of RODIN's Ambient Campus case study. As a result our work on the Petri net based model-checking has been conducted in close cooperation with this case study. Having said that, it is clear that the notation (and so input to the mobility plug-in) will be based on concepts and constructs coming from (or being based on) Event B, KLAIM and the existing work on model checking π-calculus. Under the title 'Mobile B Systems' a high level programming notation has been developed during the second year of the project for the specification of mobile applications. Furthermore, it

provides structured operational semantics through a set of rewriting rules. Finally, a translation was described from the programming notation to a class of high level Petri nets which preserves behavioural properties of mobile applications and, at the same time, makes explicit causal relationships between events involved in executed behaviours.
The work on 'Mobile B Systems' has reached the prototype plug-in implementation phase, and the folder accompanying the submission contains the plug-in implementation, example input files and documents explaining the main technical details.

5.3 Overview of plug-in functionality

The key components of the Petri net based mobility plug-in are as follows:

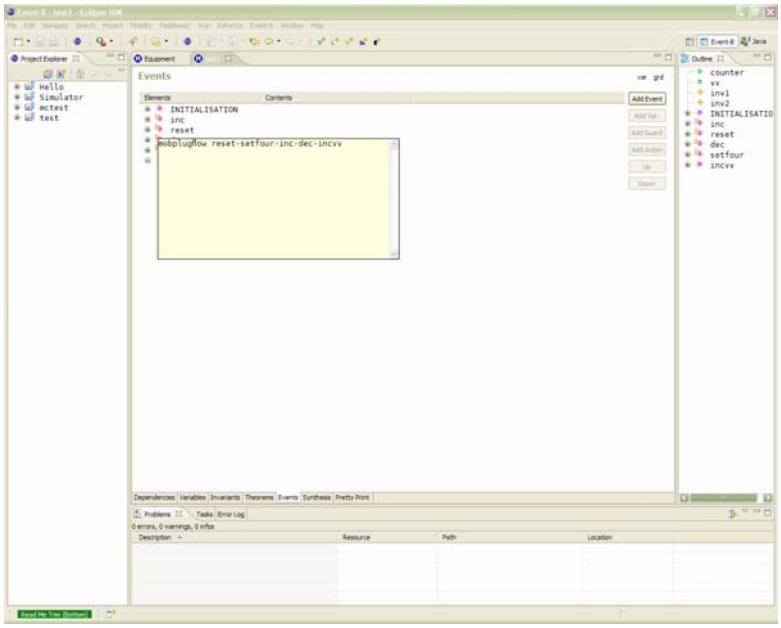
- *RODIN Platform* providing the Event-B specification of our model;
- *Process algebra editor* allowing the user to input/edit process algebra expressions;
- *Translator* taking as input Event-B specification and process algebra expressions and providing as output the 'Mobile B Systems' programming notation;
- *Translator* from the 'Mobile B Systems' notation to high-level Petri nets;
- *Unfolder* for deriving a finite prefix of the unfolding of the translated Petri net;
- *Verifier* which establishes, by working with the finite prefix, whether the necessary properties of the original input hold.

Translators are in place to ensure the combination of the Event-B notation with the process algebra expression. There are two key issues one needs to consider when building such translators. The first is a behaviour preserving translation of the combined specification into a high-level Petri net. The second issue is that the resulting high-level Petri net (to be more precise it is an M-net), must be accepted as input from the model-checking engine based on net unfolding. The translation from the modelling language to the high level net input to the model checker is completely automatic and hidden from the user. The theoretical details of this translation can be found in the technical documents accompanying the plug in submission.

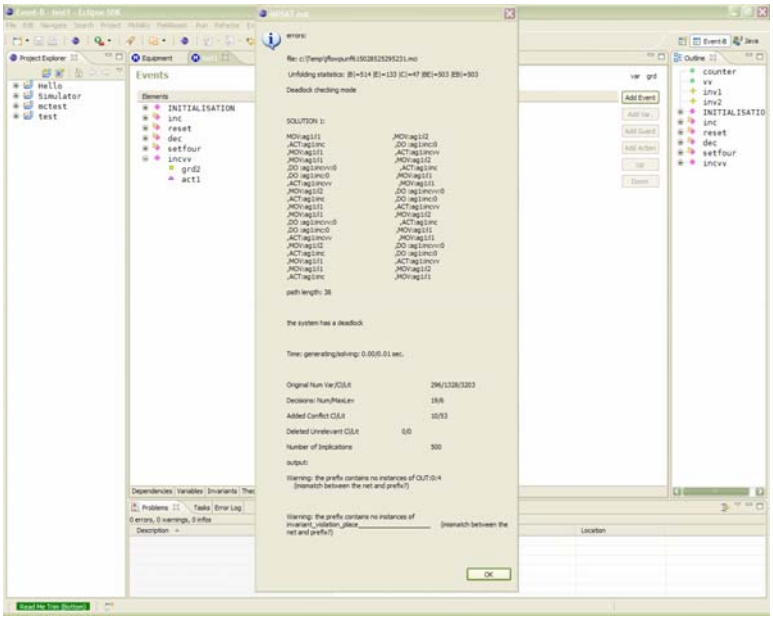
A 'Mobility' menu item has been added to the main window of the platform giving access to four major operations.

- Editing: Opens a text editor for inputting/editing a process algebra expression.
- Translation: From the Event-B model together with the process algebra expression to high level Petri nets.
- Unfold: Running the unfolder.
- Verify Properties: Running the verifier.

The user starts by creating an Event-B model of the specification. Then the user invokes the 'Editing' menu and adds the process algebra expression describing the distributed system composed of agents. It 'guides' the execution of the Event-B model quite similarly to the ProB+CSP case. The expression is saved in a file which is created and managed by the RODIN platform database. At this point, it can be noted that following the suggestions from the developers of the RODIN platform, we can use the platform's commenting system to store the process algebra expressions (see figure).



In the following step the user is ready to execute the two translators. Any errors found during the translation are also logged in the expression file and the user is informed via a pop-up window at the moment. If no errors were found then the output of the translation is saved in a file and the user will be informed of the successful translation. The actual file system is managing this file rather than the RODIN database since it is just a text file (.hl_net) with a format suitable for the unfolder to understand. Furthermore the file will be put in the same directory as the unfolder. Now it is possible to run the model checking engine starting with the execution of PUNF unfolder. After the successful creation of the high level Petri net prefix, MPSAT verifier can be invoked, the output of the verifier will also be stored with the help of the RODIN database to the process algebra expression file and the user will be informed. The model checking engine can perform two different and complementary tasks, namely checking for deadlock freeness and detecting invariant violations in the specification. In both cases in the event of an error discovery, the engine is capable of providing feedback that can be used for debugging. The error trace comes as a list of transitions' names of the high-level Petri net (or to be more precise the prefix of the Petri net). In most cases this type of feedback is not particularly useful to the user (especially users with no Petri nets experience). In order to improve the functionality of the tool, the 'translator' component used to obtain the high level net has been programmed to assign meaningful names (matching names from the modelling language specification) to each transition (see figure).



We are planning to take the following steps in the future development and enhancement of the mobility plug-in. At the moment the plug-in is offering fairly limited integration with the user interface of the RODIN platform. The user interface of the plug-in will change radically in the following months. The plan is either to provide a new 'Mobility' perspective in the platform or to extend the current 'Event-B' perspective with an extra 'Mobility' page. In both cases an editor for inputting process algebra expressions will be present together with a messages/results window (view). Furthermore, the handling of error messages will be treated in a more systematic way and the user will receive better quality feedback from the platform. Finally, on the algorithmic side of the plug-in we are planning to alleviate several of the current limitations. For example, we plan to extend the syntax of allowed process algebra expressions, making it possible to specify more realistic execution scenarios. This will involve an explicit support for tail recursion, and a possibility of specifying priority levels of atomic actions.

