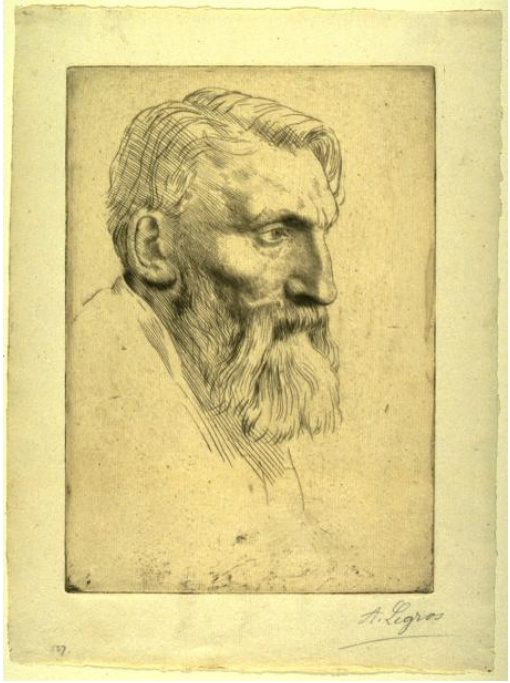
Project IST-511599

RODIN

"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D15

Description of the Rodin Prototype

Editor: *Laurent Voisin (ETH Zurich)*

Public Document

February 28, 2006

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

Contributors

Michael Butler	University of Southampton
Joey Coleman	University of Newcastle
Stefan Hallerstede	ETH Zurich
Thai Son Hoang	ETH Zurich
Farhad Mehta	ETH Zurich
Christophe Métayer	ClearSy
François Terrier	ETH Zurich
Laurent Voisin	ETH Zurich

Contents

1 Introduction

This document describes the contents of Rodin Deliverable D15 *Prototype of basic tools and platform*.

The Rodin prototype is an extensible application for developing event-B models and proving them correct. This prototype is not an industrial strength application, but rather a proof of concept. The aim, when developing it, was to ensure that the specification of the basic tools and platform (see [D10]) is indeed implementable and that the resulting application is usable.

The implementation of this prototype corresponds to the advancement of tasks 3.2 to 3.7 of work package 3 at the prototype level (these tasks are carried out in parallel) and achieves milestone M9 of the project.

After describing the *software architecture* of the prototype, we describe in turn each of its components: the *Rodin platform*, the *event-B core*, and the *event-B user interface*.

2 Architecture of the Prototype

The Rodin prototype is built on top of the Eclipse Interactive Development Environment (IDE) which provides all basic services needed by such an application. The strong advantage of this design decision is that a lot of software gets reused and that the Rodin team didn't have to *reinvent the wheel*, as already argued about in [D5].

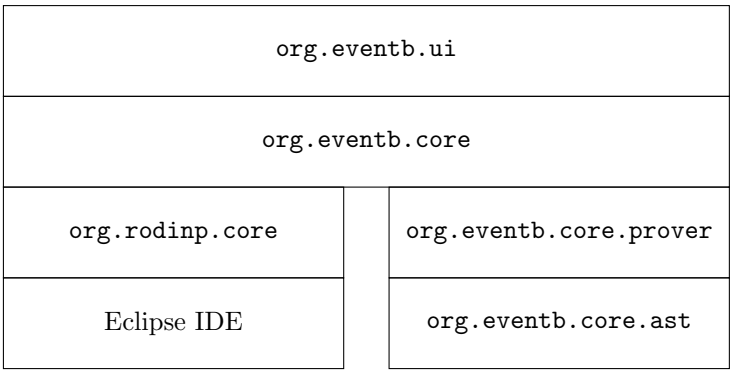
As a consequence, all Rodin software is provided as a set of plugins to the Eclipse IDE. These plugins contain the tools used during modeling and a user interface. These plugins are decomposed along three axes, which are strong design principles of the Rodin project:

- Separation of core and UI plugins: core plugins provide the basic functionality (such as static checking and proof obligation generation) and are completely independent of the user interface, which is isolated in UI plugins.
- Separation of the Rodin platform and its event-B customization: the Rodin platform is a general-purpose modeling framework and is completely independent of the formalism used, while the event-B customization is specially tailored to event-B modeling.
- Independence from Eclipse: if a piece of software doesn't rely on any service provided by the Eclipse IDE, then it should be packaged separately, so that it can be reused in another context.

The Rodin prototype consists of the following plugins:

Rodin Platform (`org.rodinp.core`). This plugin provides the core functionality of the Rodin platform: a database for storing hierarchical models and artifacts produced by tools (such as proof obligations and proofs) and an incremental builder for running tools in a reactive manner.

event-B Core Plugin (`org.eventb.core`). This plugin customizes the Rodin platform for the event-B formalism. It defines all database elements



The Rodin prototype is built by stacking plugins on top of the Eclipse IDE.

