User Manual for Rodin v.2.3

Work in Progress
Handbook Rev: 13635

rodin-handbook@formalmind.com
handbook.event-b.org

December 8, 2011

Contents

Chapter 1

Introduction

This handbook provides documentation for users of the Rodin toolset, which allows working with Event-B models.

Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extensible with plugins.

This handbook covers the use of the core platform. Documentation for developers and regarding extensions can be found in the Rodin wiki (1.1.2).

What you see here is a working draft of the documentation. We are grateful for any feedback that you may have (1.1.3).

1.1 Overview

This handbook consists of five parts:

Introduction You are reading the introduction right now. It helps you to orient yourself and to find information quickly.

Tutorial If you are completely new to Rodin, the tutorial is a good way to get up to speed quickly. It guides you through the installation and usage of the tool and gives you an overview of the Event-B modeling notation.

Reference The reference section provides comprehensive documentation of Rodin, and its components.

Frequently Asked Questions Common issues are listed by category in the FAQ.

Index We included an index especially for the print version of the handbook. In the electronic versions, you may want to try the search functionality as well.

1.1.1 Formats of this Handbook

The handbook comes in various formats:

Eclipse Help The Rodin Handbook is shipped with Rodin and can be accessed through the help system. The handbook will be updated with the standard Rodin update mechanism.

Online Help You can access the handbook online at `http://handbook.event-b.org`.

PDF Help Both online versions also include a link to the PDF version of the handbook.

Phyiscal Book If there is enough interest, we may make the handbook available through a print-on-demand retailer.

1.1.2 Rodin Wiki

This handbook is complemented by the Rodin wiki (http://wiki.event-b.org/). Sometimes, the handbook will refer to the wiki for more information. Also, plugin and developer information is usually located in the wiki.

1.1.3 Feedback

All online versions of the handbook contain a button or link for feedback. Work on the handbook will continue until at least January 2012, so your feedback will be read and will help to improve this handbook. You can also submit feedback via email to `rodin-handbook@formalmind.com`.

1.2 Foreword

It would be nice to recruit somebody for the foreword - maybe Cliff and/or Jean-Raymond.

1.3 Conventions

We use the following conventions in this manual:

Checklists and Milestones are designated with a tick. Here we summarize what we want to learn or should have learned so far.

Useful information and tricks are designated by the information sign.

Potential problems and warnings are designated by a warning sign.

Examples and Code are designated by a pencil.

We use `typewriter` font for file names and directories.

We use sans serif font for GUI elements like menus and buttons. Menu actions are depicted by a chain of elements, separated by "⟩", e.g. File ⟩ New ⟩ Event-B Component.

1.4 Acknowledgements

The content of this handbook has been growing since the formation of the European Union IST Project RODIN in 2004. Giving credit to every contributor is almost impossible and attempting to do so would almost certainly omit some people, which would contradict the spirit of this work. It should be sufficient to say that we extend our gratitude to all contributors to the Rodin Wiki (1.1.2). In particular, we would like to thank Systerel[1] for their significant contributions to the handbook as they have been the main driver behind the tool and its documentation.

We would also like to thank Cliff Jones, who never gave up the quest to improve the Rodin documentation.

The icons that you find throughout this handbook were created by Pixel-Mixer[2], who provides them for free. Thanks!

[1] http://www.systerel.fr

[2] http://pixel-mixer.com/

1.5 DEPLOY

This work has been sponsored by the DEPLOY project[3]. DEPLOY is a European Commission Information and Communication Technologies FP7 project.

The overall aim of the EC Information and Communication Technologies FP7 DEPLOY Project is to make major advances in engineering methods for dependable systems through the deployment of formal engineering methods. Formal engineering methods enable greater mastery of complexity than found in traditional software engineering processes. It is the central role played by mechanically-analysed formal models throughout the system development flow that enables mastery of complexity.

As well as leading to big improvements in system dependability, greater mastery of complexity also leads to greater productivity by reducing the expensive test-debug-rework cycle and by facilitating increased reuse of software.

The goal of the project is to achieve and evaluate industrial take-up of DEPLOY's methods and tools (which started with DEPLOY's industrial partners) as well as to perform further research on methods and tools that is considered necessary.

1.6 Creative Commons Legal Code

The work presented here is the result of a collaborative effort that took many years. To ensure that access to this work stays free and to avoid any legal ambiguities, we have decided to formally license it under the Creative Commons Share-Alike License.

[3] http://www.deploy-project.eu/

Chapter 2

Tutorial

The objective is to get you to a stage where you can use Rodin and build Event-B models. We expect you to have a basic understanding of logic and an idea why doing formal modeling is a good idea. You should be able to work through the tutorial with little or no outside help.

This tutorial covers installation and configuration for Rodin. It brings you step by step through building formal models, it provides the essential theory and provides pointers to more information.

We attempt to alternate between theory and practical application and thereby keep you motivated. We encourage you not to download solutions to the examples but instead to actively build them up yourself as the tutorial progresses.

If something is unclear, remember to check the Reference chapter (Chapter 3) for more information.

2.1 Outline

Background before getting started (2.2) We give a brief description of what Event-B is, what it is being used for and what kind of background knowledge we expect.

Installation (2.3) We guide you through downloading, installing and starting Rodin and point out platform differences. We install the provers. We name the visible views and describe what they are doing.

A Machine, and nothing else (2.4) We introduce the first machine: a traffic light with booleans for signals. We introduce guards which result in the proof obligations being automatically discharged. We explain how proof labels are read without changing to the proof perspective.

Mathematical notation (2.5) At this point we quickly go through the most important aspects of predicate calculus and provide pointers to the reference chapter and to external literature. We cover everything used by the traffic light system, we introduce all data types and we provide a brief intoduction of sets and relations. We also explain the difference between predicates and expressions. For example, we explain here how to understand the difference between TRUE and $\top$.

Introducing Contexts (2.6) We introduce contexts to apply the theoretical concepts that were introduced in the previous section. We use the Agatha-Puzzle to step by step introduce more and more complex elements. We point out that partitions are a typical pitfall, we cover theorems and also mention well-definedness.

Event-B Concepts (2.7) This is another theoretical section that provides more background about the previous examples. For instance, we analyze the anatomy of a machine and introduce all elements that a machine or context may have. We give references to literature about the theory but do not go into the details of the calculus. We describe the sees and refines concepts which will be applied in the next section. We will briefly mention concepts like data refinement and witnesses but leave the details to the literature.

Expanding the Traffic Light System (2.8) We apply what we learned in the previous section by introducing a context with traffic light colors and a refinement to integrate them. We will introduce another refinement for the push buttons.

Proving (2.9) So far all proof obligations were discharged automatically. Now we switch to the proving perspective and explore it for the first time. We change the configuration for the auto prover, invalidate proofs and show that with the new configuration they will not discharge any more. We prove a simple proof by hand and describe the provers available.

Complete Abrial Example (2.10) We will pick an interesting example from the Abrial book if we get permission. We can also use one of the Rodin Wiki Tutorial examples (e.g. Location Access Controller).

Outlook (2.11) This concludes the tutorial, but we will provide many pointers for further reading. In particular, we will point to the literature from the Deploy project and the Wiki and to plugins that solve specific problems.

2.2 Before Getting Started

Before we get started with the actual tutorial, we are going to go over the required background information to make sure that you have a rudimentary understanding of the necessary concepts.

You can skip this section, if...

- ... you know what formal modelling is
- ... you know what predicate logic is
- ... you know what Event-B and Rodin are
- ... you know what Eclipse is

2.2.1 Systems Development

Ultimately, the purpose of the methods and tools introduced here is to improve systems development. By this we mean the design and management of complex engineering projects over their life cycle. Examples include cars, air traffic control systems, etc.

"Taking an interdisciplinary approach to engineering systems is inherently complex since the behaviour of and interaction among system components is not always immediately well defined or understood. Defining and characterizing such systems and subsystems and the interactions among them is one of the goals of systems engineering. In doing so, the gap that exists between informal requirements from users, operators, marketing organizations, and technical specifications is successfully bridged."[1]

2.2.2 Formal Modelling

We are concerned with *formalizing specifications*. This allows us a more rigorous analysis (thereby improving the quality) and allows us to reuse the specification in the development an implementation. This comes at the cost of higher up-front investments.

This differs from the traditional development process. In a formal development, we transfer some effort from the test phase (where the implementation is verified) to the specification phase (where the specification in relation to the requirements is verified).

[1] http://en.wikipedia.org/wiki/Systems_engineering#Managing_complexity

2.2.3 Predicate Logic

In predicate logic statements (called predicates) can be expressed over variables that can be quantified, like in "for all values of x ...". Event-B's logic is predicate logic with the following features:

- Predicates and expressions are distinguished.
- All expressions have a data type, e.g. integer or set of integers.
- Quantification over variables, not predicates, is supported. This includes quantification over sets.
- A partial function semantics is included, i.e. the predicate $1 \div 0 = 1 \div 0$ is not a tautology because $1 \div 0$ does not represent a valid value.
- Comprehension sets are supported.
- Predicates can be evaluated to a Boolean values.

2.2.4 Event-B

Event-B is a notation for formal modelling based around an abstract machine notation ().

Event-B is considered an evolution of B (also known as classical B). It is a simpler notation which is easier to learn and use. It comes with tool support in the form of the Rodin Platform.

2.2.5 Rodin

Rodin (3.1) is the name of the tool platform for Event-B. It allows formal Event-B models to be created with an editor. It generates proof obligations (3.2.6) that can be discharged either automatically or interactively.

Rodin is modular software and many extensions are available. These include alternative editors, document generators, team support, and extensions (called plugins) to the notation which include decomposition or record support. An up-to-date list of plugins is maintained in the Rodin Wiki (1.1.2)[2].

2.2.6 Eclipse

Rodin is based on the Eclipse Platform (3.1.1), a Java-based platform for building software tools. This matters for two reasons:

- If you have already used Eclipse-based software, then you will feel immediately comfortable with the handling of the Rodin application.
- Many extensions, or plugins, are available for Eclipse-based software. There are Rodin-specific plugins as well as Rodin-independent plugins that may be useful to you. The Rodin Wiki (1.1.2), contains a list of plugins is maintained.

The GUI of an Eclipse application consists of views, editors, toolbars, quickviews, perspectives and many more elements. If these terms are unfamiliar to you, please consult Section 3.1.1 which contains references to Eclipse tutorials.

In Section 2.3, we present the Rodin-specific GUI elements.

[2]Specifically, there links were valid at the time of writing:
http://wiki.event-b.org/index.php/Rodin_Plug-ins
http://wiki.event-b.org/index.php/Installing_external_plug-ins_manually

2.3 Installation

Goals: The objective of this section is to guide you through downloading, installing and starting Rodin. In addition, we explain the update mechanisms needed to install new plugins for Rodin. Finally, we name the Rodin-specific GUI elements and describe their functions.

You can skip this section, if. . .

- . . . you know how to install and update Rodin
- . . . you know how to install new plugins for Rodin

Rodin is fairly resource intensive. You need a good computer with plenty of memory to run it. Recommended is at least 2GB of memory.

2.3.1 Install Rodin for the first time

Step 1: Download

The first step is to download Rodin. Rodin is available for download at the Rodin Download page (4.1.5)

Rodin is available for Windows, Mac OS, and Linux. No matter which platform you use, the distribution is always packed in a zip-file. Download the zip file for your system anywhere on your PC.

It is recommended that you download the latest stable version.

Step 2: Install and Run Rodin

To install Rodin, extract the contents of the zip file to a desired directory. You can run the tool by using the `rodin` executable.

Starting Rodin should bring up a welcome screen. It provides some quick guidance to Rodin. In particular, it provides instructions on installing the provers.

Please install the provers right away. It is easy and only takes a few clicks.

After dismissing the welcome screen, you should see the window shown in Figure 2.1. Here you can specify the path where Rodin stores your projects.

After specifying a path click on the OK button. Rodin should start and bring up the window shown in Figure 2.2.

When using a Linux distribution, a welcome window may open up. Exit out of this window to get to the main screen. Other problems can also occur when installing Rodin in Linux. See the release notes for details.[3]

As already mentioned in Section 2.2.6, the GUI of an Eclipse application consists of views, editors, toolbars, quickviews, perspectives and many more elements. We name the different Rodin GUI elements (i.e. views) which are visible after starting Rodin for the first time and explain their functions:

Menu bar (3.1.2) The menu bar of the Rodin programs provides file and edit operations and other commands.

Tool bar (3.1.2) The tool bar provides short cuts for familiar commands like save, print, undo and redo.

Event-B Explorer (3.1.2) The Event-B Explorer shows the projects' tree structures. It has projects as main entries and for each one its corresponding project files.

[3] See Rodin Release Notes

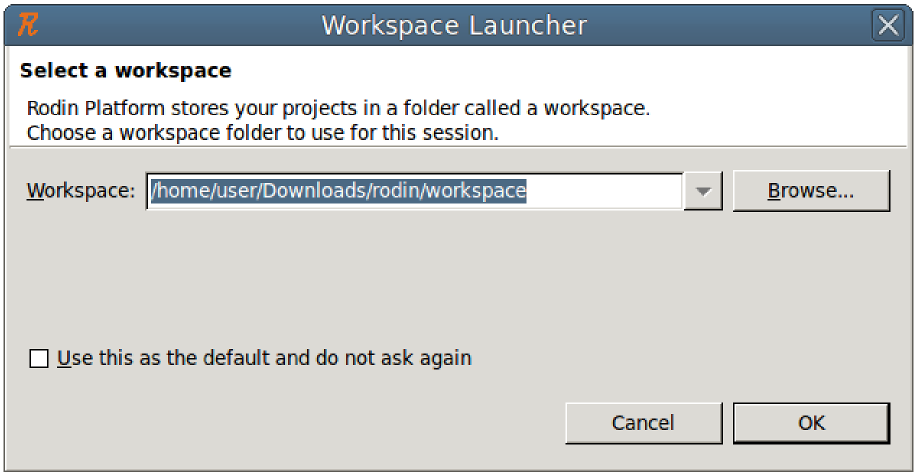

Figure 2.1: Eclipse Workspace Launcher

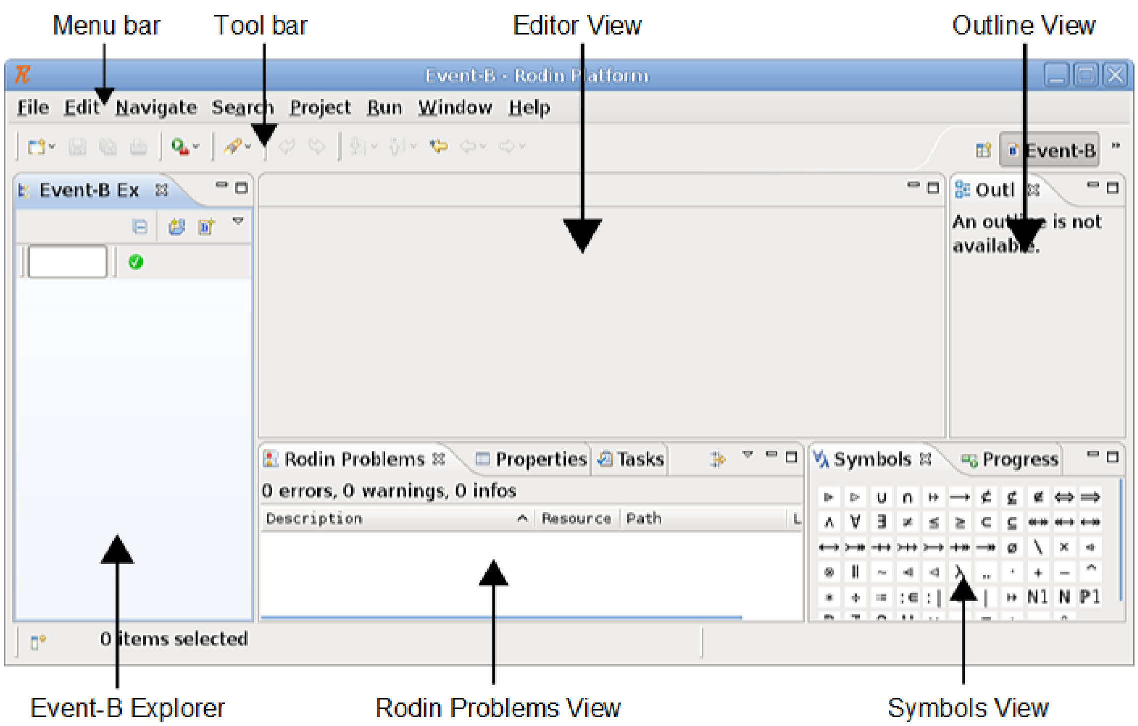

Figure 2.2: Rodin GUI

Outline View (3.1.2) A view used to view the outline of the active editor or file respectively.

Rodin Problems view (3.1.2) The Rodin Problems View shows problems (e.g., syntax errors) in the active editor.

Symbols View (3.1.2) A view that shows a list of available mathematical symbols which can be used in conjunction with the mathematical notation (3.3).

Editor View (3.1.2) The editor view contains the active editor.

2.3.2 Install new plugins

This sections shows how to install new plugins for Rodin by using the example of the Atelier B Provers plugin (3.4.4). It is highly recommended that you install this plugin because it will not be possible to prove much without it.

Open the Install Manager Help ⟩ Install New Software. . . . Click the downward arrow next to the field Work with to select the Atelier B Provers update site. Check the box next to the Atelier B Provers entry and click on the Next button (compare with figure 2.3). Follow the installation instruction to install the plugin. After installing the plugin, you will be asked to restart Rodin in order to finalize the installation.

If you are using a firewall, you may need to change the proxy settings.