5.4 Overview of plug-in integration

As mentioned in the outline the way of integrating our work with the other parts of the platform is through the 'Mobile B Systems' programming notation. It should be stressed that this language is not used as an input for the plug-in but rather serves as the middleman. In this section, we will present the modelling language together with a small example. More details about the modelling language including its structural operational semantics and the complete translation to high level Petri nets can be found in [1]. This way it will become obvious how we managed to combine Event-B notation coming from the platform with a process algebra giving mobility characteristics to the model.
The modelling language specifications, called *scenarios*, are of the following form:

```
begin_scenario
```
$l_1 \ldots l_k$ (locations)
$rl_1 \ldots rl_m$ (roles)
$ag_1 = \texttt{new}(rl_1') \ldots ag_n = \texttt{new}(rl_n')$ (agents)

E (process expression)
`end_scenario`

As an example of the above, let us consider

```
begin_scenario
L1 L2 L3
buyer seller
b=new(buyer) s=new(seller)
b:move(L2).trybuy(hat,3).migrate(L1).
                  ⟨number(seller,L1)>0⟩trybuy(cap,3).nil
||
s:move(L1).updatestock(hat,2,4,20).
                  updatestock(cap,1,1,5).nil
end_scenario
```

The two roles `buyer` and `seller` are given below.

```
role buyer{
      var POW(item*price): expect ={};
      var POW(item*price): buys ={};

      event offer(item: i, price: p){
            if i:dom(expect) then
                  if expect(i) <= p then
                        trigger(buy,i,p);
                        buys := buys n/{i|-> p};
                        expect := {i} <<| expect
                  else
                        trigger(reject,i,p)
                  end
            end
      }
      action trybuy(item:i, price:p){
            expect:=expect n/{i|->p};
            trigger(request,i)
      }
}

role seller{
      var POW(item*price):startprice ={};
      var POW(item*price):minprice ={};
      var POW(item*NAT):items ={};
      var POW(item*price):sells = {};

      event request(item:i){
            if i: dom(items) & items(i) >0 then
                  trigger(offer,I,startprice(i));
            end
      }
      event reject(item:i, price:p){
            if i:dom(items) & p>minprice(i) then
                  trigger(offer,i,p-1)
            end
      }
      event buy(item:i, price:p){
            sells:= sells n/{i|->p};
            items(i):= items(i) - 1
      }

action updatestock(item:i, NAT:num, price:min, price:max){
```

```
      startprice:=startprice<+{i|->max};
      minprice:= minprice<+{i|->min};
      items:=items<+{i|->num};
}
}
```

The process expression describes a distributed system composed of agents, each agent being an instantiation of a role. Its general format is

$$ag_1{:}pa\_act_{11}.\ \ldots.pa\_act_{1m1}.\texttt{nil} \ \|\ldots\|\ ag_k{:}pa\_act_{k1}.\ \ldots.pa\_act_{kmk}.\texttt{nil}$$

where the ag_i 's are agents and pa_act_{ij} 's are process algebra actions.
A role specification describes a set of *events* and *actions* which are procedures that update role variables and initiate further computations. A role event is invoked by the `trigger` statement with suitable arguments invoked by an agent. For example, the action `trybuy` in the `buyer` role triggers event `request` in role `seller` which in its turn may trigger `buyer`'s event `offer` and so on. An action is invoked from within a process algebra expression, with constants or role variables as parameters. An action invocation may result in a chain of event invocations corresponding to communication between roles.
Executing `move`(*l*) changes the current locality of an agent, and the function `number` (*rl* , *l*) returns number of agents associated with the role *rl* in the locality *l*.
The process expression is constructed from basic actions, which can be of one of the following forms:

- `move`*(l)* moves the current agent (i.e., that labelling the sequential sub-expression in which the action appears) to location *l*.
- `migrate`*(l)* moves the current agent to location *l* provided that in its current locality there is no other agent which would want to trigger one of the events in *ag*.
- *act(ag,**d**)* calls action *act* in agent *ag* with the actual parameters ***d***.
- ⟨*bool*⟩ is a guard, where *bool* is a well-formed Boolean expression.

In addition to that we use prefix and (at the topmost level) parallel composition.

References:
[1] A.Iliasov, V.Khomenko, M.Koutny, A.Niaouris and A.Romanovsky: *Mobile B Systems*. Technical Report, School of Computing Science, Newcastle University (to appear in March 2007)

6 ProB Plug-ins

The developers of the ProB toolsuite for B at the University of Düsseldorf have developed 2 RODIN plug-ins based on ProB. Both of these are described in the papers in Appendix A and B respectively. They are briely outlined here.

6.1 RODIN ProB Visual Animation Plug-in

Writing a formal specification for real-life, industrial problems is a difficult and error prone task, even for experts in formal methods. In the process of specifying a formal model for later refinement and implementation it is crucial to get approval and feedback from domain experts to avoid the costs of changing a specification at a late point of the development. But understanding formal models written in a specification language like B requires mathematical knowledge a domain expert might not have. We have developed a new, improved method to visualize B Machines using the ProB animator. We have implemented this method as a plug-in to the open source Eclipse platform. We also support Event-B models developed in the new Rodin platform. Our new tool offers an easy way for specifiers to build a domain specific visualization that can be used by domain experts to check whether a B specification corresponds to their expectations.

6.2 RODIN ProB Disprover Plug-in

The B-method, as well as its offspring Event-B, are both tool-supported formal methods used for the development of computer systems whose correctness is formally proven. However, the more complex the specification becomes, the more proof obligations need to be discharged. While many proof obligations can be discharged automatically by recent tools such as the RODIN platform, a considerable number still have to be proven interactively. This can be either because the required proof is too complicated or because the B model is erroneous. In this paper we describe a disprover plugin for RODIN that utilizes the ProB animator and model checker 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) and she should not attempt to prove the proof obligation. 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.

Appendix A

The ProB Plug-In for Eclipse and Rodin*

Jens Bendisposto and Michael Leuschel

Institut für Informatik
Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. Writing a formal specification for real-life, industrial problems is a difficult and error prone task, even for experts in formal methods. In the process of specifying a formal model for later refinement and implementation it is crucial to get approval and feedback from domain experts to avoid the costs of changing a specification at a late point of the development. But understanding formal models written in a specification language like B requires mathematical knowledge a domain expert might not have. In this paper we present a new, improved method to visualize B Machines using the ProB animator. We have implemented this method as a plug-in to the open source Eclipse platform. We also support Event-B models developed in the new Rodin platform. Our new tool offers an easy way for specifiers to build a domain specific visualization that can be used by domain experts to check whether a B specification corresponds to their expectations.
Keywords: B-Method, Tool Support, Model Checking, Animation.

1 Introduction

The B-method introduced by J.-R. Abrial [1] is a theory and methodology for formal development of computer systems which is based on the notion of abstract machines and refinement. The state of an abstract machine consists of typed variables which are constructed from basic types and domain specific types using constructs from predicate calculus and set theory. The machines invariant is given as a predicate logic formula, operations are specified as generalized substitutions.
Refining an abstract machine means to stepwise remove nondeterminism from the operations or to represent the state of the machine by a more concrete data structure. If a refinement is at sufficiently low level, it can be translated into executable code (e.g., Spark Ada or C using AtelierB [14]).

Proving correctness of a specification requires two activities in classical B: consistency checking, which proves that each operation preserves the machines invariant and refinement checking, which proves that one machine is a valid refinement of another. These proof activities are supported by various tools

* This research is being carried out as part of the EU funded research projects: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

Appendix A

(AtelierB [14], B4free, B-Toolkit [4]) which extract a list of proof obligations (POs) as predicate logic formulas from a specification. A substantial part of the POs can often be proved automatically, but in general some additional interactive proving has to be done. The interactive part is expensive and should be kept small or at least be done only once. Therefore a typical development is going through several cycles of refinement before attempting to do any interactive proof. Also it is not unfrequent that changes in the specification are required, meaning that many or even all proof obligations need to be reproved.

In previous work [11] we presented the Prolog based ProB animator and model checker[1]. Two of the goals of ProB are:

1. allow a user to gain confidence that the specification that is being refined and implemented does meet the requirements; this is the main goal of the animation component of ProB, allowing to check the presence of desired functionality and to inspect the behaviour of a specification
2. assist the user in the proof effort by finding counter examples to the consistency and refinement conditions, helping a user to locate errors and avoiding wasted effort inside the interactive prover.

In this work we present various improvements to the ProB tool, mainly aimed at making it a better tool for bringing formal methods to industrial developers and domain experts. In summary, the main contributions of this work are:

1. Integration into the Eclipse platform
2. A notable recent development in the B world is the RODIN platform [13], which is an open tool platform based on Eclipse[2] to support Event B [2], an evolution of B to specify reactive systems. The current version of ProB supports so-called "classical" B [1] and in order to support this new language we needed to integrate ProB into the Rodin Eclipse platform. Another incentive for this move lay in improving the graphical user interface of the tool, which was originally developed in (and limited by) Tcl/Tk.
   We thus present the integration of the ProB tool inside Eclipse, improving the user interface and enabling future direct interaction with the core Rodin proof components and visual editor.
3. It is important that specifications can be animated in such a way that domain experts can easily validate whether the specification corresponds to their expectations. While ProB allows automated animation, the visualisation may still be difficult to understand for domain experts not versed in formal methods. For example, the state of the B machine is expressed using mathematical, set-theoretic constructs. So are operation arguments and return values.
   To overcome this hurdle we have developed a generic Flash-based animation engine which allows to easily develop visualizations for a given specification.

[1] available at http://www.stups.uni-duesseldorf.de/ProB
[2] http://www.eclipse.org

Appendix A

This generic Flash movie connects through a TCP Socket to the Server, that is integrated into a new Eclipse RCP based release of the ProB animator shown in figure 13. Our tool supports state-based animations, which use simple pictures to represent a specific state of the B-model, and transition-based animations consisting of picture sequences. To avoid the creation of many different animations the tool supports the composition the visualization of several separated parts.

2 ProB

To avoid proving the wrong specification it is useful to check a specification with an animator like ProB [11]. Animating a B specification is a convenient way to gain confidence in a formal specification. In contrast to earlier animators (such as the one provided by the B-toolkit), the ProB animator is fully automatic and does not require the user to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. In some special cases, when the state space is finite, it is possible to do a complete proof with ProB's model checker. ProB supports two kinds of consistency checking.