Figure 1: Architecture of the Rodin Prototype

needed for supporting event-B modeling and it contains the three tools needed by event-B: static checker, proof obligation generator and prover.

event-B Abstract Syntax Tree (`org.eventb.core.ast`). This plugin contains a library for manipulating event-B mathematical formulas in the form of Abstract Syntax Trees. It provides all basic services for parsing, pretty-printing, type-checking, traversing, transforming, etc. mathematical formulas. This library is put in a separate plugin because it is independent of Eclipse.

event-B Sequent Prover (`org.eventb.core.seqprover`). This plugin contains a library for proving sequents. This library is the core of the event-B prover, without any dependency on Eclipse, hence a separate plugin.

event-B User Interface (`org.eventb.ui`). This plugin contributes the two graphical user interfaces of the event-B platform: the modeling UI and the proving UI.

These plugins are organized as depicted in Fig. 1, where box stacking denotes dependency: each plugins depends on the plugins drawn below it.

The sequel of this document describes more precisely the contents of the Rodin plugins.

3 The Rodin Platform

The Rodin platform is the formalism-agnostic part of the prototype. Contrary to the other parts, it's completely independent of event-B and can be reused for any other formalism, e.g., π-calculus or CSP.

This plugin has been developed because the basic functionality of the Eclipse IDE didn't fulfill exactly some of the requirements that we had for a fully-fledged modeling platform.

Firstly, Eclipse provides a service for managing projects, folders (i.e., directories) and files, independently of the underlying operating system. However, this service, implemented in the *Resource* plugin, provides only a flat view on files, while we wanted to have a hierarchical view on them, as discussed in [D5]. As a consequence, we developed a database on top of the Resource plugin.

Secondly, the Resource plugin of Eclipse also provides a mechanism for incrementally building a project, that is deriving automatically and in the background some *object* files from source files, as needed traditionally in software development (compilation process). However, this mechanism doesn't allow us to implement directly the scheduling policy depicted in [D10]. Moreover, this mechanism is quite crude and implementers need to implement the full logic of their builder from scratch. As a consequence, we developed an incremental builder that can run a tool-chain (static checking, proof obligation generation and automatic proving) with our scheduling policy.

The rest of this section gives an overview of the implementation of the Rodin database and the Rodin builder.

3.1 The Rodin Database

The Rodin database of the prototype implements parts of its specification given in [D10]:

- Most kinds of elements (database, project, file, and internal elements) are implemented. Only folder elements are missing in the prototype (they will be implemented in the final version). File and internal element types can be defined as extensions, using the extension points `fileElementTypes` and `internalElementTypes`.
- The persistence of elements is implemented, including the caching of internal elements in memory.
- Both low-level and high-level services are fully implemented, including the observer design pattern on database changes.
- The undo/redo facility is not yet implemented.
- The compare and search facilities are not yet implemented.

The Rodin database is implemented by four packages:

`org.rodinp.core` contains the published API of the database, that is a set of classes and interfaces visible to all clients.

`org.rodinp.core.basis` contains the semi-published API of the database: It is a set of abstract classes, which are intended to be sub-classed by clients that contribute new element types to the database. However, no other use of these classes should be made, like calling some internal methods exposed by the abstract classes or overriding them.

`org.rodinp.internal.core` contains the implementation of the database manager. Most notably, there are two classes for each kind of element type (project, file, internal, etc.) The first class implements a handle on the element, that is the client-visible part of the database, while the second

class, suffixed by `ElementInfo`, implements the actual element, that is the list of children and attributes.

All modifications on the database are encapsulated in operation objects, whose class inherits from `RodinDBOperation`. Class `DeltaProcessor` listens to changes of Eclipse resources and ensures that the database is kept in sync with the filesystem. Finally, class `RodinDBManager` implements the database cache and the Observer design pattern for the database.

`org.rodinp.internal.core.util` contains various utility classes for logging, string manipulation, messages loading and cache management.

3.2 The Rodin Builder

The Rodin builder implements an `IncrementalProjectBuilder` of the Eclipse Platform. In Eclipse, each project is associated with its own builder(s). If a project is associated with the Rodin nature, the Rodin builder is called when resource changes have taken place in the project. The Rodin nature is attached to a project by the Rodin database.

The Rodin builder is extensible. One can contribute tools by means of the extension point `autoTools`. The contributed tools are run when a Rodin file is updated. The extractors are invoked after a Rodin file has been changed or created, i.e., after a tool has finished for the file. Extractors are usually implemented together with a tool to dynamically extract dependencies concerning the files that are an input of the tool and insert them into the dependency graph of the Rodin builder.

The Rodin builder is implemented by two packages:

`org.rodinp.core.builder` This package contains the classes that are used by extensions of the builder to add tools to the Rodin platform. Usually adding a tool requires adding the tool itself for running the tool and cleaning up resources generated, and an extractor to tell the Rodin builder about resource dependencies of the tool. Dependencies of a file in a Rodin project are represented in a graph structure that is updated by extractors.

`org.rodinp.internal.core.builder` This package contains the implementation of the Rodin builder. It schedules the tools that have been added to the Rodin builder by its extensions. The schedule is computed according to the dependency graph maintained dynamically for each project. The computed schedule is a topologically sorted list of all Rodin file resources in a project.

4 The event-B Core

This section describes the contents of the event-B core plugins. We first describe the customization of the Rodin database for event-B, then each of the core tools for event-B: the static checker, proof obligation generator and proof obligation manager. Finally, we present the two separate plugins: the AST library and the sequent prover.