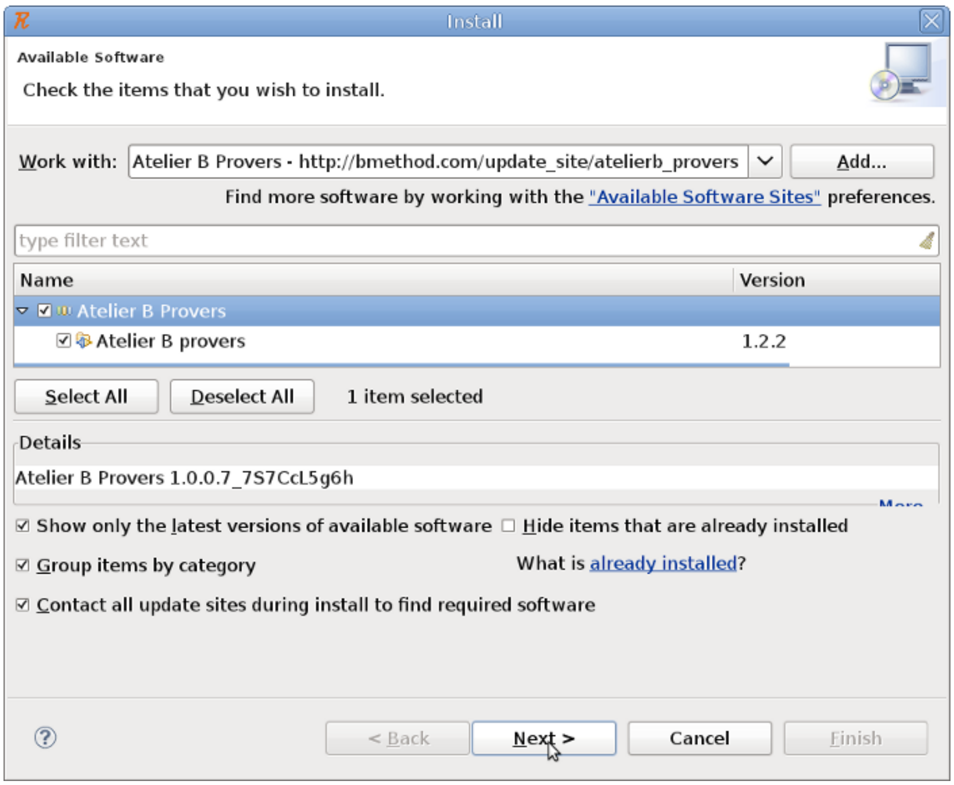

Figure 2.3: Eclipse Install Manager

2.4 The First Machine: A Traffic Light Controller

Goals: The objective of this section is to get acquainted with the modeling environment. We will create a very simple model consisting of just one file to develop a feeling for Rodin and Event-B.

In this tutorial, we will create a model of a traffic light controller. We will use this example repeatedly in subsequent sections. Figure 2.4 depicts what we are trying to achieve.

In this section, we will implement a simplified controller with the following characteristics:

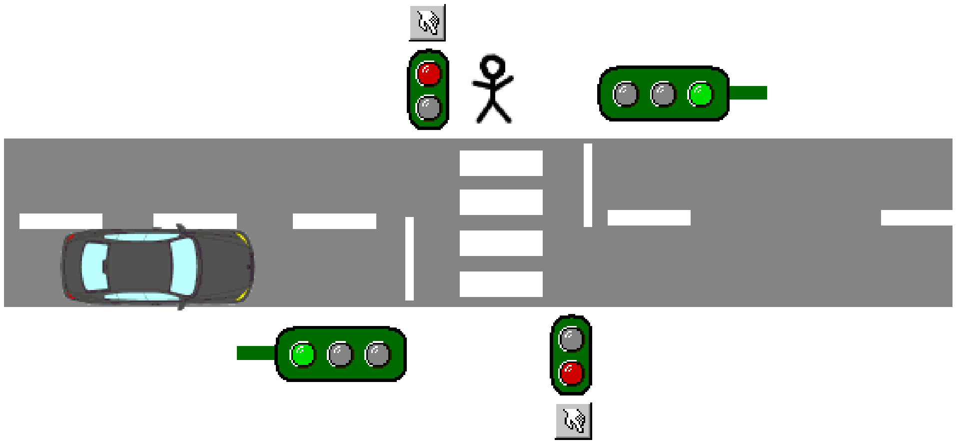

Figure 2.4: The traffic light controller

- We will model the signals with boolean values to indicate "stop" (false) and "go" (true). We do not model colors (yet) because we think we should first specify our goal (regulating the traffic) and later we should add implemtation details (the traffic light's colors).
- Too keep the initial model simple, we will not include the push button yet. We will add it later.

2.4.1 Excursus: The specification process

While this handbook is concerned with use of the Rodin tool, it's important to understand the specification process as well. Especially for beginners it can be daunting and unclear where to start with the model, what kind of data structures and abstractions to use, and so on.

We cover a few examples in this chapter that implicitly answer these questions, but there is no explicit set of instructions. For example, we will first model the traffic lights as booleans, and later refine them into actual colors. But how did we come up with this refinement strategy? Likewise, we decided to add the push buttons at a later refinement. In retrospect this may seem useful, but it leaves open the question on how we arrived at this structure in the first place.

Abrial has something to say about this in his book[4], for which some chapters are available in the Rodin Wiki.

2.4.2 Project Setup

Models typically consist of multiple files that are managed in a project. Create a new Event-B Project File ⟩ New ⟩ Event-B Project. Give the project the name `tutorial-03` as shown in Figure 2.5.

Eclipse supports different types of projects. The project must have the Rodin Nature (3.1.1) to work. A project can have more than one nature.

Next, create a new Event-B Component. Either use File ⟩ New ⟩ Event-B Component or right-click on the newly created project and select New ⟩ Event-B Component. Use `mac` as the component name, select Machine as component-type, and click Finish as shown in Figure 2.6. This will create a Machine (3.2.3) file.

The newly created component will open in the structural editor. The editor has four tabs at the bottom. The Pretty Print shows the model as a whole with color highlighting, but it cannot be edited here. This is

[4] http://www.amazon.com/Modeling-Event-B-System-Software-Engineering/dp/0521895561

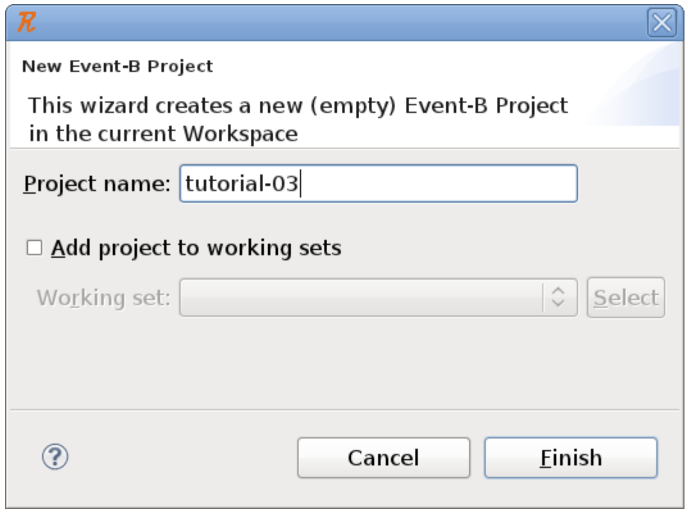

Figure 2.5: New Event-B Project Wizard

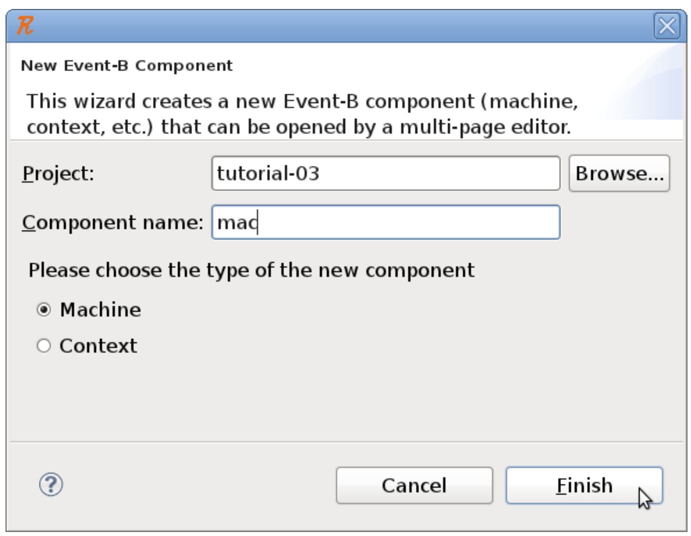

Figure 2.6: New Event-B Component Wizard

useful to inspect the model. The Edit allows editing of the model. It shows the six main sections of a machine (REFINES, SEES, etc.) in a collapsed state. You can click on the ▷ button to the left of a section to expand it.

The editor is *form-based*. This means that in well-defined places an appropriate control (text field, dropdown, etc.) allows modifications.

Alternative editors are available as plug-ins. The form editor has the advantage of guiding the user through the model, but it takes up a lot of space and can be slow for big models. The text-based Camille Editor (2.4.3) is very popular. Please visit the Rodin Wiki (1.1.2) for the latest information.

2.4.3 Camille, a text-based editor

Camille is a "real" text editor that provides the same feel as a typical Eclipse text editor, including copy and paste, undo, redo, etc. However, please note that at this time, not all Rodin plugins are compatible with Camille. Also, please consult the extensive documentation in the Rodin Wiki (1.1.2).

Camille can be installed via its update site, which is preconfigured in Rodin. Once installed, Camille is made the default editor. The structural editor can still be used by selecting it from the context menu of a file in the project browser.

For more information, please visit http://wiki.event-b.org/index.php/Text_Editor.

This contribution requires the **Camille** plugin. The content is maintained by the plugin contributors and may be out of date.

2.4.4 Building the Model

Back to the problem: Our objective is to build a simplified traffic light controller as described in 2.4. We start with the model state. Two traffic lights will be modelled and we will therefore create two variables called `cars_go` and `peds_go`.

Creating Variables

Go to the Edit tab in the editor and expand the VARIABLES section. Click on the button to create a new variable. You will see two fields. The left one is filled with the word `var1`. Change this to `cars_go`. The second field (after the double-slash "//") is a comment field in which you can write any necessary notes or explanations.

Comments: The comment field supports line breaks. Note that it is not possible to "comment out" parts of the model, as is possible with most programming languages. You can use the comment field to "park" predicates and other strings temporarily.

Create the second variable (`peds_go`) in the same way.

Upon saving, the variables will be highlighted in red, indicating an error as shown in Figure 2.7. The Rodin Problems view (3.1.2) shows corresponding error messages. In this case, the error message is "Variable cars_go does not have a type".

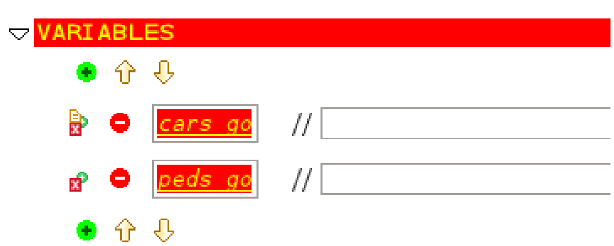

Figure 2.7: Red highlighted elements indicate errors

Types are provided by invariants. Expand the INVARIANTS section and add two elements by following the same steps as above. Invariants have labels. Default labels are generated (`inv1` and `inv2`). The actual invariant is prepopulated with $\top$, which represents the logical value "true". Change the first invariant (the

⊤, not the label `inv1`) to $cars\_go \in BOOL$ and the second invariant to $peds\_go \in BOOL$. Event-B provides the build-in datatype `BOOL` amongst others (3.3.1).

Mathematical Symbols: Every mathematical symbol has an ASCII-representation and the substitution occurs automatically. To generate "element of" (∈), simply type a colon (":"). The editor will perform the substitution after a short delay. The Symbols view shows all supported mathematical symbols. The ASCII representation of a symbol can be found by hovering over the symbol in question.

After saving, you should see that the EVENTS section is highlighted in yellow as demonstrated in Figure 2.8. Again, the Rodin Problems view gives us the error message: "Variable cars_go is not initialized". Every variable must be initialized in a way that is consistent with the model.

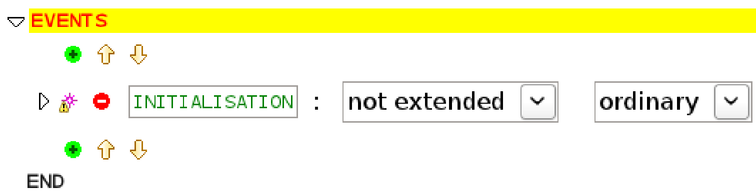

Figure 2.8: Yellow highlighted elements indicate warnings

To fix this problem, expand the EVENTS section and then the INITIALIZATION event. Add two elements in the THEN block. These are actions that also have labels. In the action fields, enter $cars\_go := FALSE$ and $peds\_go := FALSE$.

State Transitions with Events

Our traffic light controller cannot yet change its state. To make this possible, we create events (3.2.3). We will first consider the traffic light for the pedestrians, and we will create two events. One will set it to "go" and one will set it to "stop".

From now on, we won't describe the individual steps in the editor any more. Instead, we will simply show the resulting model.

The two events will look as follows:

Event $set\_peds\_go \mathrel{\widehat{=}}$
  begin
    act1 : $peds\_go := TRUE$
  end
Event $set\_peds\_stop \mathrel{\widehat{=}}$
  begin
    act1 : $peds\_go := FALSE$
  end

Event parameters

For the traffic light for the cars, we present a different approach and use only one event with a parameter. The event will use the new traffic light state as the argument. The parameter is declared in the any section and typed in the where section:

Event *set_cars* $\widehat{=}$
  any
    new_value
  where
    grd1 : $new\_value \in BOOL$
  then
    act1 : $cars\_go := new\_value$
  end

Note how the parameter is used in the action block to set the new state.

Invariants

If this model was actually in control of a traffic light, we would have a problem because nothing is preventing the model from setting both traffic lights to `TRUE`. The reason is that so far we only modeled the domain (the traffic lights and their states) and not the requirements. We have the following safety requirement:

REQ-1: Both traffic lights must not be `TRUE` at the same time.

We can model this requirement with the following invariant:

$$\neg(cars\_go = TRUE \land peds\_go = TRUE)$$

Please add this invariant with the label `inv3` to the model, use `not` and `&` for $\neg$ and $\land$.

Obviously, this invariant can be violated, and Rodin informs us of this. The Event-B Explorer (3.1.2) provides this information in various ways. Go to the explorer and expand the project (`tutorial-03`), the machine (`mac`) and the entry "Proof Obligations". You should see four proof obligations, two of which are discharged (marked with ✓) and two of which are not discharged (?).

Proof obligations: A proof obligation is something that has to be proven to show the consistency of the machine, the correctness of theorems, etc. A proof obligation consists of a label, a number of hypothesis that can be used in the proof and a goal – a predicate that must be proven. Have a look at the proof obligation labels. They indicate the origin in the model where they were generated. E.g. set_peds_go/inv3/INV is the proof obligation that the event set_peds_go preserves the invariant (INV) with the label inv3. An overview about all labels can be found in 3.2.6. The proof obligations can also be found via other entries in the explorer, like the events they belong to. Elements that have non-discharged proof obligations as children are marked with a small question mark. For instance, `inv3` has all proof obligations as children, while the event `set_cars` has one.

To prevent the invariant from being violated (and therefore to allow all proof obligations to be discharged), we need to strengthen the guards (3.2.3) of the events.

Before looking at the solution, try to fix the model yourself.

Finding Invariant Violations with ProB

A useful tool for understanding and debugging a model is a model checker like ProB. You can install ProB from the ProB Update Site, directly from Rodin. Just select Install New Software... from the Help menu and select "ProB" from the dropdown. You should see "ProB for Rodin2" as an installation option, which you can then install using the normal Eclipse mechanism.

We will continue the example at the point where we added the safety invariant (REQ-1), but didn't add guards yet to prevent the invariants from being violated.

We launch ProB by right-clicking on the machine we'd like to animate and select Start Animation / Model Checking. Rodin will switch to the ProB-Perspective, as shown in Figure 2.9. The top left pane shows the available events of the machines. Upon starting, only INITIALIZATION is enabled. The middle pane shows the current state of the machine, and the right pane shows a history. On the bottom of the main pane we can see whether any errors occurred, like invariant violations. We can now interact with the model by triggering events. this is done by double-clicking on an enabled event, or by right-clicking it and selecting a set of parameters, if applicable. We first trigger INITIALIZATION. After that, all events are enabled. Next, we trigger `set_cars` and `set_peds_go` with the parameter $TRUE$. As expected, we will get an invariant violation. In the state view, we can "drill down" and find out which invariant was violated, and the history view shows us how we reached this state (Figure 2.10). After modifying a machine, ProB has to be restarted, which is done again by right-clicking the machine and selecting ProB. Triggering events to find invariant violations is not very efficient. But ProB can perform model checking automatically. To do so, select Model Checking from the Checks menu from the "Events" View (the view on the left). After optionally adjusting some parameters, the checking can be triggered by pressing "Start Consistency Checking". Upon completion, the result of the check is shown. ProB has many more functions and also supports additional formalisms. Please visit the ProB Website for more information.

This contribution requires the **ProB** plugin. The content is maintained by the plugin contributors and may be out of date.