1. The temporal model checker explores the statespace starting in the initial state. In case of a inconsistency it returns a trace leading from the initial state into a state where the invariant is violated. These traces might help to debug a specification by indicating where the problem occurs, this is not easy by doing only mathematical proofs. Also the traces can be saved for later testing.
2. The constraint based checker tries to find a state, where applying a single operation leads into an invariant violating state. This is done by symbolic constraint solving.

In addition to the support for the developer of a formal specification, ProB can be used to communicate a formal model to a domain expert for approval. The automatic animation allows a non-expert to "play" with a formal model, while the state space visualization features [12] provide a graphical representation of the behaviour of a specification.

3 ProB for Eclipse

To use ProB as a part of the RODIN platform we developed an Eclipse plug-in version. We adopted the design philosophy of the Eclipse platform and splitted ProB into a set of plug-ins. (Figure 1) Eclipse is a collection of places-to-plug-things-in (extension points) and things-plugged-in (extensions). Multiple extensions, even if they have different purposes, can plug into an extension-point as long as they implement an extension point specific interface. A plug-in is a piece of software that contributes extensions to various different extension points or offers extension points for other plug-ins. [6]

Appendix A

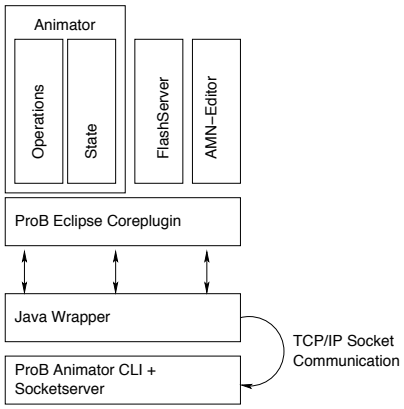


Fig. 1. Architecture of the ProB Eclipse Version

3.1 Wrapper for the Prolog based Portion

We extracted a command line (CLI) version of ProB and added a socket server. This enables ProB to be remotely controlled by a Java program. In our case we developed a low level Java wrapper that maps Prolog answers to Java objects. This wrapper is independent from Eclipse, it can be seen as a Java interface to the Prolog core. It sends Prolog queries and retrieves the answers as a string is being parsed into Java objects. Although SICStus Prolog[3], which we use for ProB does have a TCP-based solution for Java-to-Prolog communication called PrologBeans we decided to write our own implementation. Mainly because PrologBeans are stateless, it is not possible to interactively ask Prolog for more solutions to a query. SICStus Prolog also provides the Jasper interface, which does not have this limitation; but SICStus discourages developers from using the interface.[4]

3.2 ProB Core Plug-in

On top of this interface we built a core plug-in for Eclipse, that works as a foundation to all other plug-ins. The core plug-in offers an interface (Figure 2) that can be accessed by other tools to interact with the CLI. Also it notifies other plug-ins if the state of a machine changes or if a new machine was loaded into ProB. Therefore we defined two extension points *statechange* and *machinechange* and two corresponding simple interfaces. (Figure 3 and 4) The core offers some methods that can be used for *predictive* state space exploration.

[3] A commercial Prolog development system - http://www.sics.se/isl/sicstus.html

[4] And we have had our share of problems in getting it to function properly on the various platforms we support.

Appendix A

This enables the calculation of states the user will most likely visit next to improve the feeling of fluent working. Also it might be used for parallel exploration in future.

```
public void loadMachine(File name);
public List<Operation> getAllOperations();
public List<Operation> getOperations();
public State getState();
public List<Operation> getArg(String op);
public void executeOperation(Operation op);
public void setCurrentState(String StateID);
public boolean invariantKO();
```

Fig. 2. IProBCore core plug-in Interface (Excerpt)

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension-point id="statechange"
    name="de.hhu.stups.prob.core.statechange"
    schema="schema/statechange.exsd"/>
   <extension-point id="machinechange"
    name="de.hhu.stups.prob.core.machinechange"
    schema="schema/machinechange.exsd"/>
</plugin>
```

Fig. 3. ProB core plug-in plugin.xml

Typically the core is being notified that the user wants to load a machine by an external plug-in. In our case this would be the AMN editor. The core will take care of loading the machine into the CLI and notify registered plug-ins. Information about the current state such as values for variables and constants and enabled operation are automatically being retrieved from the CLI and a notification is being sent to all registered plug-ins.

3.3 AMN-Editor

The AMN[5] editor is the interface that allows the user to tell ProB which B specification he wants to animate. As soon as a file is loaded into the editor or selected as active tab (Figure 5) the editor calls the *loadMachine* method from

[5] B specifications are written in abstract machine notation (AMN)

Appendix A

```
public interface IStateChangeListener {
    public void StateChange(State s);
}

public interface IMachineChangeListener {
 public void MachineChange(Machine machine);
}
```

Fig. 4. Extension points interfaces

the core (i.e., it tells the core to load and initialise the corresponding machine). To improve the work with AMN files we developed a syntax-highlighting based on the Eclipse *AbstractDecoratedTextEditor* [6]. Currently the plug-in supports highlighting of B keywords. If the user saves a modified specification the editor restarts the animation.

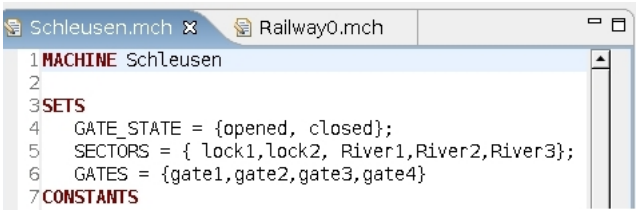


Fig. 5. AMN Editor

3.4 Animator

The animator plug-in (Figure 6) consists of two views:

1. The State view shows information about the current state of the machine. Constants and variables are shown separately, also the status of the machines invariant is displayed in an intuitive manner.
2. The operation view shows a list of all operations that are declared in the B specification. Operations which cannot be applied in the current state (because of precondition) are grayed. Enabled operation are displayed as a tree view. ProB calculates parameters to each operation from finite sets. This might be expensive to calculate, therefore the operation view does only ask for a single solution for each operation. Only if a solution exists the operation will be enabled. As soon as the user clicks on the operation more solutions will be calculated. This behavior reduces the risk of calculating things the user never wants to see and thus improving the performance of the user interface.

Appendix A

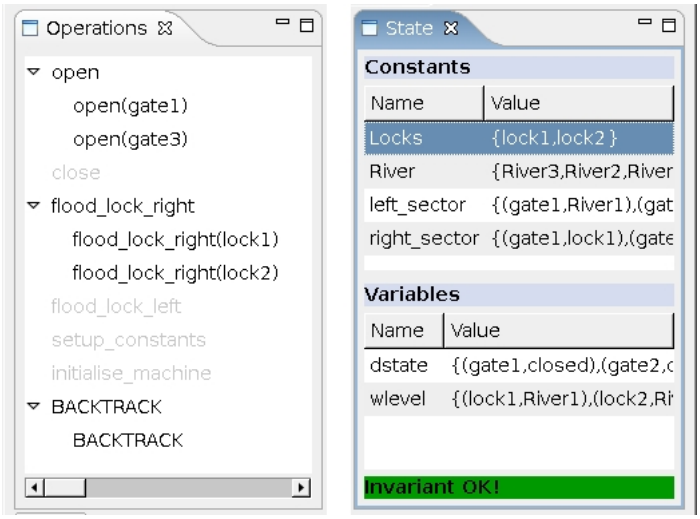


Fig. 6. Animator views

4 Flash server

On one hand a domain expert typically has no interest in crawling through a big and complicated specification. Even using classical ProB might be too complicated, because the domain expert still needs a certain level of knowledge about the mathematical notation. On the other hand an expert in formal methods should not be forced to spend much time translating a specification for domain experts.

We believe that a broad acceptance of formal methods needs tools that can mediate between domain experts and formal method experts. The ProB Flash server aims to be such a tool. Using the Flash server it is very easy to create domain specific visualizations that could demonstrate how a B specification behaves. Such an animation can be seen as a kind of prototype for the software. The domain expert can get a feeling what an operation call does and he can check whether his expectations are met.

The ProB Flash server plug-in opens a TCP serversocket and waits for client Flash movies to connect. Client and server can exchange information using XML fragments. (Figure 7) The naive approach to simply create a Flash client for each B specification (maybe using a library) has some major disadvantages. A problem using Flash in dynamic animation is the built in language Action Script. Writing proper Action Script code is error prone; for example take the sourcecode from figure 8. Since every variable that is not explicitly declared as being local will be global, running the example code will lead into an infinite loop.

Another Problem is related to the way text fields are created. The method *MovieClip.createTextField()* can be used to create a new text field but as the sig-

Appendix A

```
<message>
 <type>setup</type>
 <setup>
  <image name="background" x="0" y="0" alpha="100" url="bg.jpg" />
  <image name="lfg" x="0" y="430" alpha="100" url="lsw_down.jpg" />
  <image name="rfg" x="640" y="400" alpha="100" url="rsw_down.jpg" />
 </setup>
</message>
```

Fig. 7. XML-Fragment example

```
function aLoop() {
  for(i=0;i<3;i++) { trace(i); }
}
for(i=0;i<5;i++) { aLoop(); }
```

Fig. 8. Action Script example

nature in figure 9 shows, it does not return a reference to the new text field. The new text field can only be referenced using the instanceName given as parameter. This behavior causes, that only a certain amount of text fields can be used[6]. The signature of *MovieClip.createEmptyMovieClip()* reveals that this method returns a reference to the new instance. We use this fact in our implementation (excerpt shown in figure 10). For each textfield that should be created we instantiate a new movieclip and inside this movieclip a text field named *"label"*. The movies are stored in an array for later reference. Accessing such a field (e.g. to change it's content) can be done by searching the container clip inside the array and using *container.label.text = "new text"*.