The architecture of the prover tool has been refined since the writing of [D10]. It is now decomposed as follows:

- The *Proof Obligation Manager* (POM) merges new proof obligations produced by the POG with old proofs, and allows the commencement of a new proof (or resuming an old proof) on a given proof obligation. It thus bridges the database world with the proving world. It is implemented in plugin `org.eventb.core`.
- The *Sequent Prover* works on a given sequent and is decomposed in two parts: the *Proof Manager* (PM) takes care of the proof tree and ensures its correctness, while *prover extensions* provide the actual proving tactics and reasoners. The sequent prover is implemented in plugin `org.eventb.core.prover`.

4.1 Database Customization

The event-B core plugin contributes several element types to the database. These types are split in four groups:

Unchecked model These types are used for representing the components (machines and contexts) entered by the user. There names don't have any prefix, contrary to the element types of the other groups. Example of such elements are `machine` `constant`, `variable`, `invariant`, `event`, ...

Checked model These types are used for representing the checked components produced by the static checker. Their names are prefixed by `sc`. Example of such elements are `scMachine`, `scConstant`, `scVariable`, `scEvent`, ...

Proof obligations These types are used for representing the proof obligations produced by the POG. Their names are prefixed by `po`. Example of such elements are `poFile` `poIdentifier`, `poPredicate`, ...

Proofs These types are used for representing the proofs managed by the prover. Their names are prefixed by `pr`. Example of such elements are `PRFile`, `PRSequent`, `PRStatus` ...

Similar to the database, the published API of the event-B core plugin is in package `org.eventb.core`, while the actual implementations of element handles are in package `org.eventb.core.basis`.

4.2 The event-B Static Checker

The event-B static checker contributes two tools to the Rodin builder. One tool to statically check event-B contexts, and the other one to statically check event-B machines. The implementations of both are very similar and in accordance with the specification written in event-B itself [D10]. They are located in package `org.eventb.internal.core.protosc`.

The event-B context static checker is implemented in class `ContextSC`, which also implements the Rodin builder interfaces for running, cleaning, and extracting. The same functionality is implemented by `MachineSC` for event-B machines.

The static checker filters all elements that are not well-formed or well-typed, and produces appropriate markers with error messages. The input of the static checker are event-B contexts and event-B machines, interfaces

IContext and IMachine in package org.eventb.core. The static checker generates Rodin files containing those elements that are well-formed and well-typed. The static checker extends the Rodin database with elements for the generated files, ISCContext and ISCMachine contained in org.eventb.core.

The present implementation of the static checker is not extensible but has been implemented with extensibility in mind. To this end, it is implemented by a collection of independent rules. However, after inspecting more thoroughly the design, it happens that having independent rules doesn't allow to implement extensions easily and efficiently. A new investigation is taking place to find a better design for the static checker.

4.3 The event-B Proof Obligation Generator

The implementation of the event-B proof obligation generator (POG) follows the specification given in [D10]. It extends the Rodin builder with two tools: one POG for event-B contexts and one POG for machines. They are located in package org.eventb.internal.core.protopog.

The event-B context POG is implemented in class ContextPOG, which also implements the Rodin builder interfaces for running, cleaning, and extracting. The same functionality is implemented by MachinePOG for event-B machines.

The POG does not produce any error messages addressed to the user, i.e., it does not produce markers. All error conditions have been filtered by the event-B static checker that is required to be run before the POG. The extractors of the POG ensure this by adding corresponding dependencies to the Rodin builder. The POG uses the statically checked files (ISCContext and ISCMachine) as input and computes a Rodin file with proof obligations, extending the Rodin database by IPOFile contained in org.eventb.core.

The POG is implemented as a list of rules that usually group some related rule of the specification [D10] for better efficiency. The POG has been designed to be extensible but this is not yet available. The present design suffers from a similar deficiency like the event-B static checker.

4.4 The event-B Proof Obligation Manager

The proof obligation manager is implemented in package org.eventb.core.pom in the Eclipse plugin org.eventb.core. It reads the proof obligation file of a given event-B component and maintains it's corresponding *prover file*. Both these files are managed by the Rodin database.

Functionality. The proof obligation manager starts by reading a proof obligation file from the Rodin database. If a prover file for it already exists, it compares its proof obligations to see if the stored proofs are still valid. If no proof file exists, it generates a new proof file with empty proofs and checks it into the database. For the prototype old proof files are not taken into account, and the tool always creates a fresh prover file on every proof obligation file change.

The proof obligation manager manages proofs at the event-B file level. It can navigate between proofs for proof obligations for a single machine or context. It can make calls to the event-B sequent prover in order to try to discharge a proof obligation automatically using an automated prover, proof reuse, or any

such method provided by the event-B sequent prover. Its output prover file is the starting point for an interactive proving session.

Along with the proof (stored as a proof tree), the proof obligation manager maintains the status of proofs (pending or discharged) of all proof obligations in a proof obligation file. This status is used to give the user a quick overview of progress in the proof of his model.