2.4.5 The Final Traffic Light Model

MACHINE mac
VARIABLES
  `cars_go`
  `peds_go`
INVARIANTS
  `inv1` : $cars\_go \in BOOL$
  `inv2` : $peds\_go \in BOOL$
  `inv3` : $\neg(cars\_go = TRUE \land peds\_go = TRUE)$
EVENTS
Initialisation
  begin
    `act1` : $cars\_go := FALSE$
    `act2` : $peds\_go := FALSE$
  end
Event *set_peds_go* $\widehat{=}$
  when
    `grd1` : $cars\_go = FALSE$
  then
    `act1` : $peds\_go := TRUE$
  end
Event *set_peds_stop* $\widehat{=}$
  begin
    `act1` : $peds\_go := FALSE$

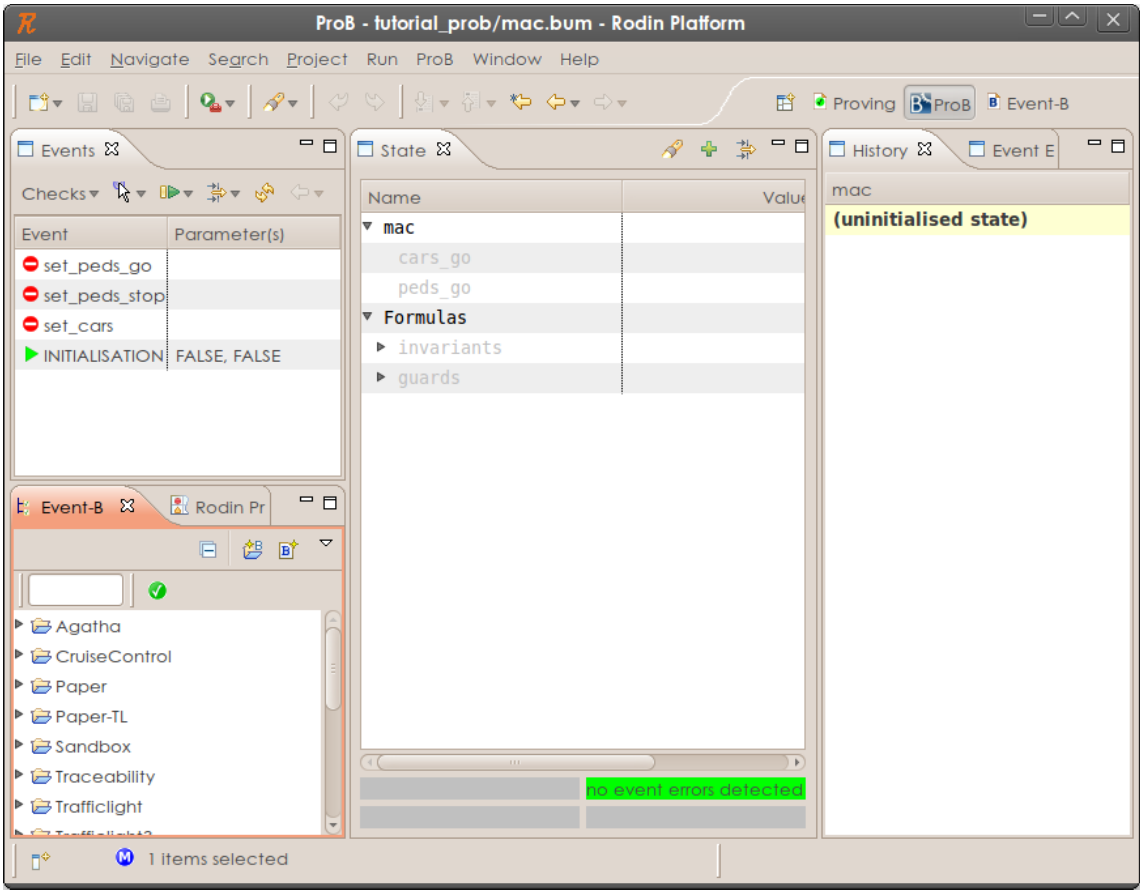

Figure 2.9: The ProB Perspective

    end
Event $set\_cars \mathrel{\widehat{=}}$
    any
        new_value
    where
        grd1 : $new\_value \in BOOL$
        grd2 : $new\_value = TRUE \Rightarrow peds\_go = FALSE$
    then
        act1 : $cars\_go := new\_value$
    end
END

2.5 Mathematical notation

Goals: In order to understand how basic properties of a model can be expressed in Event-B, we need a brief introduction of predicates, terms and data types.

In Event-B, we use a mathematical notation to describe the systems we want to model. This allows us to be very precise about the model's properties.

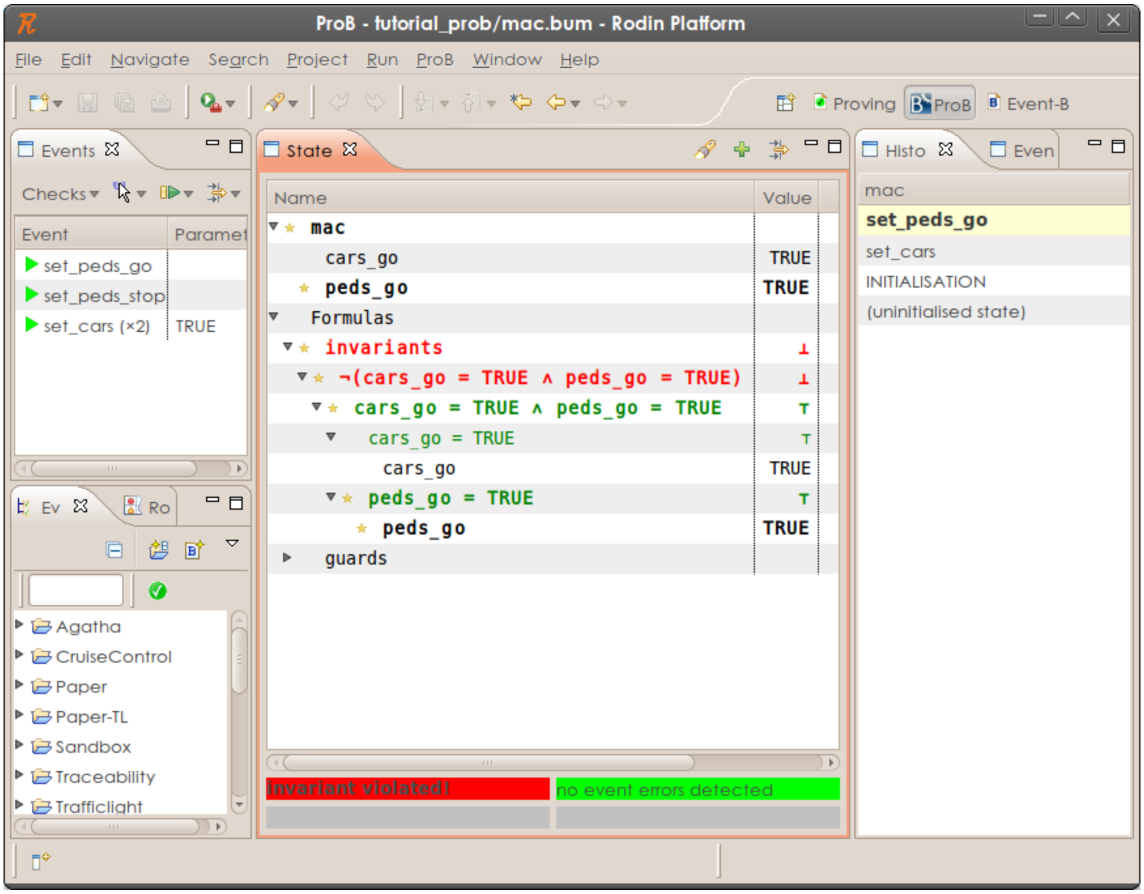

Figure 2.10: An invariant violation, found by ProB

2.5.1 Predicates

In the traffic light example, we have already encountered several predicates: The invariants of a model and the guards of an event. The proof obligations generated by Rodin are also predicates. A predicate is simply an expression that evaluates to true or false.

The simplest predicates are $\top$ (ASCII: `true`) and $\bot$ (ASCII: `false`). We can also assert if arbitrary objects of the same type are equal with $=$ or not equal with $\neq$ (ASCII: `/=`). Predicates can be combined with the usual logical operators:

	symbol	ASCII
conjunction (and)	$\wedge$	`&`
disjunction (or)	$\vee$	`or`
implication	$\Rightarrow$	`=>`
equivalence	$\Leftrightarrow$	`<=>`
negation (not)	$\neg$	`not`

We can use *universal quantification* to express a statement that should hold for all possible values a variable might have. For example, in order to show that any given number x greater than zero and multiplied with two is greater than one, we can use the following expression:

$$\forall x \cdot x > 0 \Rightarrow 2 \cdot x > 1 \qquad \text{ASCII: } \texttt{!x. x>0 => 2*x > 1}$$

When a variable is introduced by a quantifier, the type of the variable must be clear. In this case Rodin can infer that x must be of type integer because the operator $<$ is defined only on integers. Sometimes the type cannot be inferred, e.g., in

$$\forall a, b \cdot a \neq b \Rightarrow b \neq a \qquad \text{ASCII: } \texttt{!a,b. a/=b => b/=a}$$

a and b could be integers, boolean values or some other type. In this case, we must make the type of the variables explicit by stating that a and b are elements of the appropriate sets. Let's use integers again:

$$\forall a, b \cdot a \in \mathbb{Z} \land b \in \mathbb{Z} \land a \neq b \Rightarrow b \neq a \qquad \text{ASCII: } \texttt{!a,b. a:INT \& b:INT \& a/=b => b/=a}$$

The conjunction operator ($\land$) has a stronger binding that the implication $\Rightarrow$, so the above is equivalent to

$$\forall a, b \cdot (a \in \mathbb{Z} \land b \in \mathbb{Z} \land a \neq b) \Rightarrow b \neq a$$

If you are unsure which of the operators bind stronger, we advise you to use parenthesis to avoid mistakes.

Existential quantification on the other hand is used to state that there is an object of a certain type fulfilling a given property. Let's express the statement that there is a Boolean value different from TRUE.

$$\exists x \cdot x \in \text{BOOL} \land x \neq \text{TRUE} \qquad \text{ASCII: } \texttt{\#x. x:BOOL \& x/=TRUE}$$

As you can see, we again added type information for x. We put the type information for the universal quantification on the left side of the implication ($\Rightarrow$), but for existential quantification we add it via a conjunction ($\land$).

2.5.2 Data types

We have seen that each identifier (i.e. a variable, constant or parameter) must have a distinguished type. If we can introduce an identifier anywhere, we usually must also add a predicate with which the identifier's type can be determined. In the traffic light example, a variable `cars_go` was introduced and typed by an invariant $cars\_go \in \text{BOOL}$. In the next section, we'll see constants that will be typed by axioms (also predicates) and later we'll see parameters that will be typed by guards (again, predicates).

In general, each term in Event-B has a certain type. When saving a Event-B component, Rodin starts the type checker to ensure that types are correctly used. For example, the terms on both sides of an equality ($=$) must have the same type. If this is not the case, Rodin will generate an error message. For each type there exists a set that denotes exactly all elements that belong the type. We will now briefly give an overview about all types you might encounter.

Integers We have already seen numbers, which are of type integer ($\mathbb{Z}$). Example terms of type $\mathbb{Z}$ are 5, $x + 7$ and $7 \cdot y - 3$.

Booleans We have already seen the Boolean type (BOOL) in the previous section (2.4). It has exactly two elements, $\text{BOOL} = \{\text{TRUE}, \text{FALSE}\}$.

Carrier sets An user can introduce a new type by adding its name to the Sets section of a context. We see that in more detail in the next section (2.6).

Sets If we have terms of a certain type, we can easily construct sets of that type. E.g. 1 and $2 \cdot x$ denote integers ($\mathbb{Z}$) and $\{1, 2 \cdot x\}$ is a set of integers ($\mathbb{P}(\mathbb{Z})$). $\mathbb{P}(S)$ (ASCII: `POW`) denotes the power set (the set of all subsets) of S.

Pairs If we have two terms, we can construct a pair. For example, with 2 and TRUE, we can construct the pair $2 \mapsto \text{TRUE}$ (ASCII: `2|->TRUE`). The type of that pair is $\mathbb{Z} \times \text{BOOL}$, where $\times$ denotes the Cartesian product.

Set of pairs ("relations") play an important role in modeling languages like Event-B.

Please do not confuse predicates and Boolean values! For example, if you want to express the condition "if the variable b is true, x should be greater than 2", you *cannot* write $b \Rightarrow x > 2$ (That would raise a syntax error). Instead you can write $b = \mathrm{TRUE} \Rightarrow x > 2$.

In the reference section (3.3) the types of each operator in Event-B are described in detail.

2.5.3 Operations on Sets

Let's assume that we have two sets A and B of the same type, e.g. sets of integers. Then we can check if an element e is in it by $e \in A$ (ASCII: `e:A`) or on if it is not in A by $e \notin A$ (ASCII: `e/:A`). Expressing that all elements of A are also elements of B (i.e. A is a subset of B) can be done by $A \subseteq B$ (ASCII: `A<:B`). The negated form is $A \not\subseteq B$ (ASCII: `A/<:B`).

We can build the union $A \cup B$, the intersection $A \cap B$ and the set subtraction $A \setminus B$, (ASCII: `A\/B`, `A/\B` and `A\B`). The set subtraction contains all elements that are in A but not in B.

The power set $\mathbb{P}(A)$ (ASCII: `POW(A)`) is the set of all subsets of A. Thus $B \in \mathbb{P}(A)$ is equivalent to $B \subseteq A$. $\mathbb{P}_1(A)$ (ASCII: `POW1(A)`) is the set of all non-empty subsets of A.

2.5.4 Introducing user-defined types

We can introduce our own new types simply by giving such types a name. This is done by adding the name of the type to the *SETS* section of a context. We will see how this is done in practice in the next section (2.6).

For instance, if we want to model different kind of fruits in our model, we might add $FRUITS$ to our sets. Then the identifier $FRUITS$ denotes the set of all elements of this type. Nothing more is known about $FRUITS$ unless we add further axioms. In particular, we do not know the cardinality of the set or even if it is finite.

Assume that we want to model *apples* and *oranges* which are sub-sets of $FRUITS$. We do not need to introduce them in the *SETS* section of a context just because they are sets. Let's imagine such a scenario where *apples* and *oranges* are modeled as types of their own (by declaring them in the *SETS* section). And we have two variables or constants a and o with $a \in apples$ and $o \in oranges$. Then we cannot compare a and o with $a = o$ or $a \neq o$. That would raise a type error because $=$ and $\neq$ expect the same type for the left and right expression.

If we want to model sub-sets *apples* and *oranges* as described above, we can add them as constants and state $apples \subseteq FRUITS$ and $oranges \subseteq FRUITS$. If apples and oranges are all fruits we want to model, we can assume $apples \cup oranges = FRUITS$ and if no fruit is both an apple and orange we can write $apples \cap oranges = \varnothing$. This can be expressed shorter by saying that apples and oranges constitute a partition of the fruits: $partition(FRUITS, apples, oranges)$. In general, we can use the partition operator to express that a set S is partitioned by the sets $s_1, \ldots, s_n$ with $partition(S, s_1, \ldots, s_n)$. We use partitions in Section 2.6.2.

Another typical usage for user defined data types are *enumerated sets*. These are sets where we know all the elements already. Let's take a system which can be either working or broken. We model this by introducing a type $STATUS$ in the *SETS* section and two constants *working* and *broken*. We define that STATUS consists of exactly *working* and *broken* by $STATUS = \{working, broken\}$. Additionally, we have to say that *working* and *broken* are not the same by $working \neq broken$.

If the enumerated sets gets larger, we need to state for every two element of the set that they are distinct. Thus, for a set of 10 constants, we'll need $(10^2 - 10) \div 2 = 45$ predicates. Again, we can use the partition operator to express this in a more concise way: $partition(STATUS, \{working\}, \{broken\})$.

2.5.5 Relations

Relations are a powerful instrument when modeling systems. From a mathematical point of view a relation is just a set of pairs. Formally, when we have to sets A and B, we can specify that r is a relation between both by $r \in \mathbb{P}(A \times B)$ (ASCII: `r:POW(A**B)`). Because relations are so common, we can write it shorter $r \in A \leftrightarrow B$ (ASCII: `r:A<->B`).

With $a \mapsto b \in r$, we can check if two elements a and b are related in respect to b.

We use a small example to illustrate relations. Let $A = \{a, b, c, d\}$ and $B = \{1, 2, 3, 4\}$. We define the relation r with $r = \{a \mapsto 1, a \mapsto 3, c \mapsto 2, d \mapsto 1\}$. The *domain* of r are all elements occurring on the left side $\text{dom}(r) = \{a, c, d\}$ and the *range* are all elements on the right $\text{ran}(r) = \{1, 2, 3\}$.

To find out to which elements the objects of the set $s = \{b, c, d\}$ are related to, we can use the *relational image*: $r[s] = r[\{b, c, d\}] = \{1, 2\}$. Often we want to know to which object a single element b is related. We just write it as a singleton set: $r[\{a\}] = \{1, 3\}$.

Event-B supports several operators to work with relations. (3.3.5) We will not go into more detail during the course of the tutorial.

An important special case of relations are functions. Functions are relations where each element of the domain is uniquely related to one element of the range. Event-B directly supports operators to describe partial and total functions, which can injective, surjective or bijective.

2.5.6 Arithmetic

We have the usual operations on integers, $+$, $-$, $\cdot$ and $\div$ (ASCII: `+`, `-`, `*` and `/`). They can be compared with the usual $<$, $\leq$, $\geq$, $>$ (ASCII: `<`, `<=`, `>=`, `>`).

$\mathbb{Z}$ (ASCII: `INT`) denotes the set of all integer numbers. $\mathbb{N}$ and $\mathbb{N}_1$ (ASCII: `NAT` resp. `NAT1`) are the subsets of natural numbers.

If you specify two variables x and y with $x \in \mathbb{Z}$ and $y \in \mathbb{N}$, then both are of type integer ($\mathbb{Z}$). $\mathbb{N}$ is not another type. There is just the additional condition $y \geq 0$.

2.6 Introducing Contexts

Goals: In this section we introduce contexts that apply to the theoretical concepts that were introduced in the previous section (2.5). We will create a very simple model consisting of just one context file.

In this tutorial, we will create a model of the well known Agatha puzzle. We use this instead of the already introduced traffic light example because it provides us with more possibilities to apply Event-B's logic and to use operations on relations. Here is a brief description of the puzzle:

Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates no one that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. No one hates everyone. Who killed Agatha?

Contexts are used to model static properties of a model, things that do not change over time. Whereas with machines we model the dynamic properties like the traffic light above. The objective of this Section is to get familiar with contexts by modeling the Agatha puzzle.

2.6.1 Create a Context

Create a new Event-B Project File ⟩ New ⟩ Event-B Project. Give the project the name `tutorial-05`.

Next, create a new Event-B Component. Unlike in Section (2.4) use `agatha` as the component name and mark the Context (3.2.2) option in order to create a Context file instead of a Machine (3.2.3) file.

Click on Finish. Rodin should start the editor with the created Context file (see Figure 2.11).