```
my_mc.createEmptyMovieClip(instanceName:String, depth:Number) : MovieClip
my_mc.createTextField(instanceName:String, depth:Number, x:Number,
 y:Number, width:Number, height:Number) : Void
```

Fig. 9. Action Script createTextField and createEmptyMovieClip

To hide all these details of Action Script from the specifier we developed a generic animation movie that can be used to animate any B specification. Creating a domain specific visualization thus reduces to

1. Provide pictures or picture sequences that should be used to represent the states or transitions.

[6] dynamic evaluation like eval("name"+number).text = "new text" is not possible in Action Script

Appendix A

```
for (var i = 0; i<labels.length; i++) {
    var text = labels[i].attributes["text"];
    var name = labels[i].attributes["name"];
    var clip = _root.createEmptyMovieClip(name, _root.getNextHighestDepth());
    clip.createTextField("label", 0, 1, 1, width, height);
    clip.label.text = text;
    view_objects.push(clip);
}
```

Fig. 10. Action Script - creating arbitrary number of text fields

2. Write a bit of gluing code that maps state of the B machine to the graphical representation.

An animation can be composed from different parts, therefore it is not necessary to create a single image for each state. Consider a machine that consists of four variables; each variable stores a hexadecimal digit. The machine has 65536 possible states. Since the animation can be composed of four parts and each part needs 16 different pictures we have to provide 64 pictures[7].

The gluing code maps the states and transitions of the specification to the graphical representation. For this we use BeanShell[8], which allows to write interpreted java code that can be modified at runtime. We decided to use BeanShell since it has recently passed the JSR[9] voting process and therefore will be integrated into the java runtime in the future. The code has to contain two methods *statechange()* and *setup()*.

Setup will be executed when either a new client connects or the animation is restarted, it creates a XML message as shown in figure 7 to tell the client about the graphical setup of the animation. The statechange method will be executed each time the state of the model changes. An example for gluing code can be found in figure 11.

We implemented a Java abstraction layer for the Flash components (figure 12). The abstraction layer consists of

1. The *FlashCanvas* is a store for all Flash components, it supports adding and removing of components and committing updates to the clients. FlashCanvas can be seen as the Java counterpart to the root-Flash movie.
2. *AbstractFlashObject* contains methods all other Flash components need to inherit such as update management.
3. *FlashMovie* represents either a single image or an animation. In most cases such an animation consists of an imported picture sequence and the Action Script command *stop()* added to the last frame.
4. *FlashLabel* is the counterpart to a text field.
5. *FlashFont* is used to attach a textstyle to a *FlashLabel*

[7] if we decide to reuse the same set of pictures we only need 16 images at all
[8] available at http://beanshell.org/home.html
[9] See http://jcp.org/en/jsr/detail?id=274

Appendix A

```
void setup() {
    canvas.clear();
    canvas.add(new FlashMovie("background","bg.jpg",0,0));
    canvas.add(new FlashMovie("lfg","lsw_down.jpg",0,430));
    canvas.add(new FlashMovie("rfg","rsw_down.jpg",640,400));
}