As proofs are precious artifacts, before closing a prover file, or at any time in the middle of a proving session, the proof obligation manager can save the current proof trees and proof status to the prover file for persistence and later reuse. For the prototype, proof trees are not stored in the prover file.

The Prover File. Each prover file is associated to a proof obligation file and has the suffix `.pr`. The prover file records the current state of all proofs corresponding to proof obligations. For each proof obligation, the prover file contains:

- its name in the proof obligation file,
- a copy of the proof obligation,
- its current proof attempt as a proof tree,
- the status of this proof (pending or discharged).

The copy of the proof obligation and its name is stored in the prover file in order to check if the proof obligation has changed in the course of a formal development. The proof obligation manager can thus check if a proof is still valid. The current proof attempt is stored as a proof tree in order to keep the proof persistent, allowing the user to replay a proof attempt and come back to where he left off in a proof. The status of a proof is also kept track of in order to avoid replaying proofs unnecessarily in order to recompute this information.

As already mentioned, for the prototype the prover file will not store proof trees, but only the status of a given proof attempt. Incomplete proof attempts will have to be restarted from scratch.

4.5 The AST library

The AST library provides basic services for manipulating formulas in the event-B mathematical languages. The main design principles that gave rise to that library are simplicity, efficiency and extensibility, although the latest one has not been yet exercised within the Rodin prototype (no extension has been designed yet).

The core of this library is the AST datastructure where formulas are represented internally as a tree of nodes. All nodes are immutable so that references to them can be shared.

The services provided by that library are the following:

- *Parsing* a formula, that is computing its AST from a string of characters (typically entered by the end-user).
- *Pretty-printing* a formula, which is exactly the inverse of parsing.
- *Constructing* new formulas directly using the library API.

- *Type Checking* formulas, that is inferring the types of the expressions occurring within and decorating them with their type.
- *Testing* formulas for equality (up to alpha-conversion of bound identifiers), for strict equality (taking into account the names of bound identifiers), for well-formedness, etc.
- *Substituting* both free and bound identifiers of a formula, producing a new formula.
- Computing the *well-definedness predicate* of a formula.
- *Navigating* through formulas using the *Visitor* design-pattern.
- etc.

4.6 The Sequent Prover

The sequent prover is implemented in plugin `org.eventb.core.prover`. It is concerned purely with proofs of sequents expressed in the event-B mathematical language. It is independent of the Rodin database or event-B files.

4.6.1 Plugin Dependencies

The sequent prover depends solely on the event-B AST library. In particular, it uses implementations of predicates, expressions, type environments from this library. The sequent prover uses parsing, pretty printing, constructor, and destructor methods for these data types.

4.6.2 Functionality

The sequent prover implements the data structures required for proof using the building blocks provided by the AST library. Its main contributions to the rest of the tool are proof trees (implementing the interface `IProofTree`) and tactics (implementing the interface `ITactic`) used to modify them. Unless otherwise stated, all data structures are immutable. The implementation is done bottom up:

Hypotheses. A hypothesis (interface `IHypothesis`) is a container for predicates that are used as hypotheses in a sequent. It may also contain other information such as the origin of the hypothesis, or some pre-processing that may come in handy at the time of proof.

Sequents. Prover sequents (interface `IProverSequent`) consist of a set of hypotheses, a goal, and a type environment. In a prover sequent, the set of hypotheses is partitioned into selected and un-selected hypotheses, and independently into hidden and visible hypotheses. Roughly speaking, a selected hypothesis is an hypothesis which is considered, either by the end-user or a tool, as relevant to the proof to be achieved, while a hidden hypothesis is an hypothesis that the user wants to hide from automatic tools (i.e., a hidden hypothesis is invisible to automatic tools).

Rules. Inference rules (interface `IRule`) implement backward style inference rules on prover sequents. Application of a rule on a prover sequent (considered the conclusion of the rule) results in the generation of a list of prover sequents (the list of antecedents of the rule). A rule discharges a sequent when its application returns an empty list of antecedents. To keep their implementations straightforward and less error-prone, rules typically perform simple sequent manipulations.

Proof Trees. Proofs in progress are stored as proof trees. Proof trees (interface `IProofTree`) are trees with nodes containing sequents (`IProverSequent`). Nodes with children also contain a reference to the inference rule of type `IRule` used to compute their children. A proof tree node can either be an:

- open leaf node : no rule applied, extendable
- internal node : rule applied, at least one child
- discharged leaf node : rule applied, no children

The root of a proof tree contains the sequent to be proven. The open leaf nodes of a proof tree are the pending subgoals left to be proven. A proof tree is considered to be discharged when it contains no more open leaf nodes. Discharged proof trees correspond to completed proofs.

The only way to construct a new proof tree is by constructing an open leaf node from a prover sequent. The only ways to modify a proof tree are:

- To prune the children of an internal or discharged leaf node, making it open.
- To apply a rule to an open leaf node, making it internal or discharged.

The sequent field of a proof tree is immutable, whereas the rule and children fields are mutable, but only in the above two ways. The root sequent of a proof tree cannot therefore be modified. Also, once we have a discharged proof tree we are sure that only valid rule applications have been used.