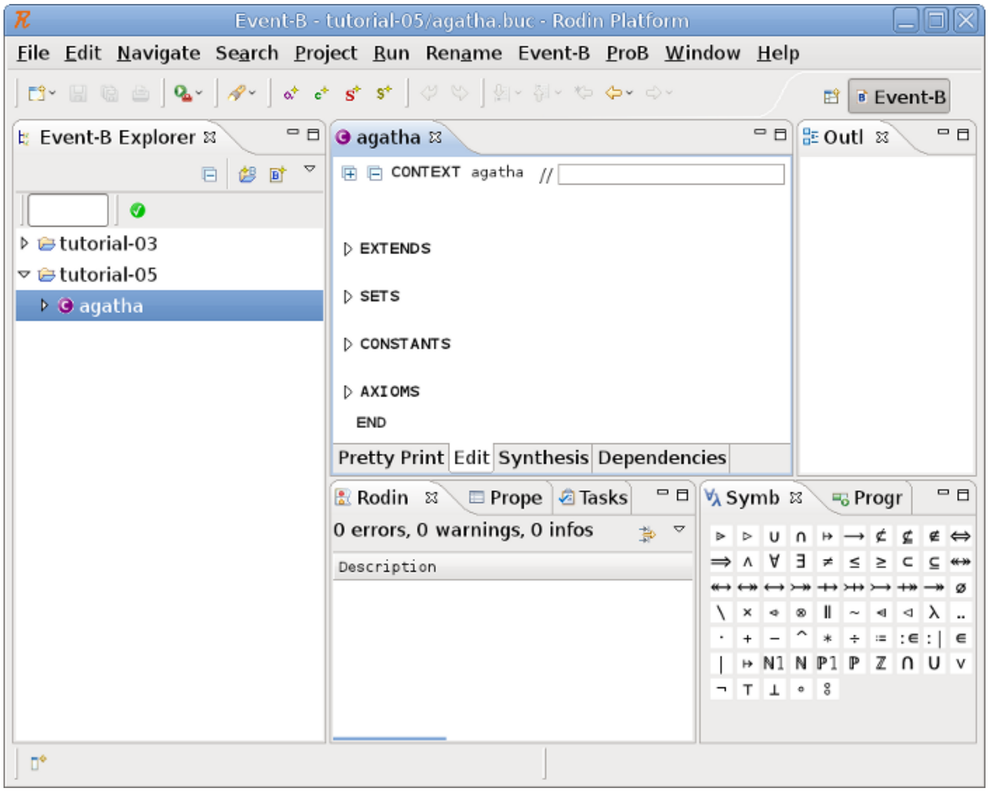

Figure 2.11: Context file opened with Structural Editor

2.6.2 Populate the Context

In this section we model the Agatha puzzle step by step.

Modelling the Persons

We have three persons in the Agatha puzzle: Agatha herself, the butler and Charles. We model the three persons as constants (one constant for each person) in the corresponding CONSTANTS section:

CONSTANTS
```
Agatha
butler
Charles
```

These constants or persons respectively are part of a set:

SETS
```
persons
```

Now, the constants itself are not very useful since they have no type (In addition, the constants will be highlighted in red, indicating a problem). The semantics of the sets (3.3.4) and constants (3.2.2) are specified in the axioms (3.2.2). As already mentioned above the persons are part of the set `persons`. We model this by creating a partition (3.3.4) in the AXIOMS section:

AXIOMS

person_partition : $partition(persons, \{Agatha\}, \{butler\}, \{Charles\})$

Please note the curly braces {} around the constants. It's very easy to forget these, which will result in typing errors that are very hard to interpret for a novice.

The New Enumerated Set Wizard (3.1.4) allows you to create the constants, the set and the axiom automatically at the same time. To access this wizard, click on the New Enumerated Set Wizard tool bar item or find it under Event-B ⟩ New Enumerated Set Wizard. This will bring up the wizard where we can enter the name of the set and the constants in the corresponding text fields. The wizard will create the enumarted set, the constants and the axiom automatically.

Modelling the Relations "Persons who hate each other" and "Who's how rich"

We create two more constants `hates` and `richer` to model the relations "Persons who hate each other" and "Who's how rich". The relations are abstract, which means that they say nothing about the concrete persons (Agatha, the butler and Charles). We define the concrete relationships between the persons later in this section.

The first constant `hates` is an arbitrary relation (3.3.5) between `persons`:

AXIOMS

hate_relation : $hates \in persons \leftrightarrow persons$

The second constant `richer` is also a relation between `persons`:

AXIOMS

richer_relation1 : $richer \in persons \leftrightarrow persons$

However, we know that the relation is irreflexive (no person is richer than itself):

AXIOMS

richer_relation2 : $richer \cap id = \varnothing$

In addition, we know that the relation is transitive:

AXIOMS

richer_relation3 : $(\forall x, y, z \cdot (x \mapsto y \in richer \land y \mapsto z \in richer) \Rightarrow x \mapsto z \in richer)$

Finally, the relation is trichotomous (one person is always richer than the other or vice versa, never both directions):

AXIOMS

richer_relation4 : $(\forall x, y \cdot x \in persons \land y \in persons \land x \neq y \Rightarrow (x \mapsto y \in richer \Leftrightarrow y \mapsto x \notin richer))$

Modelling the "Crime"

Since the objective of the puzzle is to find the killer, we have to create a new constant `killer` which is an element of `persons`:

CONSTANTS
    killer
AXIOMS
    killer_type : $killer \in persons$

In addition, the puzzle have some more relationships between the different persons which are all modelled as axioms. We know that the killer hates his victim and is no richer than his victim:

AXIOMS
    killer_hates : $killer \mapsto Agatha \in hates$
    killer_not_richer : $killer \mapsto Agatha \notin richer$

Charles hates no one that Agatha hates and Agatha hates everybody except the butler:

AXIOMS
    charles_hates : $hates[\{Agatha\}] \cap hates[\{Charles\}] = \varnothing$
    agatha_hates : $hates[\{Agatha\}] = persons \setminus \{butler\}$

The butler hates everyone not richer than aunt Agatha and the butler hates everyone whom Agatha hates. However, no one hates everyone:

AXIOMS
    butler_hates_1 : $\forall x \cdot (x \mapsto Agatha \notin richer \Rightarrow butler \mapsto x \in hates)$
    butler_hates_2 : $hates[\{Agatha\}] \subseteq hates[\{butler\}]$
    noone_hates_everyone : $\forall x \cdot x \in persons \Rightarrow hates[\{x\}] \neq persons$

Finally, we have to model the solution:

AXIOMS
    solution : $killer = Agatha$

We mark the axiom `solution` as a theorem (2.7.1) by clicking on the `not theorem` button[5] as shown in figure 2.12.

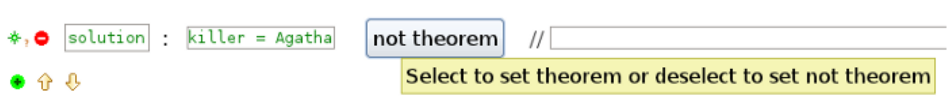

Figure 2.12: Mark an Axiom as Theorem

Theorems describe properties that are expected to be able to be derived from the axioms. Therefore, to prove a theorem you only use axioms and theorems that have already been proven.

[5] Yes, this sounds wrong. However, the button is a toggle button, so after clicking it, the label changes to "theorem". The button always displays the current state.

The introduced theorem still has to be proven. Thus Rodin generates a proof obligation called solution/THM. But at this point of the tutorial we do not want to go into the details of proof yet.

You can use ProB to animate contexts, too. Just right-click on the context in the explorer and select Start Animation / Model Checking. If ProB finds solutions for the specified constants that fulfil the axioms, an event "SETUP_CONTEXT" is enabled that assigns values to the constants. In our example, ProB should find a solution where Agatha is the murder. You can actually inspect the axioms and the theorem in the state view to see why they are fulfilled.

This contribution requires the **ProB** plugin. The content is maintained by the plugin contributors and may be out of date.

This concludes the tutorial. The following section shows the complete Context.

2.6.3 The Final Context

CONTEXT agatha

SETS

persons

CONSTANTS

Agatha

butler

Charles

hates

richer

killer

AXIOMS

person_partition : $partition(persons, \{Agatha\}, \{butler\}, \{Charles\})$

hate_relation : $hates \in persons \leftrightarrow persons$

richer_relation : $richer \in persons \leftrightarrow persons \land$
$richer \cap id = \varnothing \land$
$(\forall x, y, z \cdot (x \mapsto y \in richer \land y \mapsto z \in richer) \Rightarrow x \mapsto z \in richer) \land$
$(\forall x, y \cdot x \in persons \land y \in persons \land x \neq y \Rightarrow (x \mapsto y \in richer \Leftrightarrow y \mapsto x \notin richer))$

killer_type : $killer \in persons$

killer_hates : $killer \mapsto Agatha \in hates$

killer_not_richer : $killer \mapsto Agatha \notin richer$

charles_hates : $hates[\{Agatha\}] \cap hates[\{Charles\}] = \varnothing$

agatha_hates : $hates[\{Agatha\}] = persons \setminus \{butler\}$

butler_hates_1 : $\forall x \cdot (x \mapsto Agatha \notin richer \Rightarrow butler \mapsto x \in hates)$

butler_hates_2 : $hates[\{Agatha\}] \subseteq hates[\{butler\}] \land$
$(\forall x \cdot x \in persons \Rightarrow hates[\{x\}] \neq persons)$

noone_hates_everyone : $\forall x \cdot x \in persons \land hates[\{x\}] \neq persons$

solution : theorem $killer = Agatha$

END

2.7 Event-B Concepts

In Event-B we have two kind of components. We already encountered a *context* that describes static elements of a model. The other component is a *machine* that describes the dynamic behavior of a model.

2.7.1 Contexts

A context has the following components:

Sets User-defined types can be declared in the *SETS* section (See also 3.3.4).

Constants indexconstant We can declare constants here. The type of each constant must be given by the axioms.

Axioms The axioms are a list of predicates. indexaxiom They describe what can be taken for granted when developing a model. The axioms can be later used in proofs that occur in components that use ("*see*") this context. Each axiom has a label attached to it.

Theorems Axioms can be marked as *theorems*. In that case, we declare that the predicate is provable by using the axioms that are written before the theorem. Theorems can be used later in proofs just like the axioms.

Extends A context may extend an arbitrary number of other contexts. Extending another context A has the effect that we can make use of all constants and axioms declared in A and we can add new constants and axioms.

Rodin automatically generates *proof obligations* (often abbreviated as PO) for properties that should be proven. Each proof obligation has a name that explains where it comes from. There are two kind of proof obligations generated in a context:

- Each theorem must be proven. The proof obligation's name has the form label/THM, where label is the theorem's label.
- Some expressions are not obviously *well-defined*. E.g. the axiom $x \div y > 2$ is only meaningful if y is different from 0. Thus Rodin generates the proof obligation $y \neq 0$. A well-defined proof obligation has the name label/WD.

The order of the axioms and theorems matter because the proof of a theorem or the degree to which an expression is well-definded may depend on the axioms and theorems that are already written. This is necessary to avoid circular reasoning.

2.7.2 Machines

A machine describes the dynamic behavior of a model by means of variables whose values are changed by events. A central aspect of modeling a machine is to prove that the machine never reaches an invalid state, i.e. the variables always have values that satisfy the invariant. First we briefly summarize of which parts a machine consists:

Refines Optionally a machine can refine another one. We'll see in Section 2.7.4 what that means.

Sees We can use the context's sets, constants and axioms in a machine by declaring it in the *Sees* section. The axioms can be used in every proof in the machine as hypothesis.

Variables The variables' values are determined by an initialization and can be changed by events. Together they constitute the state of the machine. Each variable must be given a type by the invariants.

Invariants This are predicates that should be true for every reachable state. Each invariant has a label.

Events An event can assign new values to variables. The *guards* of an event specify under which conditions it might occur. The initialization of the machine is a special case of an event.

2.7.3 Events

We saw in Section 2.4 what an event basically looks like using the example of a traffic light:

Event *set_cars* $\widehat{=}$
  any
    new_value
  where
    grd1 : $new\_value \in BOOL$
  then
    act1 : $cars\_go := new\_value$
  end

We have the event's name *set_cars*, a *parameter* with the name *new_value*, a *guard* with label grd1 and an *action* with label act1. An event can have an arbitrary number of parameters, guards and events.

The guards specify *when* an event might occur, i.e. under which combinations of the values of the machine's variables and the parameters. The actions describe *what* changes apply to the variables.

Only the variables that are explicitly mentioned in the actions are affected. All the other variables keep their old values. Beside the simple assignment ($:=$), there are other forms of actions ($:\in$ or $:|$) which are explained in the reference section 3.2.3.

The *initialization* of the machine is a special form of event. It has neither parameters nor guards.

Now our aim is to prove that the invariants always hold. To do this, we must prove two things:

- The initialization leads to a state where the invariant holds.
- Assuming that the machine is in a state where the invariant holds, every enabled event leads to a state where the invariant holds.

Rodin generates proof obligations for every invariant that can be changed by an event, i.e. the invariant contains variables changed by an event. The name of the proof obligation is then event_name/invariant_label/INV. The goal of such a proof is to assert that when all affected variables are replaced by new values from the actions, the invariant still holds. The hypotheses for such a proof obligation consist of:

- All invariants, because we assume that all invariants hold before the event is triggered,
- All guards, because events can only be triggered when the guards are valid.

In the special case of an initialization, we cannot use the invariants because we do not make any assumptions about uninitialized machines.

2.7.4 Refinement

Refinement is a central concept in Event-B. Refinements are used to gradually introduce the details and complexity into a model. If a machine `B` refines a machine `A`, `B` is only allowed to behave in a way that corresponds to the behavior of `A`. We now look into more detail of what "corresponds" here means. In such a setting, we call `A` the abstract and `B` the concrete machine.

Here we give just a brief overview over the concept of refinement. Later in Section 2.8 we use refinement in an example.

The concrete machine has its own set of variables. Its invariants can refer to the variables of the concrete and the abstract machine. If a invariant refers to both, we call it a "gluing invariant". The gluing invariants are used to relate the states between the concrete and abstract machines.

The events of a concrete machine can now refine an abstract event. To ensure that the concrete machine does only what is allowed by the abstract one, we must show two things:

- The concrete events can only occur when the abstract one can occur.

- If a concrete event occurs, the abstract event can occur in such a way that the resulting states correspond again, i.e. the gluing invariant remains true.

The first condition is called “guard strengthening”. The resulting proof obligation has the label concrete_event/abstract_guard/GRD. We have to prove that under the assumption that the concrete event is enabled (i.e. its guard are true) and the invariants (both the abstract and the concrete) hold, the abstract guards holds as well. Thus the goal is to prove that the abstract guard, the invariants and the concrete guards can be used as hypothesis in the proof.

The second condition, that the gluing invariant remains true, is just a more general case of the proof obligation which ensures that an event does not violate the invariant. So the proof obligation's label is again concrete_event/concrete_invariant/INV. The goal is to prove that the invariant of the concrete machine is valid when each occurence of a modified variable is replaced by its new value. The hypotheses we can use are:

- We assume that the invariant of both the concrete and abstract machines hold before the event occurred.
- The abstract invariants where the modified variables are replaced by their new values are valid because we know that the abstract event does not violate the invariants.
- The event occurs only when the guards of both the concrete and abstract machines are true.

These two conditions are the central piece to prove the correctness of a refinement. We now just explain a few common special cases.

Variable re-use

Most often, we do not want replace all variables by new ones. It is sometimes useful to keep all of the variables. We can do this just by repeating the names of the abstract variables in the variable section of the concrete machine. In that case, we must prove for each concrete event that changes such a variable that the corresponding abstract event updates the variable in the same way. The proof obligation has the name concrete_event/abstract_action/SIM.

Introducing new events

An event in the concrete machine might not refine any event in the abstract machine. In that case it is assumed to refine *skip*, which is the event that does nothing and can occur any time. The guard strengthening is then trivially true and doesn't need to be proven. It still must be proven that the gluing invariant holds but this time under the assumption that the abstract machine's variables have not changed. Therefore, the new state of our newly introduced event corresponds to the same state of our abstract machine from before the event happened.

Witnesses

Let's consider a situation where we have an abstract event with a parameter p and in a refining event that no longer needs that parameter. We saw above that we have to prove that for each concrete event the abstract event may act accordingly. With the parameter, however, we now have the situation that we must prove the existence of a value for p such that an abstract event exists. Proofs with existential quantification are often hard to do, so Event-b uses the construct of a *witness*. A witness is just a predicate of the abstract parameter with the name of the variable as label. Often a witness has just the simple form $p = \ldots$.

2.8 Expanding the Traffic Light System: Contexts and Refinement

Goals: We apply what we learned in the previous section by introducing a context with traffic light colours and a refinement to integrate them. We will also introduce another refinement for the push buttons.

2.8.1 Data Refinement

We will continue the example from Section 2.4, where we built a simplified model of a traffic light controller. The model was simplified because we abstracted the traffic lights to TRUE and FALSE and a number of features were still missing.

We will introduce data refinement in this section. The objective is to create a mapping between the abstract traffic light values and actual colours. Figure 2.13 depicts our mapping for the traffic light.

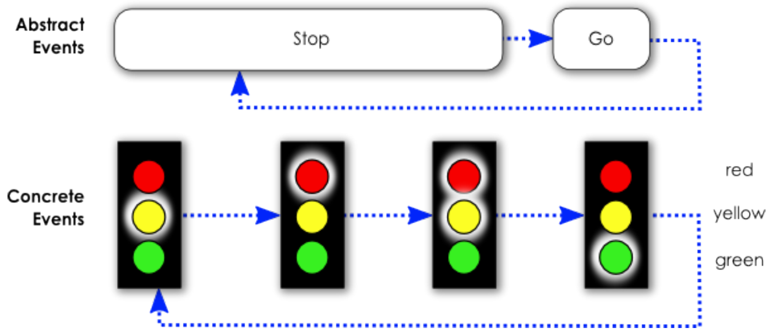

Figure 2.13: Mapping between Abstract and Concrete Events

For simplicity, the traffic light for pedestrians consists of only two lights: red and green.

We break this problem into two steps:

1. Create a context with the data structures for the colours.
2. Create a refinement of the existing model that sees the new context and refines the boolean states into colours.