void statechange() {
    if (operation.equals("open(gate1)")) { left = "lslt_auf.swf";}
    if (operation.equals("open(gate2)")) { left = "lsrt_auf.swf";}
    if (operation.equals("close(gate1)")) { left = "lslt_zu.swf";}
    if (operation.equals("close(gate2)")) { left = "lsrt_zu.swf";}
    if (operation.equals("flood_lock_right(lock1)")) {  left = "lsw_up.swf"; }
    if (operation.equals("flood_lock_left(lock1)")) { left = "lsw_down.swf"; }
    if (!left.equals("")) canvas.get("lfg").setUrl(left);
    canvas.commitChanges();
}
```

Fig. 11. Excerpt from a gluing code

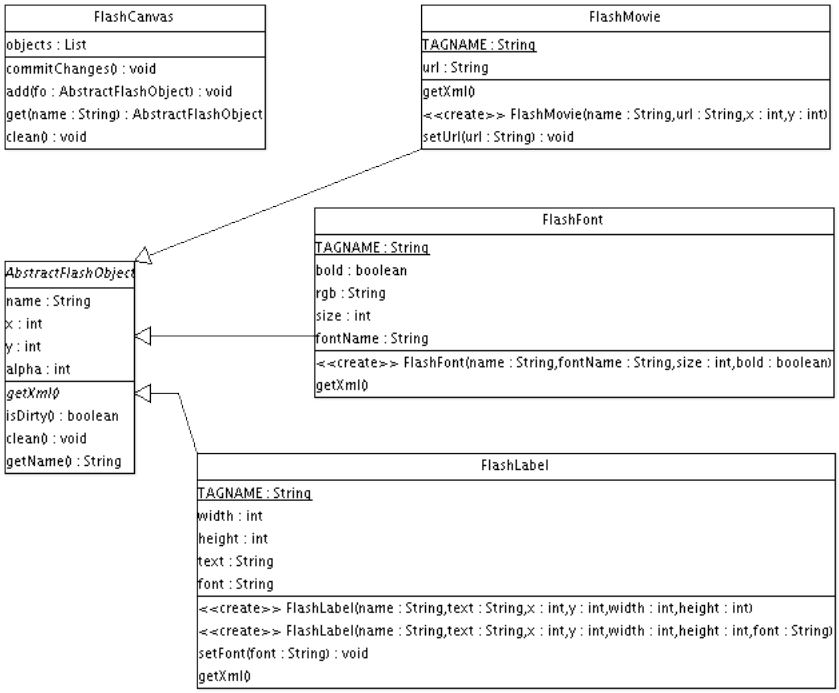


Fig. 12. Flash objects UML

Appendix A

5 Applications

We applied our generic solution to several B specifications. Examples can be seen in figure 13 and 14. Both have animated transitions between their states. The railway network example has been created within three hours during a workshop, most of the time was spent drawing the pictures. It is based on a specification by Michael Butler based on a requirements document from Siemens Transportation Systems. The artwork for the waterlock example is much more sophisticated, but it took only about two days to create the scene setup[10]; excluding the time to render the scene and the animations. Writing the gluing code took less than one hour.

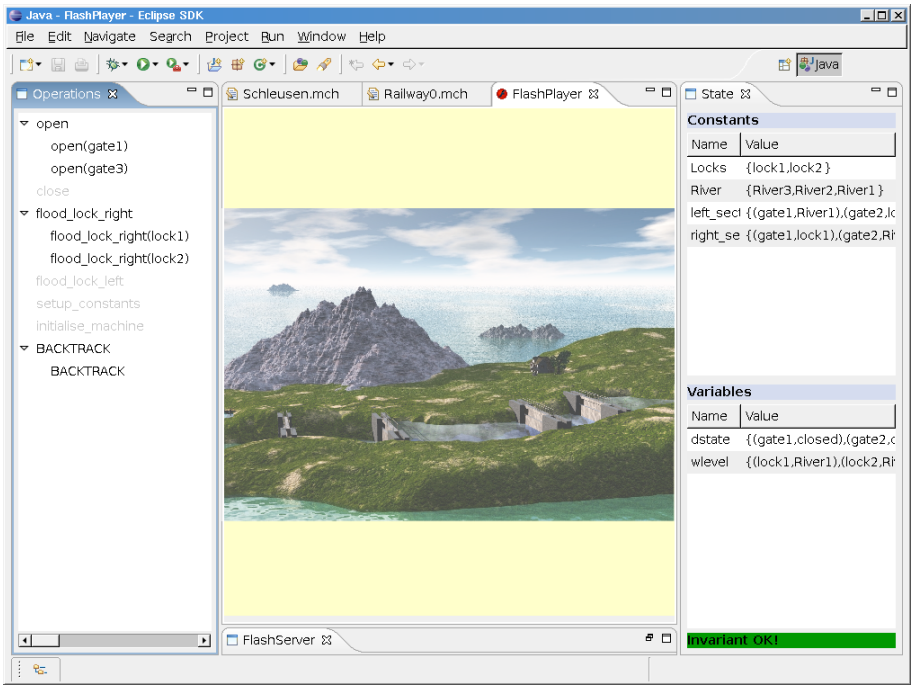


Fig. 13. Visualization of a waterlock system in ProB for Eclipse

6 ProB for Rodin

In a first stage ProB itself was extended to deal with some new features offered by Event-B. Some of the extensions are as follows, mainly ensuring that ProB correctly supports the Event-B models as allowed by B4Free:

[10] with Bryce http://bryce.daz3d.com

Appendix A

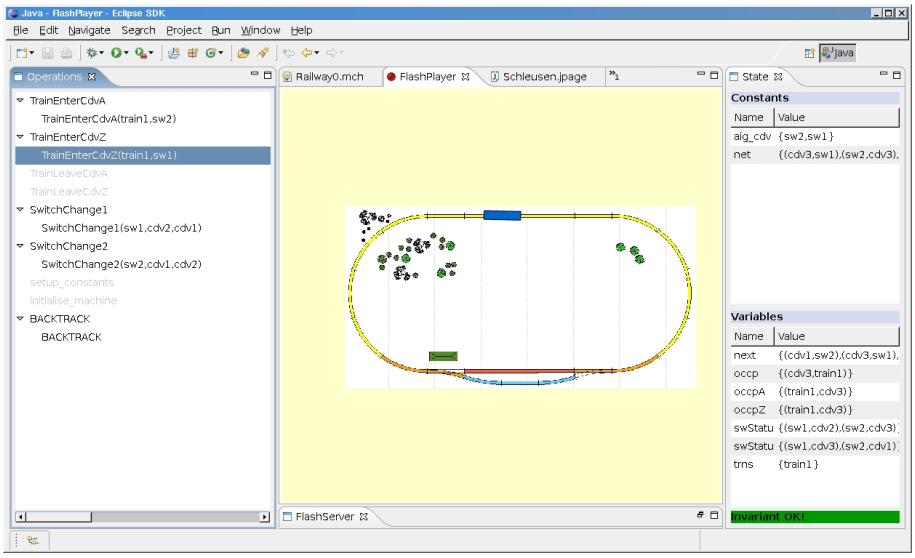


Fig. 14. Visualization of a Railway network in ProB for Eclipse

- Support for some new keywords were added to the tool: e.g., `MODEL`, `AXIOMS`, `EVENTS`, `THEOREMS`.
- The possibility to view values of top-level `ANY` variables was added to the animator. Indeed, Event-B operations themselves take no arguments, but they usually contain of a top-level `ANY` construct. The variables of that construct often act in a similar way to operation parameters in classical B and it is important for the user to be able to easily view the values of the `ANY` parameters.

In the second stage we needed to interact with the Rodin database to extract information about the Event B model. This information is then translated into a B4Free-style Event-B model, which is passed to the ProB for Eclipse animator presented in Section 3. The translation both needs to convert Unicode into ASCII, as well as Event-B constructs into classical B, such as:

- Merging a model with its associated context.
- Deducing which sets in a context are actually enumerated (in Event-B an enumerated set `E={e1,e2,...,en}` is represented as a deferred set E with a set of constants $e_1, e_2, ..., e_n$ and with associated axioms $E = \{e_1, \ldots, e_n\}$, $e_1 \neq e_2, \ldots$).
- New Event-B constructs such as set comprehension with terms or total relations, all need to be translated into equivalent classical B constructs.

A screenshot of the ProB for Rodin plug-in can be seen in Figure 16. The enabled operations can be seen in the pane on the top right; the variables and

Appendix A

constants can be seen in the pane on the bottom right. As can be seen, an invariant violation was detected by the animator. The plug-in can be installed from within the Rodin platform from the update site (see Figure 15):[11]

`http://www.stups.uni-duesseldorf.de/ProB/update/prototype`

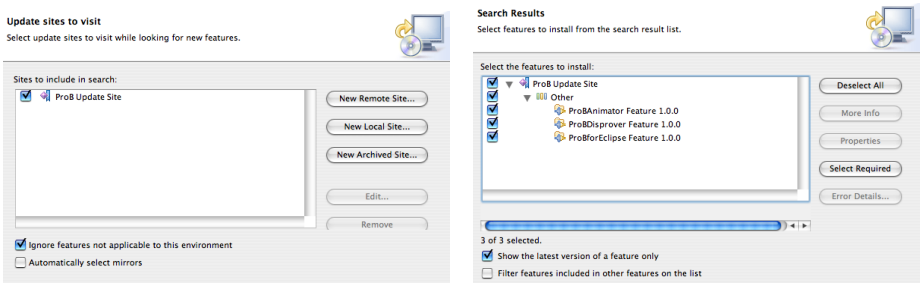


Fig. 15. Installation of ProB for Rodin

7 Related work

The most closely related work on the B side is [5, 3], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. However, the focus of these tools is test-case generation and not verification, and the subset of B that is supported is comparatively smaller (e.g., no set comprehensions or lambda abstractions, constants and properties nor multiple machines are supported). To our knowledge no graphical visualization for states or transitions is available. The company ClearSy is also currently developing a commercial visualization tool for B specifications, also based on Macromedia Flash technology.

There is some more related work on the Z side, such as [15], which presents an animator for Z implemented in Mercury, as well as the Possum animation tool [7]. The latter is probably most related, as it allows the user to write custom TclTk code that can query the state of a Z specification in order to provide a custom graphical visualization. Another animator for Z is ZANS [10]. It has been developed in C++ and unlike ProB only supports deterministic operations (called explicit in [10]). The more recent Jaza tool by Mark Utting looks very promising.

Other related work is the ALLOY analyzer developed by Jackson [8, 9]. ALLOY is a special purpose lightweight object language which does not have the same

[11] At the time of press the PROB command-line tool `probcli` still has to be manually copied into the Rodin "plugins" directory. Hopefully this will no longer be required by the time you read this.

Appendix A

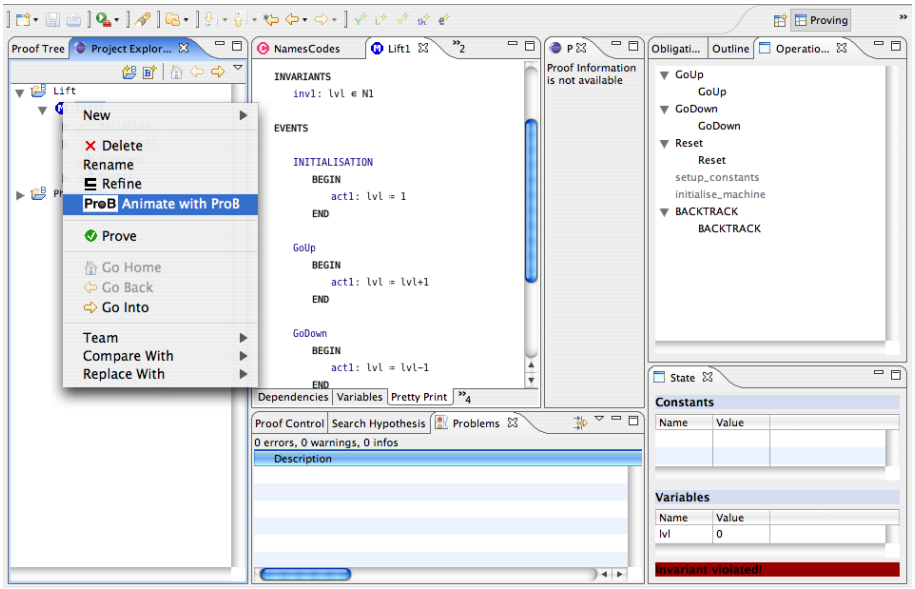


Fig. 16. ProB for Rodin

penetration as B (e.g., ALLOY is a first-order language), but is well suited to constraint checking. The tool uses SAT solvers to find counter-examples in which an operation relates a consistent before state to an inconsistent after state. ALLOY uses GraphViz to visualise the states of a specification; there is, however, no automatic animator for ALLOY.

8 Future work and Conclusion

Several items can be pointed out for the most pressing future work:

1. There are several menu commands that are supported by classical ProB but not yet in the Eclipse version; we work on porting each feature to the new version
2. Writing the gluing code is still an relatively cumbersom task, we are developing a graphical interface to setup the animation and an automatic code generator to generate the code.
3. We are also working on improving the AMN editor such that it supports syntax errors highlighting, code completion, a outline view and code folding.
4. We will extend the abstraction layer for Flash components to enable two-way-communication. Therefore we will support Flash Buttons, this will help to generate prototype user interface prototypes of B machines.
5. We are developing an extension of ProB to animate Z specifications; mainly as some of our formal specifications from industry are written in Z and cannot be easily translated into B.

Appendix A

In summary, we have presented a generic method to visualize B specifications using Flash technology. This method has been implemented within a new version of ProB integrated into the open-source development platform Eclipse and with an improved user-interface. We also support the Rodin platform and the Event-B models developed within. We hope that this new method and tool will help make formal methods more appealing in an industrial setting, notably by allowing domain experts to understand and visualize formal specifications.

References

1. J.-R. Abrial. *The B book : assigning programs to meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in b. In *B*, pages 83–128, 1998.
3. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.
4. U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
5. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P.atoen and P.tevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
6. E. Gamma and K. Beck. *Contributing to Eclipse*. Addison-Wesley, first edition, october 2003.
7. D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. *Automated Software Engineering*, 00:302, 1998.
8. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
9. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, pages 256–290, September 2001.
10. X. Jia. An approach to animating Z specifications. Available at `http://venus.cs.depaul.edu/fm/zans.html`.
11. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
12. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
13. A. Romanovsky. Rigorous open development environment for complex systems - RODIN. *ERCIM News*, 65:40–41, 2006.
14. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.
15. M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–294, Perth, Australia, February 1998.

Appendix B

Debugging Event-B Models using the ProB Disprover Plug-in *

Olivier Ligot[1], Jens Bendisposto[2] and Michael Leuschel[2]

[1] Facultés Universitaires Notre-Dame de la Paix Namur
Namur, Belgium
olivier.ligot@student.fundp.ac.be
[2] Softwaretechnik und Programmiersprachen
Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. The B-method, as well as its offspring Event-B, are both tool-supported formal methods used for the development of computer systems whose correctness is formally proven. However, the more complex the specification becomes, the more proof obligations need to be discharged. While many proof obligations can be discharged automatically by recent tools such as the RODIN platform, a considerable number still have to be proven interactively. This can be either because the required proof is too complicated or because the B model is erroneous. In this paper we describe a disprover plugin for RODIN that utilizes the ProB animator and model checker 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) and she should not attempt to prove the proof obligation. 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.
Keywords: RODIN, ProB, Event-B, B-Method, Autoprover.

1 Introduction

The B-method introduced by J.-R. Abrial [1] is a theory and methodology for formal development of computer systems which is based on the notion of *abstract machines* and *refinement*. B is used in industry, mainly for safety critical applications. It is supported by several industrial strength tools such as AtelierB [21], the B Toolkit [5] or B4Free for proving correctness and code generation and tools for animation, modelchecking and model based testing such as ProB [12] or the BZTT [4].

However, classical B is lacking certain dynamic constraints (temporal logic constraints, liveness constraints) that can be used to model how a system can

* This research is being carried out as part of the EU funded research projects: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

Appendix B

evolve. This shortcoming was one of the reasons to extend B to Event-B [2, 3] which enables the specification of reactive systems without abandoning the notion of refinement.

An Event-B specification is written as an abstract machine that consists of variables which define its state and some events. An event decomposes into a predicate, the guard, that specifies under which circumstances it might occur and some generalized substitutions called actions. For instance, if the state s of an abstract machine is $(x = 2, y = 7)$ and there is an event e with the guard *true* and the action $x := y$, then a successor state of s might be $(x = 7, y = 7)$.

A notable recent development is the EU funded research project IST 511599 RODIN, which aims to develop an open tool platform based on Eclipse that supports Event-B. The objective of RODIN is to create a unified methodology and supporting tools for cost-effective, rigorous development of software systems.

A rigorous software development requires to reason about the correctness of the formal specification. For example, one should verify that an Event-B model does not violate its invariant. Other correctness conditions are related to refinement or the properties associated with constants. The proof obligations that need to be discharged in order to establish correctness can be mechanically extracted from an Event-B model. For instance, one proof obligation will stipulate that the initialisation of an Event-B model must establish the invariant. The RODIN platform comes with a tool, the proof obligation generator, that extracts proof obligations from a model (see Figure 1).

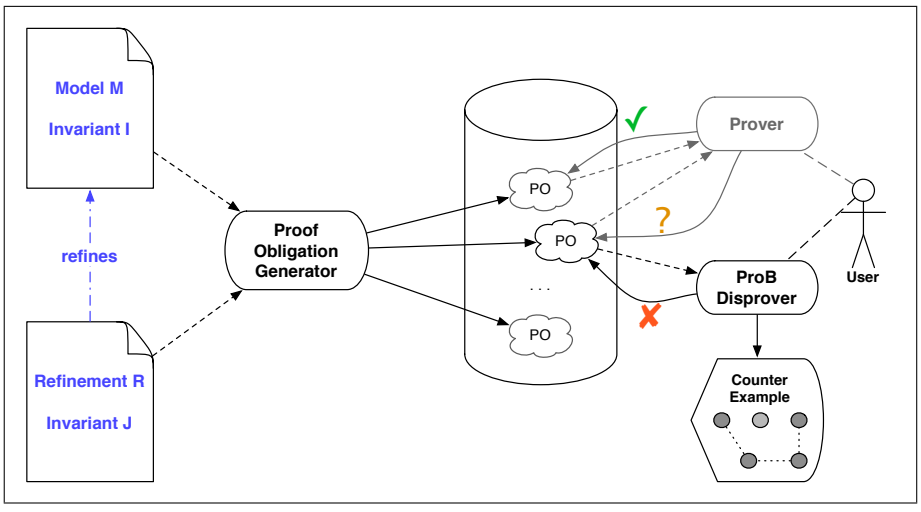


Fig. 1. Overview of proof activities and the role of the disprover

The RODIN platform also comes with some automatic provers, which can discharge a considerable number proof obligations automatically. Obviously, due

Appendix B

to incompleteness, not all proofs can be done automatically. In that circumstance the user is left wondering:

- Is the proof obligation valid and the proof simply too complicated for the automated prover? In other words, should I start up the interactive prover and try to prove the goal manually?
- Or is there a problem within the specification and should I spend time looking for the error and then correct it?

Pursuing either path can lead to considerable wasted effort. In this paper we present a tool which helps the user in this common situation: a disprover which tries to find a counterexample for the particular problematic proof obligation (see also Figure 1).

- If the disprover finds a counterexample we know that it is futile to spend time with the interactive prover. Also, the counterexample will give us a handle on the problem and help us find the error in the specification more quickly.
- If the disprover finds no counterexample, we know that—in certain circumstances at least—the proof obligation seems to be valid. Of course, we are still not sure whether the proof obligation is true in all circumstances; but we have at least gained some additional confidence about its validity.

As an example, suppose we want to prove the theorem, that every finite undirected graph has at least two nodes of the same degree.[3] Our Event-B model will contain the basic set $NODES$ and a graph consists of a set $V \subseteq NODES$ of vertices as well as a symmetric binary relation E representing the edges. The degree of a vertex v is simply $card(\{v\} \lhd E) = card(E[\{v\}])$. A sequent representing our theorem might be something like the following:

$$\begin{array}{c} V \subseteq NODES \land E \in NODES \leftrightarrow NODES \land E = E^{-1} \land card(V) \in \mathbb{N} \\ \Rightarrow \\ \exists x \exists y : x \in V \land y \in V \land x \neq y \land \\ card(\{x\} \lhd E) = card(\{y\} \lhd E) \end{array}$$

However, this theorem is not provable since we made two mistakes in the definition. While it might not be obvious which mistakes we made, the disprover plug-in finds counterexamples that will help us to identify the problems and to correct the theorem. A trivial counterexample the tool finds is the empty graph, another counterexample found by our tool with 5 vertices that contains self loops is shown in Fig. 2. As can be seen, all vertices have a different degree. So we need to strengthen the left side of our implication to disallow self loops and graphs with less than two nodes, after which the disprover can no longer find a counterexample:

[3] This example is inspired by a talk given by Leslie Lamport at B'2007.

Appendix B

$$V \subseteq NODES \land E \in NODES \leftrightarrow NODES \land E = E^{-1} \land card(V) \in \mathbb{N} \land$$
$$card(V) > 1 \land id(NODES) \cap E = \varnothing$$
$$\Rightarrow$$
$$\exists x \exists y : x \in V \land y \in V \land x \neq y \land$$
$$card(\{x\} \lhd E) = card(\{y\} \lhd E)$$

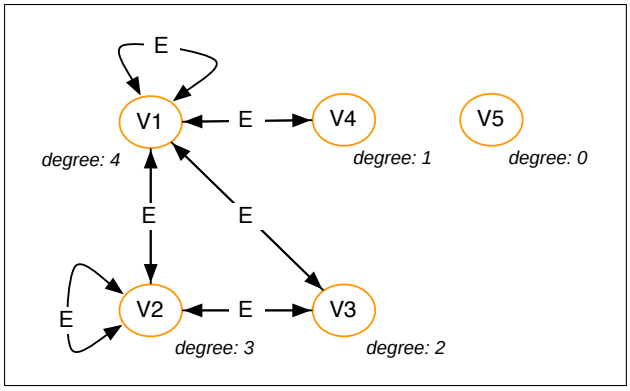


Fig. 2. Counterexample found by the disprover

RODIN can be extended by third parties, in particular it is possible to add external proving tools. We have thus developed a prover plug-in, which works as a disprover. Our plug-in is based on the Prolog based animator and model checker ProB [12]. The ProB animator is fully automatic and does not require the user to guess the right values for the operation arguments or choice variables. The undecidability of animating B is overcome by restricting animation to finite sets and integer ranges, while efficiency is achieved by delaying the enumeration of variables as long as possible. The main idea of our work is to translate an individual proof obligation into a B machine such that the animator can be used to find counterexamples. Of course, one could have used the ProB model checker itself on the whole Event-B model. This is an alternate validation option, but this will "only" find sequences of operations which violate the invariant starting from some valid initialisation; i.e., it will not detect problems if the invariant is too weak (see [12]). Furthermore, by restricting our attention to a single, problematic proof obligation we can increase the likelihood of the disprover finding counterexamples.