Tactics. The growth of proof trees is restricted to applying rules on open leaf nodes. Applying rules individually can be a tedious process since rules typically perform small steps in a proof. A user typically wants to perform a series of rule applications in one shot. Tactics (interface `ITactic`) facilitate this.

Tactics are proof tree transformers. Applying a tactic on a proof tree modifies the proof tree in a desired way. What is important to note is that :

- Tactic application leaves the root sequent of the proof tree untouched. This is ensured by the immutable nature of sequents in proof tree nodes.
- Application of tactics in the end always get translated into pruning or applying rules on proof tree nodes. Tactics still have to go through the proof tree API. What is achieved is greater convenience without compromising the integrity of the proof tree data structure.

The sequent prover provides a number of pre-written tactics (static methods `conjI()`, `hyp()`, `cut()`, `doCase()`, etc. in the class `Tactics`) to be used in proofs.

The sequent prover also provides ways to compose pre-existing tactics to easily construct new ones. For instance, the tactic `norm` is constructed by combining simpler tactics in the following way:

```
public static ITactic norm(){
  ITactic Ti = repeat(compose(conjI(),impI(),allI()));
  ITactic T = repeat(compose(hyp(),trivial(),Ti()));
  return repeat(onAllPending(T));
}
```

The methods `repeat()`, `compose()`, and `onAllPending()` take tactics as input and return tactics as output.

Tactics may be used in automated, or interactive proof sessions. Tactics provide a powerful, uniform and safe way to modify proof trees independent of internal implementation details of the sequent prover.

Tactics intended for interactive use come with methods to check their applicability to a particular goal or hypothesis. These methods are used by the proving user interface to give the user a choice of tactics to apply.

New tactics can be added to the prover when bundled as prover extensions. Although not present in the prototype, the concept of prover extensions will be described in the next section.

4.6.3 Prover Extensions

The sequent prover will provide extension points for prover extensions. A prover extension extends the reasoning capabilities of the prover. It typically may contain:

- new prover tactics,
- external reasoners.

From the users perspective, the main aim of a prover extension is to provide *new tactics* for use within a proof. Internally, a prover extension may also provide *external reasoners*.

New Tactics. As seen earlier, the sequent prover provides ways to compose pre-existing tactics to construct new ones. Developers can create their own specialized tactics for particular types of problems. In case a tactic needs to be called interactively with user input, an extension to the interactive proof UI is also needed.

Providing new tactics in this way does not compromise the safety of the system since only pre-existing reasoning tools are used. It may be the case that a developer wants to add a new reasoning method to the prover, in which case he needs to provide an external reasoner as part of the prover extension and provide tactic wrappers for it.

External Reasoners. An external reasoner extends the reasoning capabilities of the prover. The interface for external reasoners is contained in the file `IExternalReasoner`. Its main task is to generate valid sequents (of a given form) for the prover to use in proofs. A special rule (essentially performing modus ponens) is present in the prover in order to use this generated sequent in a proof.

To check the validity of theses generated sequents is the responsibility of the external reasoner. A faulty external reasoner can make the prover unsound. It is therefore advisable to write external reasoners as a last resort, only when writing new tactics for the desired problem is not feasible.

The sequent prover provided in the prototype does not offer extension points for prover extensions but has been designed in order to accommodate this in the near future.

5 The event-B User Interface

The Rodin prototype provides two UIs for event-B: the Modeling UI and the Proving UI. They are extensions of the Eclipse Platform UI, and are built on top of a workbench that provides the overall structure and presents an extensible UI to the user. The two UIs are implemented in one plugin named `org.eventb.ui`.

The Modeling UI is built on top of the Rodin database, customized by the event-B core plugin. Users create and modify components through the Modeling UI. Once saved and thanks to the Rodin builder, components get checked by the static checker. The static checker provides feedback (error messages, warnings, etc.) to the Modeling UI.

Users interact with the Prover using the Proving UI for discharging proof obligations.

Sec. 5.1 gives the details for the Modeling UI. Sec. 5.2 on page 17 describes the design of the Proving UI.

5.1 The Modeling UI

A snapshot of the Modeling UI can be seen in Fig. 2 on the next page. This is derived from the classical Eclipse Platform UI. It contains five distinct parts:

"Project Explorer" shows a tree structured view of the current workspace, i.e.,list of projects with their editing contents (machines, contexts, etc.)

"Event-B Editor" is a specific editor for creating and modifying event-B components.

"Content Outline" shows the tree structured view of the currently edited component.

"Message Area" displays error/warning messages.

"Wizards" simplifies the creation of a new project or a new event-B component.

The following sub-sections discuss each part in details.