2.8.2 A Context with Colours

Start by creating a context called `ctx1`, as described in Section 2.6. We model the colours of the traffic light as a so-called "enumerated set" (see 3.3.4): We explicitly specify all elements (the three colours) of a new user-defined data type. We define the constants:

CONSTANTS
```
red
yellow
green
```

We introduce the new data type as a set:

SETS
```
COLOURS
```

And last, we need to provide typing of the constants. We do this by creating a partition (3.3.4):

AXIOMS

`type :` $partition(COLOURS, \{red\}, \{yellow\}, \{green\})$

⚠ Please note the curly braces {} around the colours. It's very easy to forget these, which will result in typing errors that are very hard to interpret for a novice.

This completes the context.

2.8.3 The Actual Data Refinement

The easiest way to create a refinement is by right-clicking on the machine in the project browser and selecting Refine (in this case, we will be refining the machine mac from the project tutorial-3). This will create a "stub" consisting of all variables and events. Please use this method to create a machine with name `mac1`.

In refinements, the Edit View of the Editor hides some information from the abstract machine by default. This can be particularly important when you modify a refined event. The Pretty Print View shows all the element with elements from the abstract machine in *italics*.

For this tutorial, make sure that you right-click on the machine and select refine from the drop-down menu. If you have created a machine the normal way and later edited the refines section, the tutorial will assume that you have events (e.g. sets_peds_go) and variables that you do not have.)

First we have to make the machine aware of the context by adding a sees (3.2.3) statement:

MACHINE mac1
REFINES mac
SEES ctx1

We will start with the traffic lights for the pedestrians. It has only two colours (red and green) and only one of them is shown at a time. We introduce a new variable called peds_colour to represent which of the lights is shown. The variable has a corresponding invariant and initialization (the changes are shown in the following code snippet). The *extended* keyword in the initialisation means that all actions from the refined initialisation are copied:

VARIABLES
    `peds_colour`
INVARIANTS
    `inv4` : $peds\_colour \in \{red, green\}$
EVENTS
Initialisation
    extended
    begin
        `init4` : $peds\_colour := red$
    end
END

Next, we will create a gluing invariant (3.2.3) that associates peds_go from the abstract machine with the variable peds_colour that we just created. The gluing invariant will map TRUE to green and FALSE to red:

INVARIANTS
    `gluing` : $peds\_go = TRUE \Leftrightarrow peds\_colour = green$

In its current state, this gluing invariant can be violated: if the event set_peds_go is triggered, for instance, the variable peds_go will change but peds_colour will not. We expect that this will result in undischarged proof obligations (3.2.6). We can check this by expanding the machine in the Event-B Explorer. Indeed, we now see two undischarged proof obligations (compare with Figure 2.14).

To fix this, we have to modify the two events in question. Let's start with set_peds_go. First, we change the event from extended to not extended in the Editor as shown in figure 2.15.

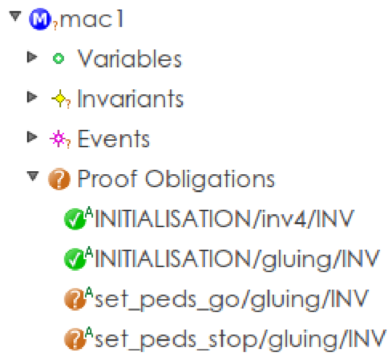

Figure 2.14: Mapping between Abstract and Concrete Events

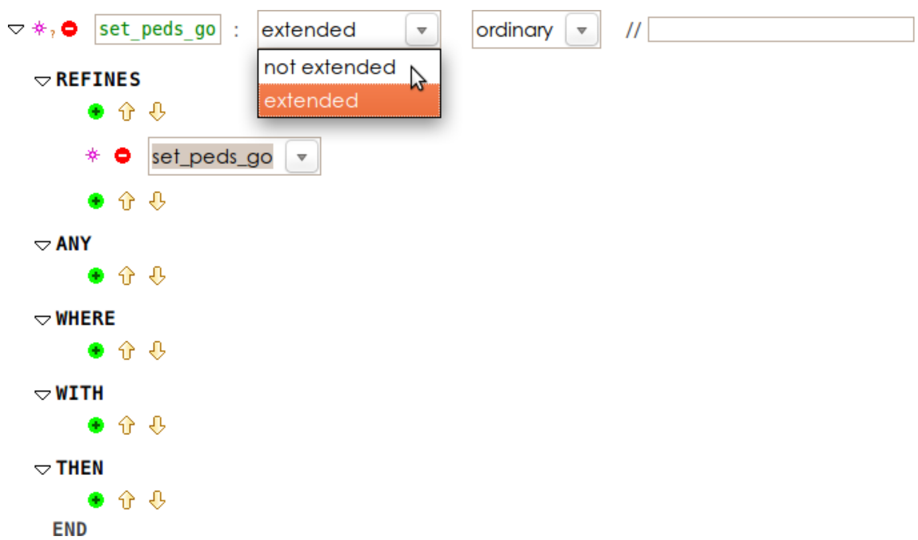

Figure 2.15: Switch from extended to not extended

This change will copy the guard and action from the abstract machine, so that we can modify it. We can now replace the action with the corresponding action regarding peds_colour (replacing peds_go := true with peds_colour := green). While we are at it, we can also rename the name of the event to something more fitting (e.g. set_peds_green).

Next, we perform the corresponding change on set_peds_stop (change the action to peds_colour := red and rename the event set_peds_red). Lastly, the event set_cars also contains a reference to peds_go that must be replaced (in the second guard, replace peds_go = FALSE with peds_colour = red).

Once all references to peds_go have been replace, we can remove the variable peds_go from the VARIABLES section. You will also need to change the INITIALISATION event to not extended and remove the action which initialises the variable peds_go. Now you shouldn't have any errors or warnings, and all proof obligations should be discharged.

If you get the error message "Identifier peds_go has not been declared", then there are references to the refined variable left somewhere in the model. It can be helpful to use the Pretty Print view, as it will show the "inherited" elements from the abstract machine as well.

2.8.4 The refined machine with data refinement for peds_go

```
MACHINE mac1
REFINES mac
SEES ctx1
VARIABLES
    cars_go
    peds_colour
INVARIANTS
    inv4 : peds_colour ∈ {red, green}
    gluing : peds_go = TRUE ⇔ peds_colour = green
EVENTS
Initialisation
    begin
        act1 : cars_go := FALSE
        init4 : peds_colour := red
    end
Event set_peds_green ≙
refines set_peds_go
    when
        grd1 : cars_go = FALSE
    then
        act2 : peds_colour := green
    end
Event set_peds_red ≙
refines set_peds_stop
    begin
        act1 : peds_colour := red
    end
Event set_cars ≙
refines set_cars
    any
        new_value
    where
        grd1 : new_value ∈ BOOL
        grd2 : new_value = TRUE ⇒ peds_colour = red
    then
        act1 : cars_go := new_value
    end
END
```

2.8.5 Witnesses

The refinement of set_cars is more difficult since the event uses a parameter (the new value for cars_go). In order to refine it, we need a witness (3.2.3).

A witness is to an event's parameter what a gluing invariant is to a variable: it is a mapping between the abstract parameter and the new parameter and allows the abstract parameter to disappear. In this example, the abstract parameter new_value is of type BOOL, and we introduce a new parameter new_value_colours of type COLOURS.

The naming of a witnesses' label is mandatory and must be the name of the abstract parameter.
In our example, the label must be new_value

Let's get started. We first provide the new variable, gluing invariant, typing invariant and initialization as we have done before (at this point you can also rename the gluing invariant from the last section as gluing_peds in order to be able to determine between the two gluing invariants). Note that the traffic light for the cars can show more than one colour at a time. Therefore, the variable contains a set of colours instead of just one colour (as modelled for peds_colour):

VARIABLES
    cars_colours
INVARIANTS
    inv5 : $cars\_colours \subseteq COLOURS$
    gluing_cars : $cars\_go = TRUE \Leftrightarrow green \in cars\_colours$
EVENTS
Initialisation
    begin
        init5 : $cars\_colours := \{red\}$
    end
END

We also have to modify the guard on set_peds_green, which is something that you should now be able to figure out yourself (just replace cars_go = FALSE with green $\notin$ cars_colours).

The interesting piece is the last event, set_cars, which we rename as set_cars_colours. We change the parameter to new_value_colours and type it as a subset of COLOURS.

The witness appears in the with section of the event. The label **must** be new_value. The value itself must describe the relationship between the abstract parameter new_value and the new parameter new_value_colours. As we use the parameter as the new value for the variable cars_colours, the witness is an adaptation of the gluing invariant (we just replace cars_colours with new_value_colours).

In most cases, the witness is a slightly modified gluing invariant.

Here is the resulting event:

Event *set_cars_colours* $\widehat{=}$
refines *set_cars*
    any
        new_value_colours
    where
        grd1 : $new\_value\_colours \subseteq COLOURS$
        grd2 : $green \in new\_value\_colours \Rightarrow peds\_colour = red$
    with
        new_value : new_value = TRUE $\Leftrightarrow$ green $\in$ new_value_colours
    then
        act1 : $cars\_colours := new\_value\_colours$
    end

Now you can get rid of the variable cars_go and its inisialisation clause, and there will not be any errors or warnings. But even though all proof obligations are now discharged, we're not done yet. Even though the traffic light doesn't violate the safety property from the abstract machine, it doesn't behave the way described in Section 2.8.1. We still have to ensure that the lights are activated in the proper sequence. We can impose this behavior by adding four guards each of which define one transition:

grd_y_r : $cars\_colours = \{yellow\} \Rightarrow new\_value\_colours = \{red\}$
grd_r_ry : $cars\_colours = \{red\} \Rightarrow new\_value\_colours = \{red, yellow\}$
grd_ry_g : $cars\_colours = \{red, yellow\} \Rightarrow new\_value\_colours = \{green\}$
grd_g_y : $cars\_colours = \{green\} \Rightarrow new\_value\_colours = \{yellow\}$

2.8.6 Discussion

Notice that we have used two very different approaches to model the traffic lights for cars and pedestrians. For the pedestrians, we created one event for each state transition. For the cars, we handled all states in one single event.

You will often be confronted with situations where many modelling approaches are possible. You should consider two main factors when modelling: (1) readability of the model and (2) ease of proof. In this case, both approaches are equally good (although we wouldn't recommend mixing different approaches in one model. We did it here only to demonstrate both approaches).

We will cover deadlock freeness later in Section 2.10.1. If you are interested in the topic, it may interest you to examine the traffic light model for deadlocks. Consider cars_colours = { green, red }. This is a legal state, but it would block set_cars_colours forever. A model checker (such as ProB) could find it. In this case, however, this is not a problem because with the given initialization and events this state is not reachable in the first place.

We hope that this section helped you to understand the power of abstraction. The safety invariant $\neg(cars\_go = TRUE \wedge peds\_go = TRUE)$ from Section 2.4.4 was very simple. We could now introduce colours because we are confident that the invariant will still hold (assuming, of course, that our gluing invariant is correct).

2.8.7 The Refined Machine with All Data Refinement

MACHINE mac1
REFINES mac
SEES ctx1
VARIABLES
    peds_colour
    cars_colours
INVARIANTS
    inv4 : $peds\_colour \in \{red, green\}$
    inv5 : $cars\_colours \subseteq COLOURS$
    gluing_peds : $peds\_go = TRUE \Leftrightarrow peds\_colour = green$
    gluing_cars : $cars\_go = TRUE \Leftrightarrow green \in cars\_colours$
EVENTS
Initialisation
    begin
        init4 : $peds\_colour := red$
        init5 : $cars\_colours := \{red\}$
    end
Event *set_peds_green* $\widehat{=}$
refines *set_peds_go*
    when
        grd1 : $green \notin cars\_colours$
    then
        act2 : $peds\_colour := green$
    end
Event *set_peds_red* $\widehat{=}$
refines *set_peds_stop*
    begin
        act1 : $peds\_colour := red$
    end
Event *set_cars_colours* $\widehat{=}$
refines *set_cars*

```
        any
            new_value_colours
        where
            grd1 : new_value_colours ⊆ COLOURS
            grd2 : green ∈ new_value_colours ⇒ peds_colour = red
            grd_y_r : cars_colours = {yellow} ⇒ new_value_colours = {red}
            grd_r_ry : cars_colours = {red} ⇒ new_value_colours = {red, yellow}
            grd_ry_g : cars_colours = {red, yellow} ⇒ new_value_colours = {green}
            grd_g_y : cars_colours = {green} ⇒ new_value_colours = {yellow}
        with
            new_value : new_value = TRUE ⇔ green ∈ new_value_colours
        then
            act1 : cars_colours := new_value_colours
        end
END
```

2.8.8 One more Refinement: The Push Button

We will demonstrate another application of refinement: introducing new features into an existing model. A typical traffic light system allows the pedestrians to request a light change by pressing a button. We will introduce this feature in a new refinement.

We could have introduced the push button in the initial machine, but introducing it later allows us to structure the model and makes it easier to understand and navigate.

We will realize this feature by introducing a new boolean variable for the push button. We will introduce an event that notifies the model that a push button has been pressed. Upon allowing the pedestrians to cross, we will reset the push button. This is a simplification of the problem. In practice, a lot would depend on the controller's capabilities. Specifically, how does the push button notification get to the controller software? How is the pressing/depressing sequence handled? In this example, the event directly sets the controller's state. This demonstrates the concept of feature refinement, without introducing too much complexity for a tutorial example.

As in the previous section, we create a new refinement mac2 by right-clicking on mac1 and selecting Refine. A stub is generated that contains the events from the abstract machine. We simply add a new variable for the push button (including typing and an initialisation clause). We also introduce an event that sets the button. This event doesn't work while the pedestrians have a green light.

```
VARIABLES
    button
INVARIANTS
    type_button : button ∈ BOOL
EVENTS
Initialisation
    extended
    begin
        init_button : button := FALSE
    end
Event push_button ≙
    when
        grd : peds_colour = red
    then
        act : button := TRUE
    end
END
```

Now we need to integrate the push button with the traffic light logic:

- Upon pressing the button, the pedestrians must eventually get a green light.
- At some point, the button variable must be reset.

As we will see in the following discussion, this be more tricky than it first appears. For now, we will introduce a guard preventing the car lights from turning green when button is true, and we will reset the button when the pedestrian lights turn red:

```
Event  set_peds_red ≙
extends  set_peds_red
     begin
          act_button : button := FALSE
     end
Event  set_cars_colours ≙
extends  set_cars_colours
     where
          grd_button : ¬(cars_colours = {red} ∧ button = TRUE)
     end
```

2.8.9 Discussion

There are a number of problems associated with the model in its current state. Let's start with the resetting of the button: The way we built our model so far, set_peds_red can be triggered any time; there is not a single guard which prevents this. Therefore, the button could be reset any time without the pedestrian light ever turning green.

This could be prevented with additional guards. For instance, the traffic light events could require an actual change in the light's status. This in turn could lead to deadlocks.

But even if we introduce such guards, we could get stuck in a situation where cars would never get green light any more. Consider the following scenario: (1) pedestrians get green light; (2) the light turns red; (3) a pedestrian presses the button again; (4) this prevents the car lights from turning green. Instead, the pedestrians get a green light again and the cycle continues.

There are tactics to address all these issues. However, it is rarely possible to generate proof obligations for such scenarios (without making the model much more complicated). It can be useful to use model checkers to validate the model's behaviour or even to use temporal logic to articulate properties of the model.

As an exercise, try to improve the model to address these issues.

2.9 Proving

Goals: The goal of this section is to get familiar with the Proving Perspective and to carry out a simple proof by hand.

2.9.1 The Celebrity Problem

In this section, we will work on the model of the so-called celebrity problem. We use a new model instead of the traffic light because it provides us with some proofs where manual interaction is necessary. In the setting for this problem, we have a "knows" relation between persons. This relation is defined so that

- no one knows himself,
- the celebrity knows nobody,

- but everybody knows the celebrity.

The problem's goal is to find the celebrity. We want to model an algorithm fulfills this task.

Make sure that you have no existing Project named Celebrity. If you do, then rename it. To do so, right click the project and select Rename...

Import the archive file Celebrity.zip[6] to you Event-B Explorer. To do this, select File ⟩ Import ⟩ General ⟩ Existing Projects into Workspace. Then select the according archive file and click on Finish.

Rodin takes a few seconds to extract and load all the files. Once it is done, it shows that there are a few problems with this project as you can see in the Rodin Problems View (compare with figure 2.16).

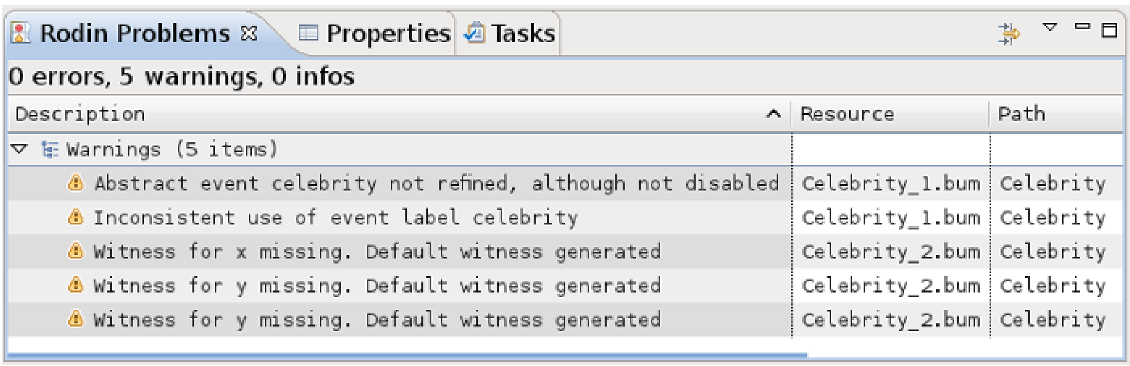

Figure 2.16: Warnings in the Rodin Problems View

We will describe how the model is organized below in 2.9.3.

In the first part of this section, our goal is to fix these problems. Let's take a look at the warning stating that the event label "celebrity" is misused ("Inconsistent use of event label celebrity"). Double-click on the warning to open the `Celebrity_1` machine. Select the event celebrity. The problem is that the event is not declared as a refinement. To solve the problem, expand the event and add a new entry in the REFINES section (Click on the button to create a new entry). This declares that the event is a refinement of an event with the same name in the abstract machine (3.2.3). As this is the case here, we can now save the project and the warning should disappear.