The rest of the paper is structured as follows. First we provide some background on proof in Event-B in Section 2. Then we present the underlying methodology of our disprover plug-in in Section 3, before discussing the actual implementation in Section 4. We conclude with remarks on how to use the disprover as a prover and other future work in Section 5.

Appendix B

2 The Event-B proving subsystem

This section gives an introduction to proofs in Event-B. We will also discuss the architecture of the RODIN proving subsystem, its Kernel prover and how external reasoners work.

Proving correctness A proof in Event-B is constructed using (a slight variation of the) sequent calculus [18]. A *sequent* in Event-B is of the form $\Gamma \vdash \Sigma$ where Γ is a finite set of predicates called *hypotheses* and Σ is a single predicate called *goal*. A sequent basically means that the goal should be a logical consequence of the hypotheses Γ. Proofs of sequents are carried out using an *inference rule*. An inference rule contains a finite set of sequents A — the *antecedent* — and a single sequent C — the *consequent*. An inference rule means: if we can prove all sequents within A, then C has also been proven. It is also possible that a rule has the empty set as antecedent, this means that C has been proven. A proof for a sequent can thus be viewed as a finite tree. Each node of this *proof tree* contains a sequent s as well as an inference rule r whose consequent is s. The children of a node are the sequents in the antecedent of its rule r. Leaf nodes are those nodes where the associated inference rule has the empty set as antecedent. To actually discharge a proof obligation *po*, we need to find a finite proof tree whose root node is labelled with *po*.

Note that the antecedent of sequents might contain a subset called *type environment*, where the predicates only carry type information such as *x is an integer* ($x \in \mathbb{Z}$) or *y is a member of a basic set Y* ($y \in Y$). Sequents of the type environment can be statically checked by a type checker and thus have the empty set as antecedent (unless there is a typing error). We call a sequence of inference rules that discharge a certain type of sequents a *proof tactic*. It can be seen as a kind of pattern for proving.

The RODIN proving subsystem In RODIN, a considerable number of proofs can be done automatically by the proving subsystem that consists, as shown in Fig. 3, of the proof obligation generator and the Event-B Kernel Prover [17]. The proof obligation generator extracts all proof obligations from the Event-B model that need to be discharged in order to prove correctness of the model and stores them in a XML database file. After all POs have been generated, the kernel prover tries to discharge valid POs automatically.

As shown in Fig. 3, the Event-B kernel decomposes into the proof manager and a set of prover plugins. While the proof manager is responsible for storage, traversal, composition and reusage of proofs, the prover plugins try to generate valid inferences in order to discharge the proof obligations. The proof manager also maintains the state of current proofs for all proof obligations and decides if they have to be discharged and calls external provers if they are non-interactive. There are also interactive reasoners that require the user to apply them, the ProB disprover plug-in is such an interactive plug-in. In the next section we

Appendix B

show the underlying principles of our plug-in, before showing in Section 4 how it was integrated into the RODIN platform.

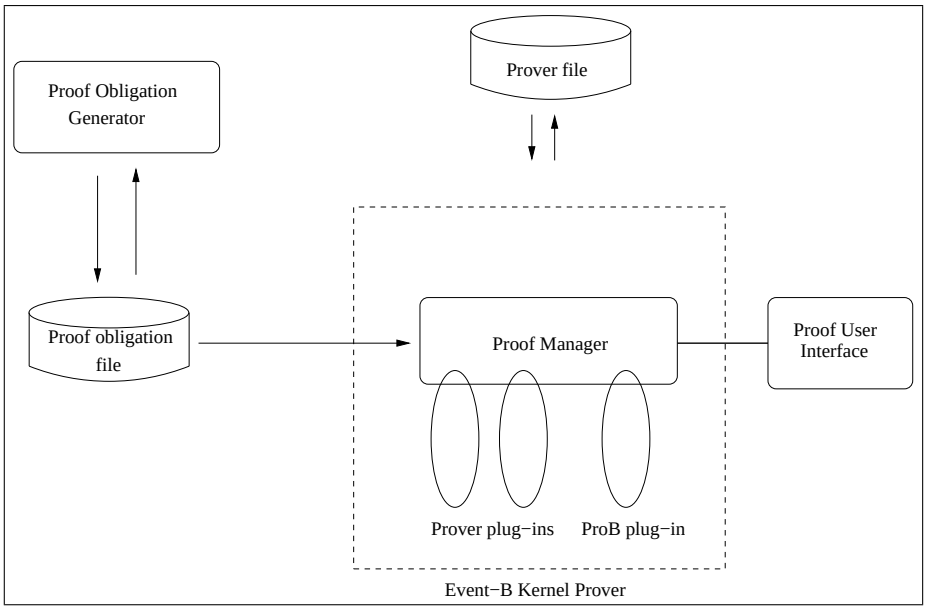


Fig. 3. Architecture of the RODIN proving subsystem

3 The principle of disproving using ProB

In the following, we will explain, how a sequent can be translated into a B machine that can be used with ProB.

Finding counterexamples Let $G(x_1, ..., x_k)$ be the goal of a sequent s and let $H_1(x_1, \ldots, x_k), \ldots, H_n(x_1, \ldots, x_k)$ be the hypotheses. To find a counterexample for s, we need to check if the predicate

$$\exists x_1, ..., x_k : (H_1(x_1, ..., x_k) \wedge ... \wedge H_n(x_1, ..., x_k)) \Longrightarrow \neg G(x_1, ..., x_k) \qquad (1)$$

holds. If it does, then we can extract a concrete counterexample by finding a valuation for $x_1, ..., x_k$ which makes the implication true. Finding values that satisfy a propositional boolean formula is NP complete, and for the first order logic formulas that can occur within sequents, the problem is undecidable. To overcome this difficulty, we have to restrict sets to relatively small, finite domains. As a consequence, we know that in principle it is not possible to guarantee that

Appendix B

a disproving algorithm can automatically find a counterexample if one exists. In other words, the absence of a counterexample does not mean in general, that a proof obligation is valid. (There are, however, certain cases where the absence of a counterexample discharges a proof obligation. We discuss these cases in section 5.)

Transforming sequents into B machines ProB can be used to find a counterexample for a given sequent, but it needs a classical B machine that encodes the sequent as its input. Fortunately this encoding is — at least in principle — not difficult to obtain. We create a B machine, that contains an operation *disproveHypotheses* with the predicate from equation (1) as its guard. The operation is enabled, if and only if ProB can find a counterexample.

In order to construct this stand-alone machine, we need to extract some information from the original Event-B specification such as axioms[4], carrier sets, parameters, variables (including type information) and constants. Furthermore, we need information about the sequent to be (dis-)proved, such as the hypotheses and the goal. The translation of these information is in most cases straightforward, for example we construct the `SETS` clause of the machine by enumerating the set definitions from the original Event-B specification. In some cases the translation is less obvious. For instance, we translate the axioms together with the type information of the constants into the `PROPERTIES` clause. We generate new definitions called `TypeEnvironment` and `Hypotheses` inside the `DEFINITIONS` clause. The `TypeEnvironment` is a subset of the hypotheses that only contains predicates dealing with type information. A schema of the B machine constructed from a given sequent $H_1, H_2, \ldots H_n \vdash G$ is shown in Listing 1.1.

Selecting Hypotheses The RODIN proving subsystem allows the user to select a subset of hypotheses that are in the database, these hypotheses are either directly derived from the specification or previously proven. Obviously if a subset of H proves G, then H also proves G.

$$H' \subseteq H \wedge H' \vdash G \Rightarrow H \vdash G$$

Thus a user can restrict the hypotheses in a sequent to an arbitrary subset of so-called *selected hypotheses*, by removing hypotheses that are of not relevant for the proof. By default, a particular set of hypotheses which deals with the involved variables are automatically selected by RODIN. The user can also decide to hide a particular subset of hypotheses, this subset are called *hidden hypotheses*. In fact, there are thus two alternatives:

- run the external disprover with the *selected* hypotheses or
- run it with *all* hypotheses except the hidden ones.

In any case, the user can choose which alternative to apply (our plug-in provides two buttons) and change his mind later.

[4] An axiom is treated like a sequent $true \vdash A$

Appendix B

Listing 1.1. Schema of an abstract machine constructed from a sequent