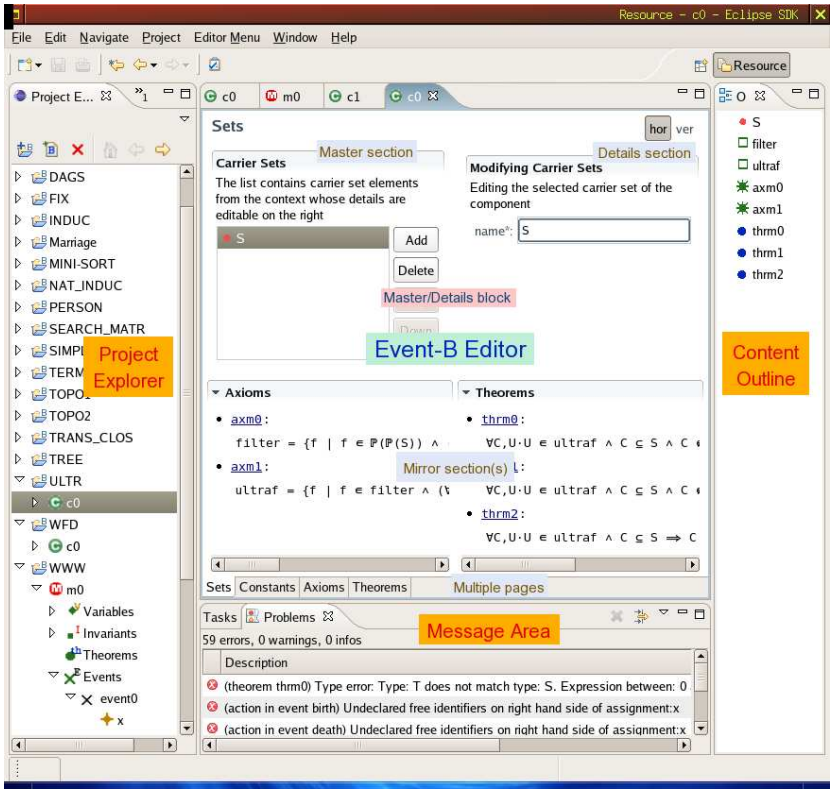


Figure 2: Modeling UI Workbench

5.1.1 The *Project Explorer*

The *Project Explorer* contains a tree-structured view of the list of all *projects* which are developed in the current workspace. This is built on top of the "Rodin Database". This is an extension of the Eclipse "View". The tree contained in the view also shows the contents of the projects, i.e.,the components such as machines and contexts, together with their internal elements, such as variables, invariants, etc. This view is not exactly the same as that of the *Rodin Database*, but it is modified for better navigation on components.

The *Project Explorer* is implemented by the following classes in package `org.eventb.internal.ui.projectexplorer`:

`ProjectExplorer` The main class for the *Project Explorer*.

`ProjectExplorerActionGroup` Implements the actions available in the *Project Explorer*.

`ProjectExplorerContentProvider` Implements the connection with the *Rodin Database* in order to get the relevant data and feed this to the *Project Explorer*'s tree viewer.

The following classes provide some support for the *Project Explorer*:

`TreeNode` Implements nodes which are displayed in the *Project Explorer*. They are not contained in the *Rodin Database* and purely used for displaying purpose of the *Project Explorer*. For example, a"Variables" node which is displayed as children of a machine and contains all variable children of this machine.

5.1.2 The *Event-B Editor*

In a classical Eclipse Platform UI, the Editor region is displayed in the middle of the workbench. The *Event-B Editor* is used for editing event-B related components, such as machines and contexts. All instances of the *Event-B Editor* will be displayed in the Editor region. The *Event-B Editor* is a "multi-page", "form" editor.

Firstly, the *Event-B Editor* as a *form* editor is in contrast with a normal text editor. Users edit an event-B component by typing various information into different forms. Users do not see the actual sequential "source file" any more. This is specifically built to give users a clean and dynamic way for interacting with the *Rodin Database* and does not have to take care of keywords, etc.

Secondly, the *Event-B Editor* is a *multi-page* editor in a sense that there are several different pages for editing different "parts" of a component. For example, when editing a machine, the editor provides five pages, namely: *Dependencies* (for choosing a context on which this machine depends), *Variables*, *Invariants*, *Theorems* and *Events*. Similarly, for a context, there are four pages for editing *Carrier Sets*, *Constants*, *Axioms* and *Theorems*. Each page of the editor can be decomposed into two areas. The top area is the proper editing area where elements can be edited, while the bottom area contains "mirror" sections that display additional information relevant to the edited elements. For instance, in the *Sets* page, the top area displays the sets of the current context, while the bottom area displays the axioms and theorems of the same context.

With the design of *Event-B Editor* (i.e. *multi-page* and *form*), users only need to enter some minimal text into the editor. A typical example of using this editor is that a user goes through different pages, using buttons to add or delete some elements, using forms to change some elements. After entering, modifying or removing a (usually small) number of such elements, the user can "save" the component and the Rodin builder (running in the background) will start appropriate processes for static checking, proof obligation generating and even (auto) proving the component.

The *Event-B Editor* is implemented by the following classes in package `org.eventb.internal.ui.eventbeditor`:

`EventBEditor` The main class implementing the *Event-B Editor*.

`EventBEditorContributor` The contributor class of the Editor, which manages the installation/de-installation of global actions.

The pages of the *Event-B Editor* are implemented by the classes in package `org.eventb.ui.internal.eventbeditor`. Except for the *Dependencies* Page, the other pages are instances of `EventBFormPage`, which contains a "Master/Details Block" at the top, and several "mirror" sections for displaying relevant information at the bottom of the page.