The three remaining warnings state that witnesses (3.2.3) are missing. Double click on the warning to open the concrete model (here `Celebrity_2`). Then expand the celebrity event and add a witness in the WITH section (Click on the button to create a new entry).

A default witness wit1 has been created, with a default value $\top$ (e.g. the predicate "true") which we need to change. Its name will have to be x if we want it to be a witness for the parameter x of the corresponding abstract event in the machine `Celebrity_1`. The abstract event has the assignment $r := x$, while the concrete one has the assignment $r := b$. So the content of the witness is $\mathsf{x} = \mathsf{b}$. The event should now look as follows:

Event *celebrity* $\widehat{=}$
refines *celebrity*
  when
    grd1 : $R = \varnothing$
  with
    x : $\mathsf{x} = \mathsf{b}$
  then
    act1 : $r := b$

[6]The URL of the resource is: http://handbook.event-b.org/current/files/Celebrity.zip

end

Edit the content and save the file. One warning will disappear, and only two will remain.

Try completing the other two witnesses on your own. A hint: Both witnesses are simple equalities, and both can be found by comparing the third guard of the abstract event with the second guard of the concrete one. Remember to give the witness the name of the variable it stands for. If you completed this step correctly, there should be no warning, info or error left in the Rodin Problems View (3.1.2).

The following Section 2.9.2 shows the final `Celebrity_2` machine.

2.9.2 The Final Second Refinement

MACHINE Celebrity_2
REFINES Celebrity_1
SEES Celebrity_c0
VARIABLES
  `r`
  `R`
  `b`
INVARIANTS
  `inv1` : $R \subseteq P$
  `inv2` : $b \in P$
  `inv3` : $b \notin R$
  `inv4` : $Q = R \cup \{b\}$
EVENTS
Initialisation
  begin
    `act1` : $r :\in P$
    `act2` : $b, R : | b' \in P \wedge R' = P \setminus \{b'\}$
  end
Event *celebrity* $\widehat{=}$
refines *celebrity*
  when
    `grd1` : $R = \varnothing$
  with
    `x` : `b = x`
  then
    `act1` : $r := b$
  end
Event *remove_1* $\widehat{=}$
refines *remove_1*
  any
    x
  where
    `grd1` : $x \in R$
    `grd2` : $x \mapsto b \in k$
  with

```
      y : b = y
   then
      act1 : R := R \ {x}
   end
Event  remove_2 ≙
refines  remove_2
   any
      x
   where
      grd1 : x ∈ R
      grd2 : x ↦ b ∉ k
   with
      y : b = y
   then
      act2 : b := x
      act1 : R := R \ {x}
   end
END
```

2.9.3 The Celebrity algorithm

We take a brief tour through the model to see how the problem and algorithm is specified. The celebrity problem itself is described in the context Celebrity_c0. There are three constants. P is the set of persons, each represented by a number, c is the celebrity we are looking for and k is the "knows" relation between the persons. The axioms encode the properties about the "knows" relation that we stated above.

In the most abstract machine Celebrity_0 we specify what the algorithm should do. The variable r can be any person initially and the event celebrity finds the celebrity in one step. After the event celebrity occurred, r contains the result of the algorithm. You might ask yourself, what is the problem if we can just pick the celebrity and assign it to the result? The answer is that we wrote down our problem in a way that we know that there is a celebrity c and we specify the algorithm by stating it should return it. Later in the refinement we model how to find the celebrity without using c anymore. By the refinement relation we know then that the algorithm works correctly.

So let's have a look at the first refinement Celebrity_1. A variable Q is introduced which contains a subset of the persons, the potential celebrities. We start with Q being all persons. Two new events remove_1 and remove_2 are added that remove persons from Q who cannot be the celebrity. remove_1 removes a person that knows somebody while remove_2 removes a person that is not known by any other person. As an invariant we state that the celebrity is always in Q. When there is just one person left in the set, we know that this is the celebrity.

The second refinement Celebrity_2 then splits the potential celebrities Q into one arbitrary person – the candidate b – and the "rest" R. remove_1 then removes a person x from R if b knows x. remove_2 checks if there is a person x in R that does not know the candidate. If found, x is the new candidate b and is removed from the rest R. If R is empty, we know that the candidate is the celebrity.

The third refinement then makes some more assumptions about the given problem. The context Celebrity_c1 extends Celebrity_c0 and states that there are $n + 1$ persons with the numbers $0 \,..\, n$. Instead of having an abstract data structure like a set, the third refinement just introduces an index variable a that points to the first person of the rest. Instead of taking an arbitrary element from R as in the second refinement, the remove events just takes the first element a. a is then removed from R just by increasing it by one. When a is larger then n, R is empty and b contains the result.

This last refinement works only on the following three integer variables: The index a, the candidate b and the result r. Each event is deterministic and in every step only one event is enabled. The events together can be interpreted as an implementation of the algorithm:

```
r := 0                  // initialisation act1
a := 1                  // initialisation act2
b := 0                  // initialisation act3
while a ≤ n do          // guard in remove_1 and remove_2
  if a ↦ b ∈ k then     // guard in remove_1 and negated in remove_2
    a := a + 1          // action in remove_1
  else                  // a ↦ b ∉ k
    b := a              // action act1 in remove_2
    a := a + 1          // action act2 in remove_2
  end if
end while
r := b                  // action in celebrity
```

2.9.4 The First Proof

In this section, we prove the model of the Celebrity Problem. To do this, click on the box in the upper right hand corner that has a little plus sign to switch to the Proving Perspective. You can switch between perspectives using the shortcut bar as shown in Figure 2.17.

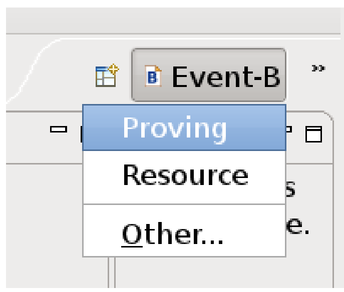

Figure 2.17: Switch Perspective

If the Proving Perspective is not available in the menu, select Other... 〉 Proving. This will open a new window which shows all available perspectives.

We should now see the window in Figure 2.18.

The `Proving Perspective` contains three new important views:

Proof Tree View (3.1.5) Here we see a tree of the proof that we have done so far and the current position in it. By clicking in the tree, we can navigate inside the proof. Currently, we have not started with the proof, so there is no new place to move to.

Proof Control View (3.1.5) This is where we perform interactive proofs.

Goal View (3.1.5) This window shows what needs to be proved at the current position inside the proof tree.

Expand the `Celebrity_1` machine in the Event-B Explorer. Then expand the Proof Obligations section. We can see that the auto prover (3.1.6) did quite a good job. Only three proofs have not been completed[7] (a completed proof is indicated by a green check).

Let's start with the proof remove_1/inv2/INV of `Celebrity_1`. To do this, double click on the proof obligation remove_1/inv2/INV. We should now see the window as shown in Figure 2.19.

[7] Interestingly enough, this number can vary: Provers can be configured in the preferences, and changes there can have an impact on the ability to automatically discharge proofs. In addition, all provers have time-outs. On a slow machine, some proof obligations may not get discharged, compared to a faster machine with the same timeout.

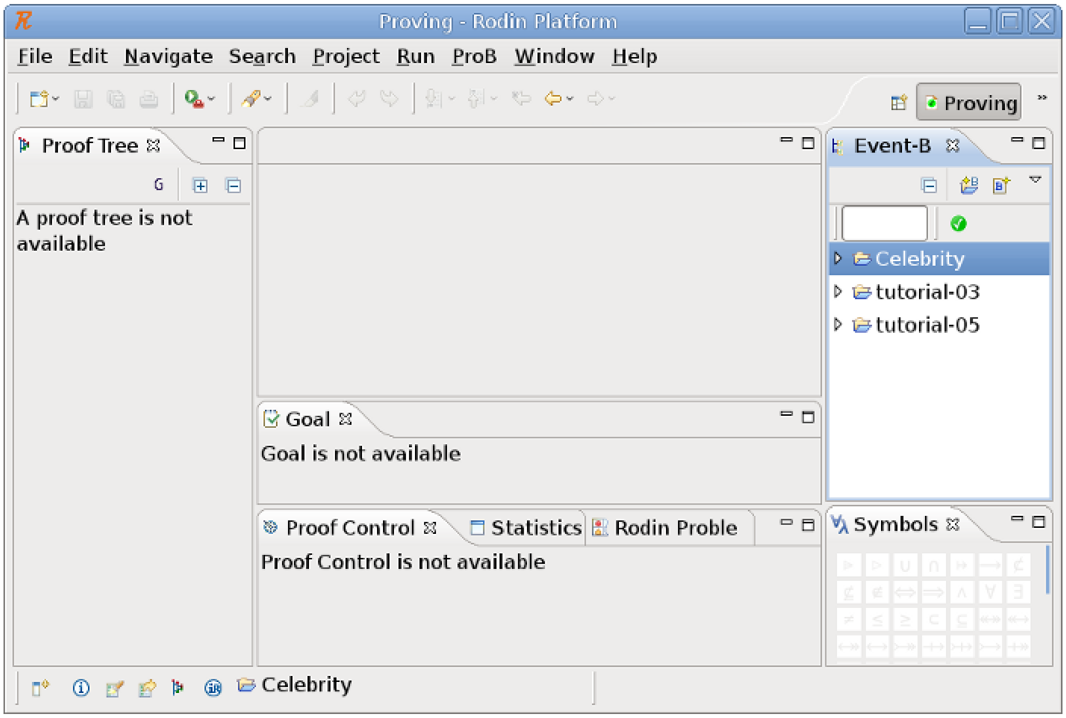

Figure 2.18: Rodin Proving Perspective

Make sure that you understand the different buttons in the Proof Control View (3.1.5).

What has to be proven is that the event `remove_1` preserves the invariant `inv2`, $c \in Q$. The event's action assigns to Q the new value $Q \setminus \{x\}$. Because we know that invariant $c \in Q$ held before the assignment, it suffices to prove that we do not remove c from Q, i.e. $x \neq c$. Type this into the Proof Control View (3.1.5) and press the ah button.

In order to revert a step, click on a node in the `Proof Tree View` and click on the button in the `Proof Control View` or open the context menu of a node and select `Prune`.

Take a look at the Proof Tree View. The root node should now be labeled with `ah` $(x \neq c)$, that is the hypothesis we just added. The node has three children: The first is the proof of well-definedness condition of $x \neq c$ which is just $\top$ and trivially proven. The second is the proof of the hypotheses $\neg x = c$. The third is the proof of the original goal where the new hypotheses can be used.

The new goal is $\neg x = c$. Now, try selecting the right hypotheses by yourself in order to complete the proof (Hint: What axiom states that the celebrity does not know anybody?). To do this, click on the button in the Proof Control View. On the left side we should see now the Selected Hypotheses View (see Figure 2.20). If you cannot find the right hypotheses, you may also just select all hypotheses. To add the selected hypothesis to the Selected Hypotheses View just click on the button.

Now click on the p0 button to prove the goal $\neg x = c$ with the Predicate prover on selected hypothesis. The goal should be discharged and in the Proof Tree you should see that the first two children of the root node are proven. The Proof Control View should now show the original goal $c \in Q \setminus x$ and $x \neq c$ is now one of our hypotheses. Click a second time on the p0 button in order to finalize the proof. The smiley in the Proof Control View should now become green indicating that all sequents of the proof tree are discharged as shown in Figure 2.21.

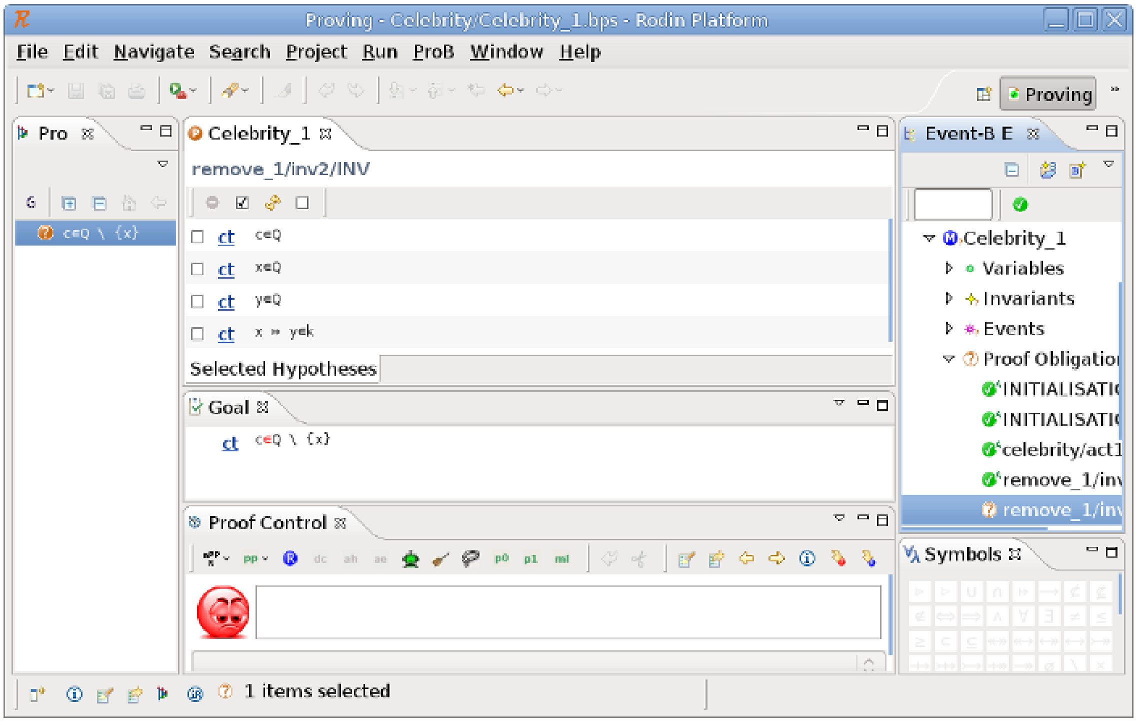

Figure 2.19: Proof Obligation

After saving the proof, the proof obligation remove_1/inv2/INV in the Event-B Explorer should also become a green chop.

There are alternative ways to prove the proof obligation. For instance, we can use the button to load those hidden hypotheses that contain identifiers in common with the goal into the Selected Hypotheses View, and we can also use it with the selected hypotheses.

In order to move to the next undischarged proof obligation, you may also use the Next Undischarged PO button ( ) of the Proof Control View (3.1.5). The next proof can be solved the same way as the last one.

As an exercise, try to prove `Celebrity_2`. A small hint: We have to fill in an existential quantifier. First, look in the list of hypothesis to see if you find any variable that is in P and select that hypothesis. Then instantiate b' and R' correctly (For instance, if you want to instantiate b' with c, then $P\backslash\{c\}$ is a good choice for R') by typing the instantiations in the Goal View (3.1.5) and then clicking on the red existential quantifier. Now all open branches of the proof tree can be proved with p0. After this, we have completed all the proofs, and the model is ready for use.

2.9.5 Proving — an Art or a Science?

Proving can be quite frustrating, both for beginners and advanced users. Especially beginners sometimes get the impression that proofing is just "clicking around" that sometimes works and sometimes doesn't. And when it works, it's not really clear why — the proof tree is also difficult to read, even for experienced users. We provided some additional guidance on provers in Section 3.4.5 that may be of help.

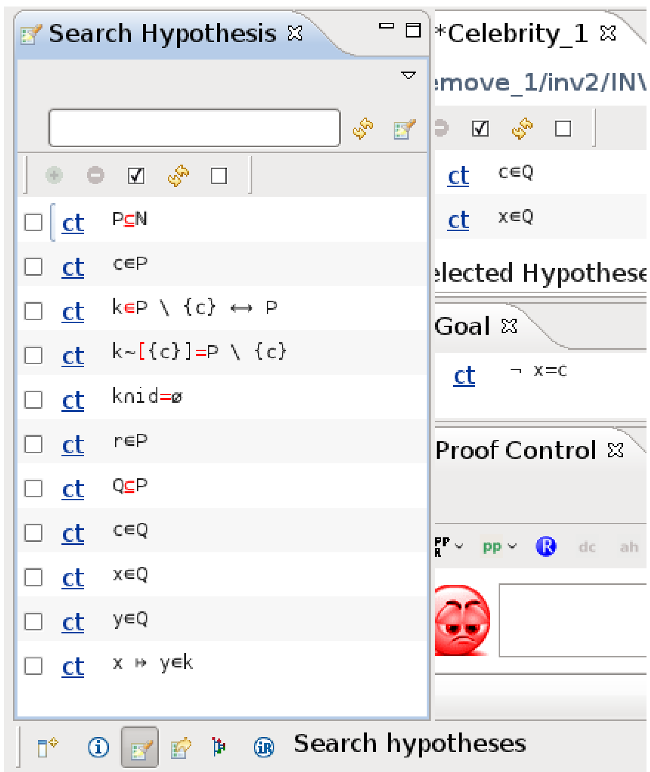

Figure 2.20: Search Hypothesis View

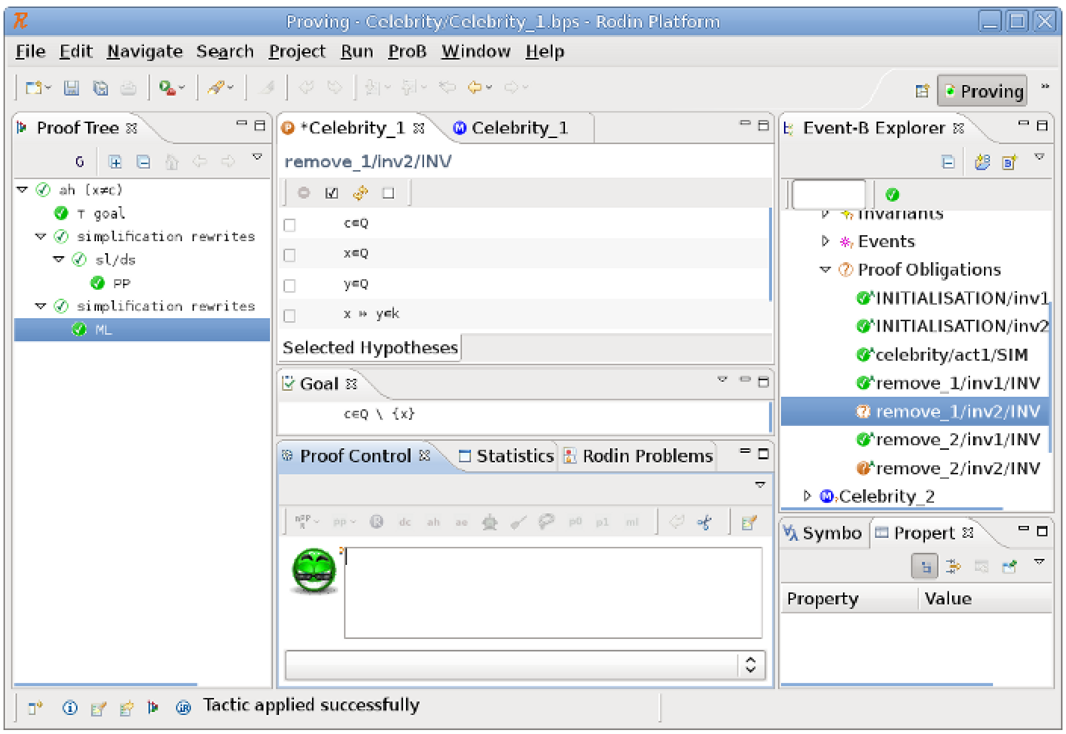

Figure 2.21: The green smiley indicates that all sequents of the proof tree are discharged

2.10 Location Access Controller

Goals: In this section, we will take a closer look at a few more complex proofs. For this we use the model of a location access controller. The goal is to develop the proofs that ensure there are no deadlocks present in the initial model and in the first refinement.