```
1   MACHINE Disprove
2   DEFINITIONS
3       TypeEnvironment == H1(x1,...,xk) & ... & Hi(x1,...,xk);
4       Hypotheses == TypeEnvironment &
            Hi+1(x1,...,xk) & ... & Hn(x1,...,xk);
5       Goal == G(x1,...,xk)
6   OPERATIONS
7       disprove(x1,...,xk) =
8           PRE Hypotheses & not(Goal)
9           THEN skip
10          END
11  END
```

4 Implementation of the ProB disprover plug-in

In previous work [6], we have developed a version of ProB that integrates with Eclipse. Its main component is the Eclipse ProB plug-in as shown in Fig. 4. It allows third party tools to use ProB for several tasks, thus it can be seen as a Java abstraction layer for the Prolog part of ProB. The disprover uses this core plug-in to find counterexamples. Therefore it creates — when applied to a node of the proof tree — a B machine as described in Section 3 and starts animating this machine. If the operation `disprove` is enabled, we have found a counterexample.

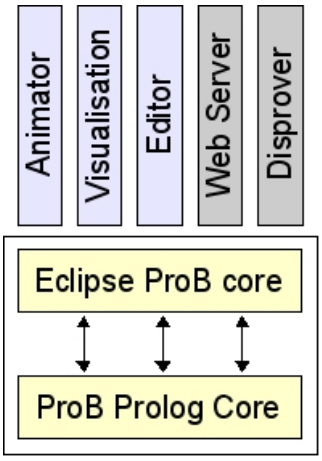


Fig. 4. Architecture of the ProB Eclipse Version

Appendix B

Our disprover plug-in consists of a user interface (UI) that displays the results of a proof and a core component that encapsulates the proof logic. The UI is an extension to the RODIN proving user interface. It allows the user to select a node in the proof tree, that he wants to check with ProB. The core plug-in provides a way to apply the ProB disprover. Its role is to:

- Translate the sequent into a B machine.
- Call ProB through the Eclipse ProB core plug-in.
- Return results to the user interface.
- Handle failures, time outs and user cancel requests.

Displaying counterexamples A first approach was to display counterexamples in a separate window. This solution was however not very useful because there is no connection between the counterexample and the proof obligations. We thus take another approach to resolve this problem by applying a case distinction [1] to the node in the proof tree.[5] As seen in the previous section, a counterexample can be described by a predicate

$$C_p \equiv x_1 = e_1, ..., x_k = e_k$$

Now we apply a case distinction to the node. This results in two child nodes with the sequents

1. $H, C_p \vdash G$
2. $H, \neg C_p \vdash G$

The first sequent is the case where the counterexample was found (C_p makes G false). The second one is the remaining case, where the counterexample is not considered. The user can then repeat the step of applying the disprover plugin to the second predicate to try to find a further counterexample.

Figure 5 shows the tool displaying a counterexample. To launch the external disprover, one has to push the green button.[6] The advantages of this approach are the flexibility of the exploration – by exploring the proof deeper if one wants to find other counterexamples – and the connection with the proof itself.

[5] Original idea by Farhad Mehta.
[6] The red button is for all hypotheses, the green for selected hypotheses.

Appendix B

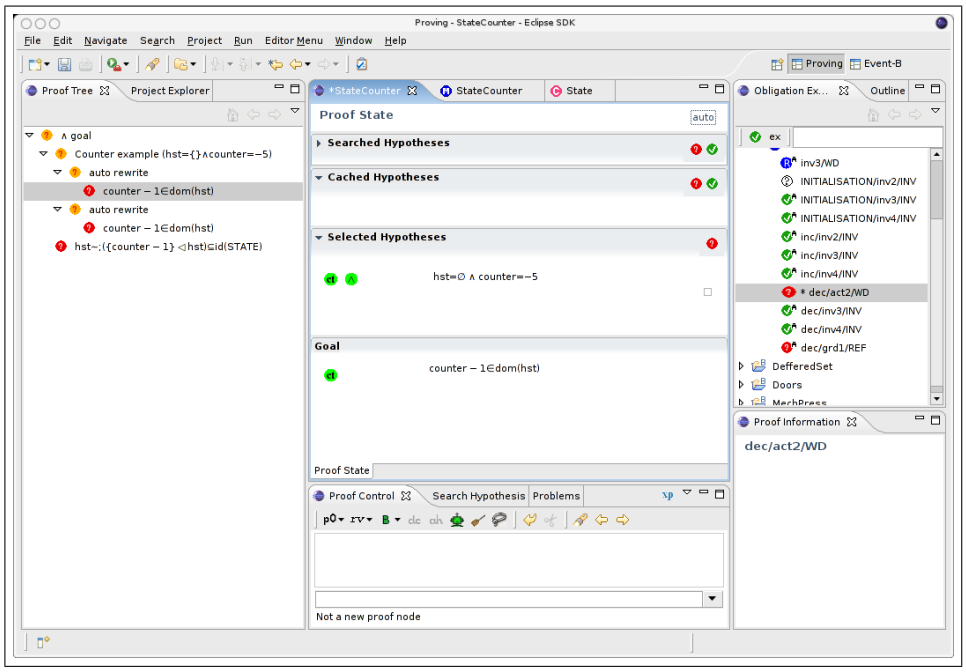


Fig. 5. Screenshot showing the display of a counterexample

5 Future Work and Conclusion

Using ProB **as a prover** If the ProB disprover fails to find a counterexample for a particular proof obligation, we cannot infer that the proof obligation is true. This is due to two reasons:

- **deferred sets:** If a B machine uses deferred sets (i.e., sets which are not explicitly enumerated in the SETS clause), then the cardinality of those sets is not a priori fixed; the set could even be infinite. ProB, however, will check the proof obligation only for some finite cardinalities of the deferred sets, and thus may fail to find existing counterexamples. For example, ProB will fail to find a counterexample for the formula $\exists n.(n : \mathbb{N} \wedge card(S) < n)$, where S is a deferred set without further restrictions.
- **integers:** If an integer variable occurs inside a proof obligation, whose value is *not* determined by the rest of the proof obligation, ProB will enumerate the variable only within a finite interval (between user determined MININT and MAXINT). Again, ProB may thus fail to find counterexamples for integer values which lie outside of MININT..MAXINT.

However, if a B machine contains neither deferred sets nor integer variables, the ProB disprover can actually also be used as a *prover*. This condition can

Appendix B

be easily checked statically, in which case our disprover can inform the Rodin platform that the PO has actually been proven. Some practical B specifications fall into this category. For example, the Volvo vehicle function used in [12]. Another example is a Hamming encoder [8], for which Dominique Cansell has used ProB to prove some essential theorems (which would have been extremely tedious to prove by hand).[7]

In future work, we are planning to implement a static analysis which safely infers intervals for the integer variables. If all variables can be proven to lie within a finite range, ProB could be used as a prover on this larger class of specifications (provided MININT and MAXINT cover all those ranges).

More future work An empiric evaluation of the use of ProB as a disprover is required if one wants to see if the plug-in is efficient or can be more optimized. A number of tests have already be done but more benchmark tests on several operating systems would be welcome.

When using relations or functions in the Event-B, the possible values for the variables of a sequent grows extremely large. For example, given $r : A \leftrightarrow A$, where A has a cardinality of 4, we have $2^{4*4} = 65536$ possibilities for r. Given $x : \mathbb{P}(A \leftrightarrow A)$ we even have $2^{2^{4*4}} = 2^{65536}$ possibilities for x. ProB has to investigate these possibilities in order to search for a counterexample. Symmetry reduction is one way to ease this task, and we plan to check whether we can make use of ProB's recent developments [13, 14] in that area for our disprover plug-in. Another option is to partion the configuration space into several areas, and let different instances of ProB running in parallel take care of the corresponding exploration.

Related work A very popular tool for validating models and finding counterexamples is Alloy [11], which makes use SAT solvers (rather than constraint solving). However, the specification language of Alloy is first-order and thus cannot be applied "out of the box" to Event-B models.

Earlier related work are the model generators FINDER [19] and MACE [15] which can also be used to find counterexamples. The prover Isabelle now also has a quick check function [7], looking randomly for counterexamples. There are many more related works, such as the more recent [20], and even several CADE and IJCAR workshops on disproving have been organized. There is also considerable work on combining model checking [9] with theorem proving in general (e.g., [16, 10]).

Conclusion In summary, we have presented a method to use the existing model checker ProB as a tool for proof support, by trying to find counterexamples for individual proof obligations. We have also discussed under which circumstances the model checker can be used as a prover. We have presented the implementation within Eclipse, using the Rodin Event-B platform and have shown how

[7] Private communication by Dominique Cansell.

Appendix B

this has enabled to use the model checker in a very targeted and convenient way. We believe that a model checker can provide a very valuable support for the B developer, avoiding unnecessary time spent trying to prove a false proof obligation.

References

1. J.-R. Abrial. *The B book : assigning programs to meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial. Extending B without changing it. (For Distributed System). Proc. of 1st Conf. on B Method. pages 169–191, 1996.
3. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 83–128, London, UK, 1998. Springer-Verlag.
4. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.
5. U. B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.
6. J. Bendisposto. Integration of the ProB modelchecker into Eclipse. Bachelor's thesis, July 2006.
7. S. Berghofer and T. Nipkow. Random Testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.
8. D. Cansell, S. Hallerstede, and I. Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Nov 2004. Chapter 16.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.
11. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:256–290, 2002.
12. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
13. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry Reduction for B by Permutation Flooding. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
14. M. Leuschel and T. Massart. Efficient Approximate Verification of B via Symmetry Markers. *Proceedings International Symmetry Conference*, pages –, Januar 2007.
15. W. McCune. MACE 2.0 Reference Manual and Guide. *CoRR*, cs.LO/0106042, 2001.
16. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

Appendix B

17. RODIN Consortium. Rodin deliverable D10 - Specification of basic tools and platform. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf.
18. RODIN Consortium. Rodin deliverable D7 - Event B language. Technical report, 2005. Available at http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf.
19. J. K. Slaney, E. L. Lusk, and W. McCune. SCOTT: Semantically Constrained Otter System Description. In A. Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 764–768. Springer, 1994.
20. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. In *Foundations Of Computer Security Workshop*, 2002.
21. F. Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.