A master section of a *Master/Details block* can be either a section with a table viewer (e.g. in *Variables* or *Invariants* Page), or a section with a tree viewer (e.g. in *Events* Page). Elements are created/removed through the Master section.

A detail section of the a *Master/Details block* contains several editing rows for modifying (i.e.,changing name, content) the element which is selected in the corresponding Master section.

In package `org.eventb.ui.internal.eventbeditor`, the conventions for the supporting classes are:

`..MasterDetailsBlock` implements a *Master/Details block*.

`..MasterSection` implements a *Master* section of a block.

`..DetailsSection` implements a *Details* section of a block.

`..MirrorSection` implements a *Mirror* section, which displays the related information in the editing pages.

`..Page` implements a page of the *Event-B Editor*.

5.1.3 The *Content Outline*

The tree structure (i.e., contents) of the currently edited component is shown in the *Content Outline* section (on the right) of the Eclipse Workbench. The *Content Outline* provides a convenient way to navigate through the editing component. By selecting one or several elements in the *Content Outline*, the corresponding *Event-B Editor* shows the *page* for editing these elements. The *Content Outline* is implemented by class `EventBContentOutlinePage` in package `org.eventb.internal.ui.eventbeditor`.

5.1.4 The *Message Area*

The *Message Area* displays the error and warning messages that have been produced by building tools, such as the static checker. This contains the standard Problems View, in which the different markers (Problem, Warning) are displayed.

Since there are some limitations in the Problems View, we may provide our Rodin Problems View later —with similar concepts— which will be tailored to suit our Rodin Markers.

5.1.5 *Wizards*

Besides three Views and an Editor, the Modeling UI also provides a set of wizards for creating new elements. There are wizards for creating a new project and for creating a new component (machine, context). They can be invoked from the menu File/New, or through the local menu/toolbar of the *Project Explorer*. These wizards are implemented in package `org.eventb.internal.ui.wizards` with the following conventions:

`New..Wizard` The main class for the wizard.

`New..WizardPage` Implements a dialog which will be used when the wizard is invoked, in order to create a new element.

5.2 The Proving UI

The (Interactive) Proving User Interface is under development and it also follows the classical Eclipse Platform UI (Fig. 3 on the following page) with a slight change. It has five distinct parts:

"Obligation Explorer" The *Obligation Explorer* shows a tree structure view of the workspace from projects to proof obligations in components.

"Proof State" This is an editor which shows the current state of the proof.

"Proof Information" This view shows the information related to the current proof obligation the user is working on.

"Proof Tree" This shows the proof tree of the current proof.

"Proof Control" The user can drive the proof and browse proof obligations using this view.

The following sub-sections present each part in details.

5.2.1 The *Obligation Explorer*

Similar to the *Project Explorer*, the *Obligation Explorer* displays a tree structured view of the list of all *projects* which are developed in the current workspace. The tree contained in the view shows the proof related contents of the projects, i.e.,the components such as machines and contexts, together with their proof obligations. This view is not exactly the same as that of the *Rodin Database*, but it is modified for better navigation through components.

The *Obligation Explorer* is implemented by the following classes in package `org.eventb.internal.ui.obligationexplorer`:

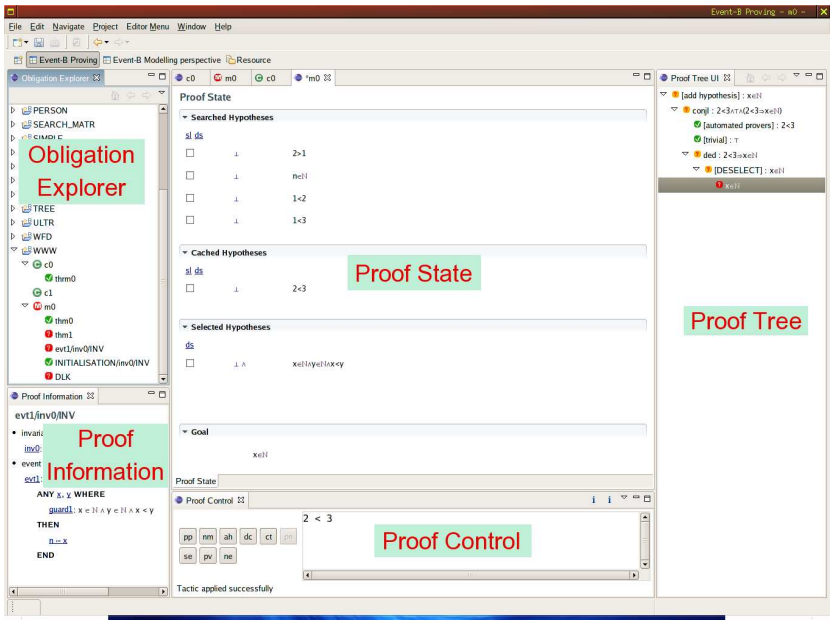


Figure 3: Proving UI Workbench

`ObligationExplorer` The main class for the *Obligation Explorer*.

`ObligationExplorerActionGroup` Implements the actions associated with the *Obligation Explorer*.