Through the model used in this section, we study a complete system and remind the proof rules of formal development. The system's task is to control the access of certain people to different locations of a site. The system is thus based on whether a person has (or does not have) access to a particular location.

Before describing the initial model, import the archive file Doors.zip[8] that contains the model. To do this, select File ⟩ Import ⟩ General ⟩ Existing Project into Workspace. Then select the according archive file and click on Finish. It will take Rodin a few seconds to extract and load all the files.

2.10.1 Initial Model

Let us look at the initial model which consists of `doors_ctx1` and `doors_0`. There are two carrier sets in the model context. One is for people (P) and the other is for locations (L). There is a location called outside ($outside$) and a relation (aut) which reflects where people are allowed to go. Everyone is permitted to go outside. The model machine has one event, pass, which changes the location of a person and one variable, sit, which denotes where a person is located.

Deadlock Freeness

Looking through the initial model, you will see that everything already has been proved. This is true, but Rodin does not do any deadlock freeness proof yet, so we will have to add them by ourself. A model is considered as deadlocked if the system reaches a state with no outgoing transitions. The objective of this section is to develop proofs for deadlock freeness for the initial model and for the first refinement.

Consider the event pass from the initial model:

EVENTS
Event *pass* $\widehat{=}$
  any
    p
    l
  where
    grd11 : $p \mapsto l \in aut$
    grd12 : $sit(p) \neq l$
  then
    act11 : $sit(p) := l$
  end
END

Since the initial model has only one event (pass), the system might deadlock when both guards of the event (grd11 and grd12) are false. In this case, to prove that no deadlocks can occur requires proving that someone can always change room. Clearly, we want to avoid this happening. We must therefore prove that the two guards are always true. To do this, add a new derived invariant (a theorem) to `doors_0` called DLF (click the `not theorem` button to make it switch to `theorem`) and change the predicate so that it is the conjunction of the two guards. The difference between a "normal" invariant and one that is marked as theorem is that is must be proven that the theorem always holds if the other invariants declared before hold. We do not need to prove that an event preserves the invariant marked as theorem because this is a logical consequence when it preserves the other invariants.

[8]The URL of the resource is: http://handbook.event-b.org/current/files/Doors.zip

INVARIANTS

DLF : $\exists p, l \cdot (p \mapsto l \in aut \land sit(p) \neq l)$

You can also use ProB to search for deadlocks. Right-click on the machine you want to check and start the animation with the "Start Animation / Model Checking" menu entry. After starting the animation, go to the Event View in the ProB perspective (see 2.9). There are two ways to search for deadlocks:

- Just press on the `Check` button and mark `Find Deadlocks` before starting the check by pressing the button `Start consistency checking`. ProB then systematically "executes" all events and tries to find a state where no event is enabled.
- An alternative is to select `Deadlock Freedom Checking` after clicking on the triangle right to the `Check` button. ProB then prompts you for an optional predicate. Just leave that empty for the start. The difference to the first alternative is that ProB searches now for variable values where all invariants are true but none of the guards.

This contribution requires the **ProB** plugin. The content is maintained by the plugin contributors and may be out of date.

Save the machine. We will see in the Event-B Explorer View that the auto-prover (3.1.6) fails to prove the theorem DLF/THM.

If you cannot find the proof obligation DLF/THM, maybe you forgot to mark the invariant as a theorem by clicking on the `theorem` button. Another reason that you don't see the proof obligation DLF/THM could be that you forgot to rename the invariant "DLF".

Let us analyze whether this is an inconsistency in the model. Switch to the `Proving Perspective` and double click on the proof obligation DLF/THM. In the Proof Control view, first disable the post-tactics (there is a small downward pointing arrow in the upper right hand corner above the toolbar, see Figure 2.22. Click on this arrow and make sure that the option Enable post-tactic is unchecked in the dropdown menu.) We turn off the post-tactics because we want to see the proof develop in its different stages. Then select the root node in the Prove Tree, right-click on it and select Purge. This removes any proof that might be already started by the auto-provers. By doing this we want to assure that we you have the same proof as in this tutorial.

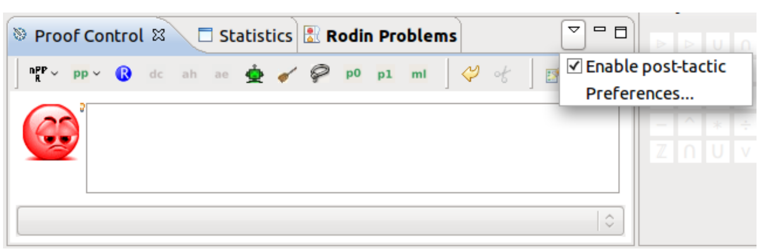

Figure 2.22: Disabling the proof post-tactics in the Proof Controlling View

In order to succeed with the proof, we need a pair $p \mapsto l$ that is in aut but not in sit. Having a look the axioms, we find axm4 of `doors_ctx1`, which states that there is a location l different from $outside$ where everyone is allowed to go:

AXIOMS

axm4 : $\exists l \cdot l \in L \setminus \{outside\} \land P \times \{l\} \subseteq aut$

So for every person p in P, $p \mapsto l$ and $p \mapsto outside$ are in **aut**. (In words: every person is allowed to go both to the outside and to a location l). The basic idea of our proof is that a person is either outside and can to the location l. When it is not outside, it can walk outside.

What we implicitly assumed was that there is actually a person, so we need to prove now is that P is non-empty. This holds since carrier sets are always non-empty, but is a bit hard to prove. Now add the hypothesis $\exists x.x \in P$ using the ah button after entering the predicate into the Proof Control text area. In the Proof Tree view you can now see three new nodes that have to be proven:

- $\top$ is the trivial well-definedness condition, just click on to prove it.
- $\exists x \cdot x \in P$ is the hypothesis that we introduced, just click on to prove it.
- $\exists p, l \cdot (p \mapsto l \in aut \land sit(p) \neq l)$ is the original goal but we can use the introduced hypothesis in the proof. We now continue with the proof of this goal.

Click on the existential quantifier of the new hypothesis $\exists x \cdot x \in P$ (appearing in the Selected Hypothesis view) as demonstrated in Figure 2.23. You see that it is automatically instantiated. That means that we can use x from now on in our proof as an example for a person and we have the new hypothesis $x \in P$.

An existential quantification can be proven by giving an example for the variables. First, we instantiate p in the goal with x: enter x in the yellow box corresponding to p in the Goal View and click on the existential quantifier as shown in Figure 2.24.

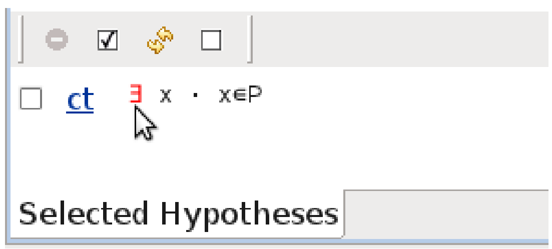

Figure 2.23: Click on the existential quantifier in order to ...

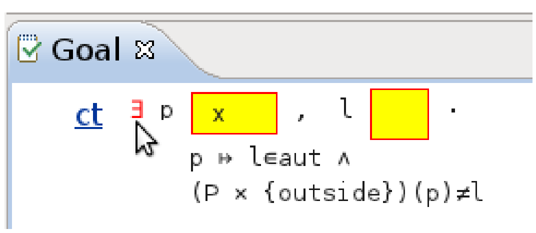

Figure 2.24: ... instantiate it, in this case by substituting x.

The instantiation produces two new nodes in the Proof Tree view. The first goal is the trivial well-definedness condition $\top$ and can be easily discharged by pressing . The remaining goal is $\exists l \cdot (x \mapsto l \in aut \land sit(x) \neq l)$ which results from the old goal by replacing p by x. You can see the the current proof tree in Figure 2.25. In the node ah we added the hypothesis, in $\exists$ hyp we instantiated x from a hypothesis and in $\exists$ goal we instantiated p in the goal.

Now we need an example for the remaining variable l. There are two situations we want to distinguish: The person x could be outside or not. To do this, type $sit(x) = outside$ into the Proof Control view and click on the button dc (dc for distinguish case). Again, you get three new goals.

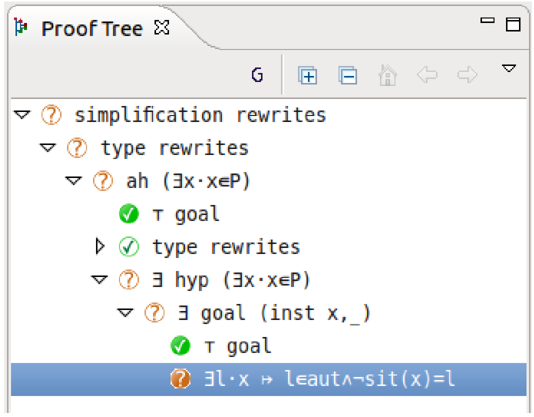

Figure 2.25: The proof tree after instantiating p with x.

- The first is the well-definedness condition of $sit(x) = outside$. sit must be a function and x in its domain. This is easy to prove since sit is a total function (3.3.5). Just press .
- Then second node has the original goal but $sit(x) = outside$ as a hypothesis.
- The third node has the original goal but $\neg sit(x) = outside$ as a hypothesis.

Let's continue with the case $sit(x) = outside$: When x is outside, it can always go to the l of axm4. To search for axm4, type *outside* into the Proof Control text field and click the button . Now click on the ∃ symbol in the axm4 (see Figure 2.26) to instantiate l. Now we have l as an example for a location which is not outside and where everybody can go. Our goal still is $\exists l \cdot x \mapsto l \in aut \wedge sit(x) \neq l$. Note that the existential

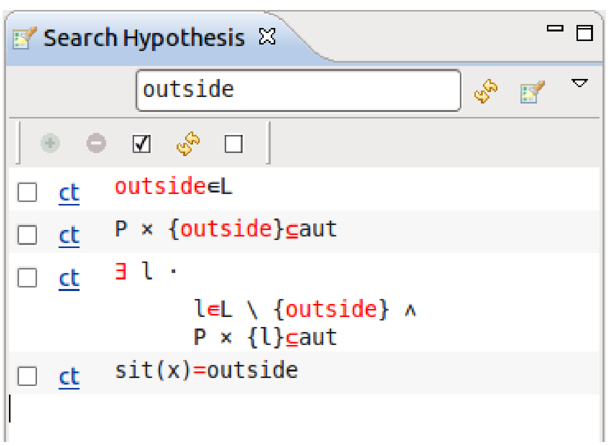

Figure 2.26: Searching hypothesis for *outside*: The third one is axm4.

quantification introduces a new l which has not (yet) anything to do with our location l where anybody can go. Now type l into the yellow box of the goal and press the ∃ symbol to state that we use our l as an example for the l in the existential quantification. Again, we have first the trivial goal ⊤ as well-definedness condition, just press . The remaining goal should be $\exists l \cdot x \mapsto l \in aut \wedge sit(x) \neq l$. This can be proven by the already selected hypothesis $sit(x) = outside$, $l \in L \setminus \{outside\}$ and $P \times \{l\} \subseteq aut$. Just press .

Now our second case of the case distinction remains where x is not outside ($sit(x) \neq outside$). Then x can just go outside. Again the goal is $\exists l \cdot x \mapsto l \in aut \wedge sit(x) \neq l$. Type *outside* as an example for a location

l into the yellow box and press the $\exists$ symbol. Press to discharge the trivial well-definedness condition $\top$. The new goal should be $x \mapsto outside \in aut \wedge sit(x) \neq outside$.

To prove this we need the information that x has the right to go $outside$. This is stated in the axiom $P \times \{outside\} \subseteq aut$. Have a look at the Search Hypothesis view. There should be still the result from the last search for $outside$. (If not, repeat the search by entering $outside$ into the Proof Control and press). Select $P \times \{outside\} \subseteq aut$ (in Figure 2.26, it's the second entry) and press to add it to your selected hypothesis. Then the auto-prover has enough information, just click and the last goal of our theorem should be proven.

We just summarize the proof. Compare this with your final proof tree (like in Figure 2.27).

- added hypotheses: $\exists x \cdot x \in P$
  - well-definedness condition $\top$: automatically proven
  - the hypotheses: automatically proven
  - instantiation of x in the hypotheses $\exists x \cdot x \in P$
    - using x as an example for the $\exists p \ldots$ in the goal
      - well-definedness condition $\top$: automatically proven
      - case distinction $sit(x) = outside$
        - well-definedness condition (sit is a function with x in its domain): automatically proven
        - first case: instantiation of l from axiom axm4
          - using l as an example for the $\exists l \ldots$ in the goal
            - well-definedness condition $\top$: automatically proven
            - automatically proven
        - second case: using $outside$ as an example for the $\exists l \ldots$ in the goal
          - well-definedness condition $\top$: automatically proven
          - hypotheses $P \times \{outside\}$ selected
            - automatically proven

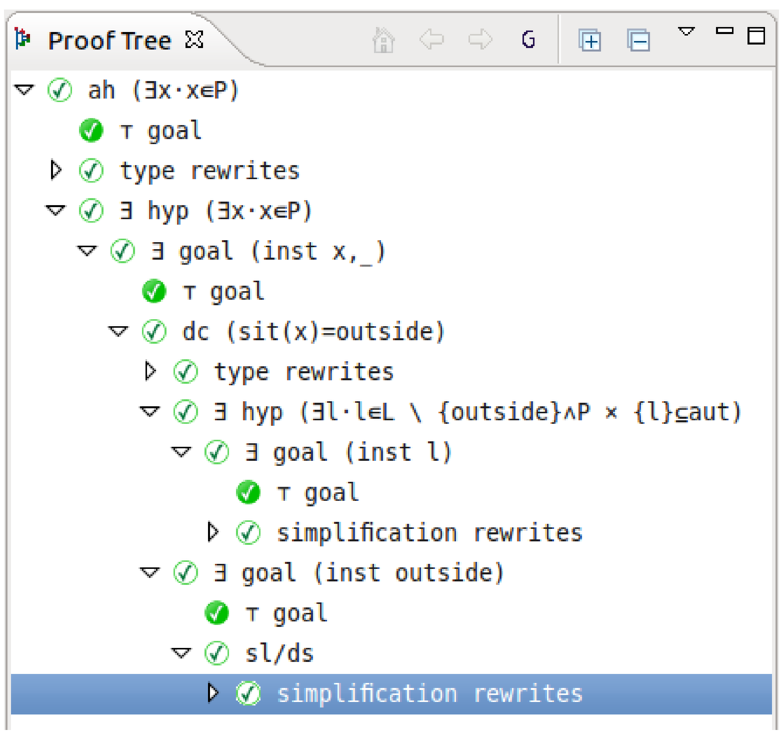

Figure 2.27: Searching hypothesis for $outside$: The third one is axm4.

2.10.2 First Refinement

Now we are going to explain the main complexity of our model: the deadlock freeness proof for the first refinement.

The difference between the first refinement and the initial model is that a new constant com been added in order to describe which rooms are connected. Additionally, we have a constant exit, which will be explained later.

The event INITIALISATION does not change, but the event PASS is refined as a consequence. We assume that a person can move to another location l if they have the authorization to be in l (already defined in the abstraction) and also if the location l is connected to the location p where the person is at this precise moment (represented by sit(p)).

We need to add a new guard that is more exact that that of the machine it refines:

$$(sit(p) \mapsto l \in com) \Rightarrow (sit(p) \neq l)$$

We are faced with a difficulty here; it is not possible to prove that the refined event PASS does not happen less often than the abstract event PASS of its predecessor. To prove this we would have to prove that the guard of the abstract event implies that of the concrete event.

The issue is that this condition cannot be verified in general. Moreover, the failure to prove the above condition indicates that there are possibilities that certain people could stay permanently blocked in locations. People are also more limited in the ways that they may move because they can only move between rooms that are connected.

Now we know that this model contains a problem ignored in the document requirements. We must now find a sufficient solution.

Please note that post-tactics should still be disabled before starting this part of the tutorial.

At the beginning of this section we need to come back to the Event-B Perspective. As described in Section 2.10.1, open door_1 machine and add a derived invariant (theorem) called DLF as follows:

INVARIANTS

DLF : $\exists q, m \cdot (q \mapsto m \in aut \wedge sit(q) \mapsto m \in com)$

Save the file. Once again, the prover fails to prove the deadlock freeness automatically. Actually all we want to prove is that "any person authorized to be in a location must also be authorized to go in another location which communicates with the first one".

This can be illustrated as demonstrated in Figure 2.28.