`ObligationExplorerContentProvider` Connects to the *Rodin Database* for fetching relevant data and feed them to the *Obligation Explorer*'s tree viewer.

5.2.2 *Proof State*

The *Proof State* (as mentioned earlier) is indeed a multi-page and form editor. The main page of the editor displays the current goal and various kinds of hypotheses. The *Proof State* is implemented by the following classes in package `org.eventb.internal.ui.prover`:

`ProverUI` Main class for the *Proof State*.

`ProofsPage` Implements the main page of the editor.

`GoalSection` Displays the goal in the main page.

`HypothesesSection` Abstract class for showing different set of hypotheses in the main page.

`SelectedHypothesesSection` Displays selected hypotheses in the main page.

`CachedHypothesesSection` Displays cached hypotheses in the main page.

`SearchHypothesesSection` Displays searched hypotheses in the main page.

`HypothesisRow` Displays one hypothesis in an hypotheses sections.

The other pages can be used for showing information related to the proof such as the list of all hypotheses. No other page is implemented in the prototype.

The *Proof State* is connected to the "User Support" (see Sec. 5.3 on the next page) which provide the information to display. Each instance of the editor corresponds to one instance of the *User Support*. Multiple instances of the editor can be open at the same time in order to work in parallel on different proof obligations.

5.2.3 *Proof Information*

This view displays information relevant to the current proof obligation, such as the event and the invariant that gave rise to it. This view also allows the user to go back to the Editing UI by selecting the appropriate element. For instance, if the user finds some errors and want to modify the event related to the current proof obligation, the user can easily do so by selecting this event in the *Proof Information* view.

The *Proof Information* is associated with the *Proof State*, and thus always displays the information related to the current active editor. The information shown in this view is provided by the same instance of the *User Support* in the editors. In package `org.eventb.internal.ui.prover`, the *Proof Information* is implemented by the following classes:

`ProofInformation` This implements a page book view (where each page gives the proof information corresponding to one editor).

`IProofInformationPage` The interface for each page in the view.

`ProofInformationPage` The implementation of each page in the view. This is connected to the *User Support* in order to get the relevant information to show in the page.

5.2.4 *Proof Tree*

The *Proof Tree* is implemented in a similar way to the *Content Outline* view of the *Proof State* (the editor). This gives the current *Proof Tree* and helps to navigate through the proof, such as jumping between subgoals, backtracking one or more proof steps, reusing some part of the current or other proofs that have already been done.

In package `org.eventb.internal.ui.prover`, the *Proof Information* is implemented by the following classes:

`ProofTreeUI` This implements a page book view (where each page gives the proof tree corresponding to one editor).

`IProofTreeUIPage` The interface for each page in the view.

`ProofTreeUIPage` The implementation of each page in the view. This contains a tree viewer.

`ProofTreeUIContentProvider` Implements the connection with the *User Support* in order to get the relevant data and feed it to the *Proof Tree*'s tree viewer.

`ProofTreeUIActionGroup` Implements the set of actions that can be applied to the proof tree.

5.2.5 *Proof Control*

The *Proof Control* provides buttons for controlling the proof process. Beside the list of buttons, it also provide a Text Area for entering a formula (in the Mathematical Language). During interactive proof steps, messages are shown in the bottom of this view. In package `org.eventb.internal.ui.prover`, the *Proof Information* is implemented by the following classes:

`ProofControl` This implements a page book view (where each page gives the proof control corresponding to one editor).

`IProofControlPage` The interface for each page in the view.

`ProofControlPage` The implementation of each page in the view. This is connected to the *User Support* in order to get relevant information to display in the page.

5.3 The User Support

The *User Support* implements the connection between the *Proving UI* and the sequent prover (interactive prover). This maintains the state of the proofs, and provides information to display in the *Proving UI*. Each instance of the *User Support* corresponds to one instance of the *Proof State*, hence also to one instance of a *Proof Information*, *Proof Control* and *Proof Tree*.

The *Proving UI* sends the task to do to the *User Support* (for example, apply a tactic at the current proof node). The *User Support* then passes this task to the sequent prover. The sequent prover performs the task and modifies the state of the proof accordingly. Then, the *User Support* computes the difference between the previous state (before performing the task) and the new state (after performing the task) and notifies the *Proving UI* of the changes.

The *User Support* is responsible for keeping the state of the proofs (including the set of cached and searched hypotheses).

In package `org.eventb.core.pm`, the *User Support* is implemented by the following classes:

`UserSupport` The main class for the *User Support*.

`ProofState` The implementation for keeping track of the state of one proof obligation, including the proof tree. One instance of the *User Support* manages a set of these proof states.

Other supporting interfaces (in package `org.eventb.core.pm`) and classes (in package `org.eventb.internal.core.pm`) are used to implement the delta mechanism in the *User Support*.

References

[D5] C. Métayer et al. *Final Decisions.* Rodin Deliverable D3.1 (D5). 28th February 2005.

[D7] C. Métayer et al. *Event-B Language*. Rodin Deliverable D3.2 (D7). 31st May 2005.

[D10] L. Voisin (Ed) *Specification of Basic Tools and Platform*. Rodin Deliverable D3.3 (D10). 31st August 2005.