To begin with, switch over to the proving perspective and double click on DLF/THM to begin proving. At the beginning of the proof, there aren't any selected hypothesis, so we need to select a few. The old deadlock freeness invariant will be useful, and axm7 of doors_ctx2 as well.

AXIOMS

axm7 : $\exists l \cdot l \in L \setminus \{outside\} \wedge outside \mapsto l \in com \wedge P \times \{l\} \subseteq aut$

We will try to avoid using exit since we want to keep things as simple as possible. Because sit and aut are inside the invariant, we also are likely to need

$$sit \subseteq aut \wedge sit \in P \rightarrow L.$$

Once again, the prover automatically rewrites the existential quantifiers in the hypotheses. We now look at the proof. There is an easy case if $sit(p) = outside$. Add this case as previously using the Case Distinction button (dc). To do this, you first need to instantiate the value for p. To do this, use the hypothesis $\exists p, l \cdot p \mapsto l \in aut \wedge sit(p) \neq l$ and then click on the existential quantifier to create the expression $p \in P$ (see Figure 2.24). Initialize the value of q with the value of p (type p into the yellow box next to q). For m, you

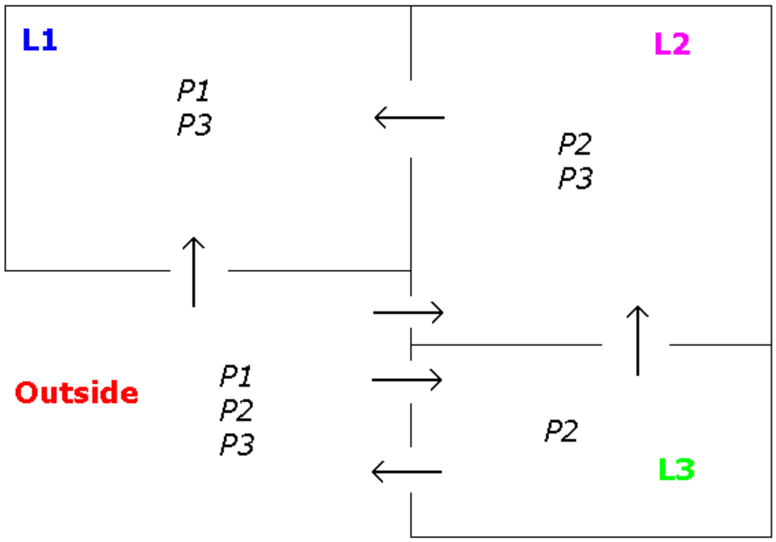

Figure 2.28: The floor plan of the location to be controlled.

will use the existential quantifier of axm7 of doors_ctx2 to instantiate m (add the axiom as a hypothesis and then click on the existential quantifier next to the l. Once the variable has been initialized, type it into the yellow box next to m).

For the other case, we will need the notion of **exit**. This function **exit** connects locations to locations and defines at every location except outside.

We can look at the axioms about exit:

AXIOMS

axm3 : $exit \in L \setminus \{outside\} \rightarrow L$

axm4 : $exit \subseteq com$

axm5 : $\forall s \cdot s \subseteq exit^{-1}[s] \Rightarrow s = \varnothing$

axm6 : $aut \rhd \{outside\} \subseteq (aut; exit^{-1})$

The axioms state that:

- (axm3) Every room except the outside has exactly one exit.
- (axm4) An exit must be a room that communicates with the current one
- (axm5) A chain of exits leads to the outside without any cycles or infinite paths
- (axm6) Everyone allowed in a room is allowed to go through its exit.

In our proof, we still need to show that anyone who is not outside can walk through a door. For this, axm5 is useless, so we add all hypotheses containing exit except for axm5. Now we have to instantiate q and m correctly and to conclude that the proof should not be too hard. Once again, for q, the choice p is obvious. But it is not quite as easy for m. Expressed in language, m must be the room behind the exit door of the room that p is currently in.

Try translating this into set theory. But do not worry if you get it wrong. You can still go back in the proof by right-clicking at the desired point in the proof tree and choose Prune in order to retry.

This concludes the tutorial. Please note that this tutorial gave you only an introduction to proving.

2.11 Outlook

Congratulations – if you made it this far, you should have a good foundation for getting some real work done with Rodin. In the following we'd like to provide you with a few pointers that will help you to make your work as efficient as possible.

Use the Reference Section and FAQ If you have a specific issue or if you quickly need to look something up, check the reference (3) and FAQ (4) of this handbook.

Online, PDF and Eclipse-Version of the Handbook There are three versions of this handbook. You can access it directly through Rodin by using the built-in help browser (Help 〉 Help Contents). The Eclipse-Version is useful because it can be used offline.

Use the Rodin Wiki The Rodin Wiki (1.1.2) contains the latest news regarding Rodin and a wealth of information that is not in the scope of this handbook. Be sure to check out it out.

Find useful Plugins There are many plugins available, so be sure to check them out. There is a good chance that they will make your life easier.

Subscribe to the mailing lists The wiki lists the existing mailing lists which include a list for users and for developers. We strongly recommend subscribing to the announcement list.

Rodin in Industry If you are considering using Rodin in an industrial setting, be sure to explore the testimonies from the Deploy (1.5) project, in which industrial partners describe their experiences with Rodin.

We wish you success in your modeling projects!

Chapter 3

Reference

3.1 The Rodin Platform

In this section, we describe the details of the tool platform, as it is presented to the user. You will find a description of all GUI elements that you may encounter.

3.1.1 Eclipse in General

From the Eclipse Website[1]: Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services.

From Wikipedia[2]: Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Perl, PHP, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy and Scheme. It can also be used to develop packages for the software Mathematica. The IDE is often called Eclipse ADT (Ada Development Toolkit) for Ada, Eclipse CDT for C/C++, Eclipse JDT for Java, and Eclipse PDT for PHP.

Eclipse provides the technical foundation of Rodin.

Project Constituents and Relationships

The primary concept in doing formal developments with the Rodin Platform is that of a project. A project contains the complete mathematical development of a Discrete Transition System. It is made of several components of two kinds: machines and contexts. Machines contain the variables, invariants, theorems, and events of a project, whereas contexts contain the carrier sets, constants, axioms, and theorems of a project. Figure 3.1 shows an overview.

We remind the reader of the various relationships existing between machines and contexts. This is illustrated in the following figure. A machine can be "refined" by another one, and a context can be "extended" by another one (no cycles are allowed in both these relationships). Moreover, a machine can "see" one or several contexts. A typical example of machine and context relationship is shown in Figure 3.2.

[1] http://www.eclipse.org/
[2] http://en.wikipedia.org/

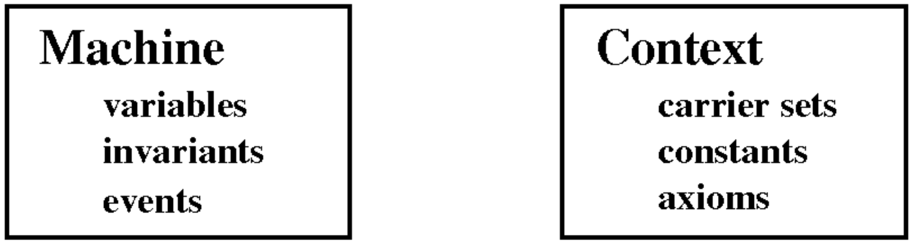

Figure 3.1: Overview Machine and Context

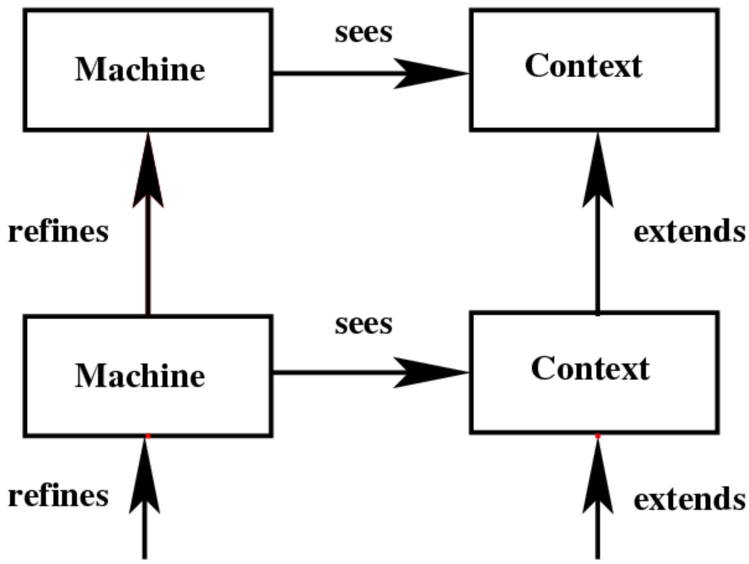

Figure 3.2: A typical example of machine and context relationship

Rodin Nature

Eclipse Projects can have one or more natures to describe their purpose. The GUI can then adapt to their nature. Rodin Projects must have the Rodin-Nature. If you create an Event-B project, it automatically has the right nature. If you want to modify an existing project, you can edit the `.project` file and add the following XML in the `<natures>` section:

```
<nature>org.rodinp.core.rodinnature</nature>
```

3.1.2 The Event-B Perspective

Figure 3.3 shows an overview of the opening window of the Event-B Perspective. The following subsections identify the different Rodin GUI elements (i.e. Views) which are visible and explain their functions.

Menu bar

The menu bar provides file and edit operations and other useful Event-B specific operations. We will briefly describe the most important menu items here.

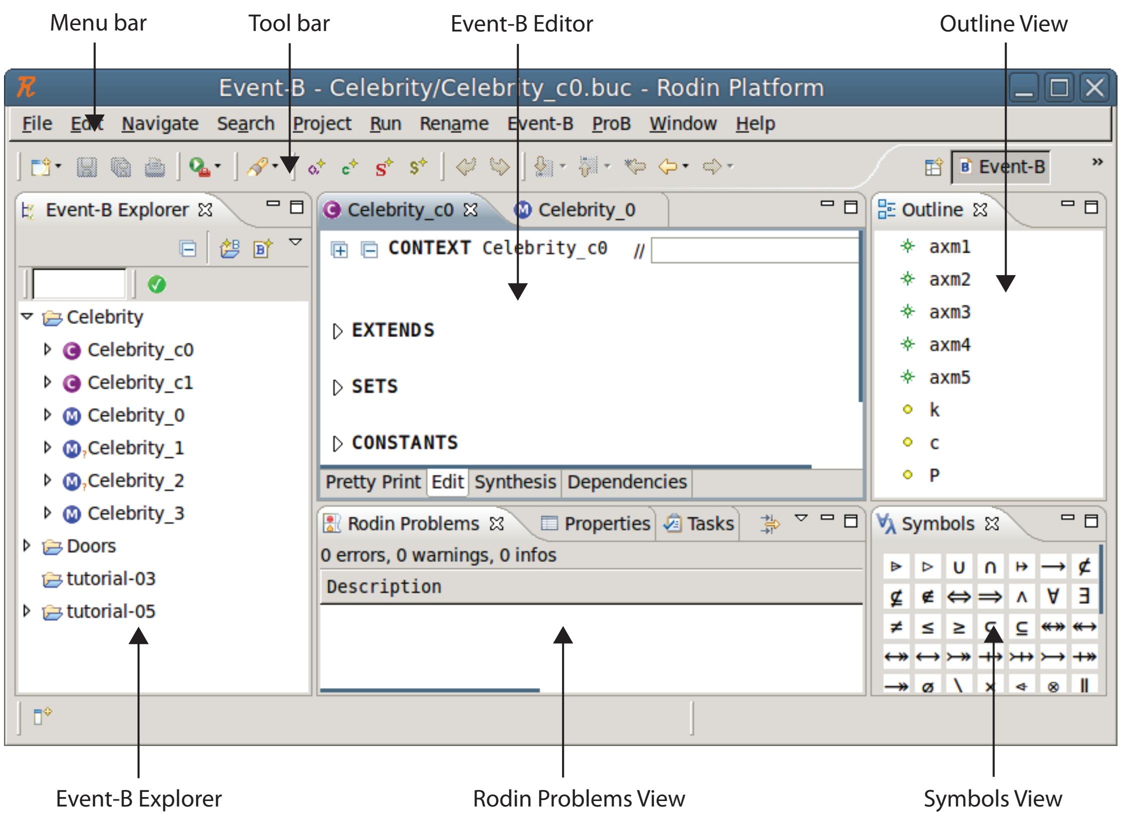

Figure 3.3: Overview of the Event-B Perspective

Rename menu When opening a machine or context file, the following actions for automatically renaming the Event-B model elements are available for the user:

One action is available when editing context files (see Figure 3.4).

- Automatic Axiom Labelling: this action will rename the axioms alphanumerically renaming according to their order of appearance.

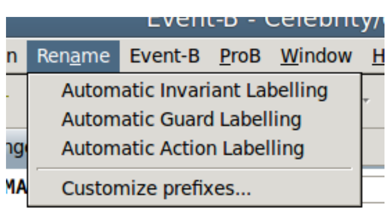

Figure 3.4: Automatic rename actions for machine files

Three actions are available for machine files (see Figure 3.5).

- Automatic Invariant Labelling: this action will rename the invariants alphanumerically according to their order of appearance.

- Automatic Guard Labelling: this action will rename the guards alphanumerically according to their order of appearance,
- Automatic Action Labelling: this action will rename the actions alphanumerically according to their order of appearance.

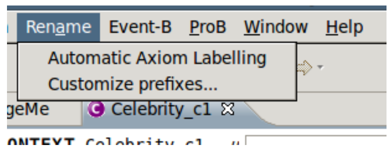

Figure 3.5: Automatic rename actions for context files

Event-B menu When opening a machine or context file, some wizards for creating Event-B model elements are available for the user. The different wizards are described in Section 3.1.4.

Tool bar

The tool bar provides short cuts for familiar commands like save, print, undo and redo. The tool bar also provides short cuts to the wizards for creating elements like axioms, constants, enumerated sets, etc., which are described in Section 3.1.4.

Editor View

The editor view contains the active Event-B editor which is described in Section 3.1.4.

Outline View

The outline view displays the outline of the active Event-B editor and lists elements like axioms, variables, etc..

Rodin Problems View

When the Static Checker discovers an error in a project, a marker is added to this project and to the faulty component in the Event-B Explorer. The error itself is shown in the Rodin Problems view (i.e. syntax errors) of the active Event-B editor.

By double-clicking on the error statement, you are transferred automatically into the place where the error has been detected so that you can correct it easily.

Symbols View

The symbols view is intended to give users a convenient way to type in mathematical symbols into the various model editors. If an editor is open and a text field is active (the cursor is blinking), then clicking a symbol inserts it at the current position as demonstrated in Figure 3.6.

The ASCII code corresponding to the symbol over which the mouse hovers is also displayed as a tooltip so that the user can also learn how to input symbols directly.

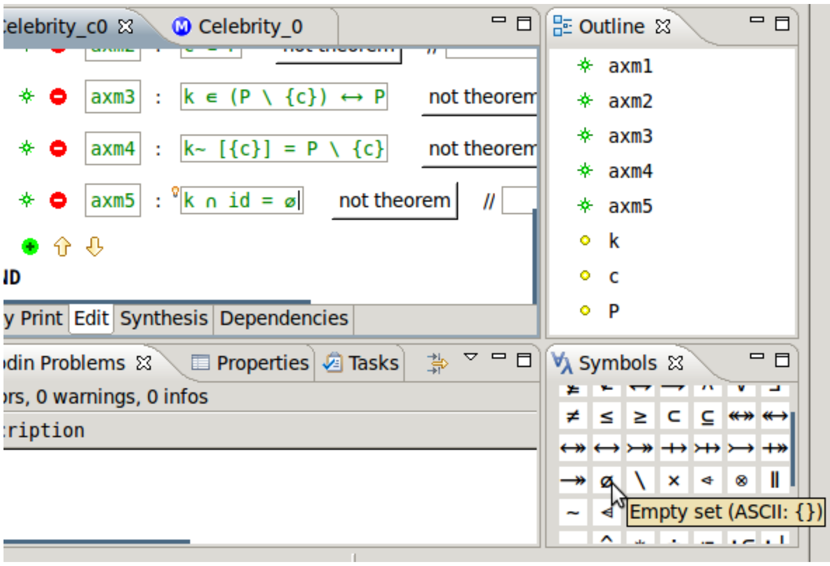

Figure 3.6: Clicking a symbol inserts it at the current position

Event-B Explorer

Projects can be found in the RODIN platform in the Event-B Explorer. This is usually situated on the left hand side of the screen as shown in Figure 3.3. The Event-B Explorer contains a list of name of the current projects. Next to each project name is a small triangle. By pressing it, one can expand a project and see the machines and contexts that it contains.

The icons (or) next to the components help identify whether they are a context or machine respectively.

When expanding a machine or a context, you can explore its elements. Double clicking on a specific element (i.e. a variable) opens the Event-B editor and marks the position of the variable in the machine or context as shown in Figure 3.7. Furthermore, proof obligations are displayed when clicking on the small triangle next to the label Proof Obligations (for more information see Section 3.1.5).

3.1.3 Customizing a perspective suitable for RODIN

So far, you have needed two different perspectives to work with RODIN. But it actually is possible to work with only one perspective. In this section, we try to customize a perspective so that we do not need any other. If you have experience with customizing Eclipse perspectives, you may only want to read the next paragraph which contains a few thoughts about a good perspective for RODIN.

As a start, we should think about what we want the perspective to look like. The proving perspective already is quite nice, but we may want to use a little bit more editing space when in the Event-B perspective. To create more space, we could move all windows that currently are on both sides of the editing area onto one side, as they never really need to be used simultaneously. For a bit more space, we could dock all of these windows onto the so-called Fast View bar so that they disappear when they are not needed. It would also be nice to be able to split the screen and work on several compontents at once. For example, we could have an abstract machine on one side of the editing surface, and the concrete machine on the other.

For the most part, the perspectives can be customized by dragging and dropping the different windows. First of all, you need to find the Fast View Bar. Usually, it is at the bottom end of the Eclipse window, but it also can be on the side or hidden inside the Shortcut Bar. For our purposes, it probably is best to have it on the right side of the screen. Place it there by dragging it with the mouse. Now, add some items to it. To

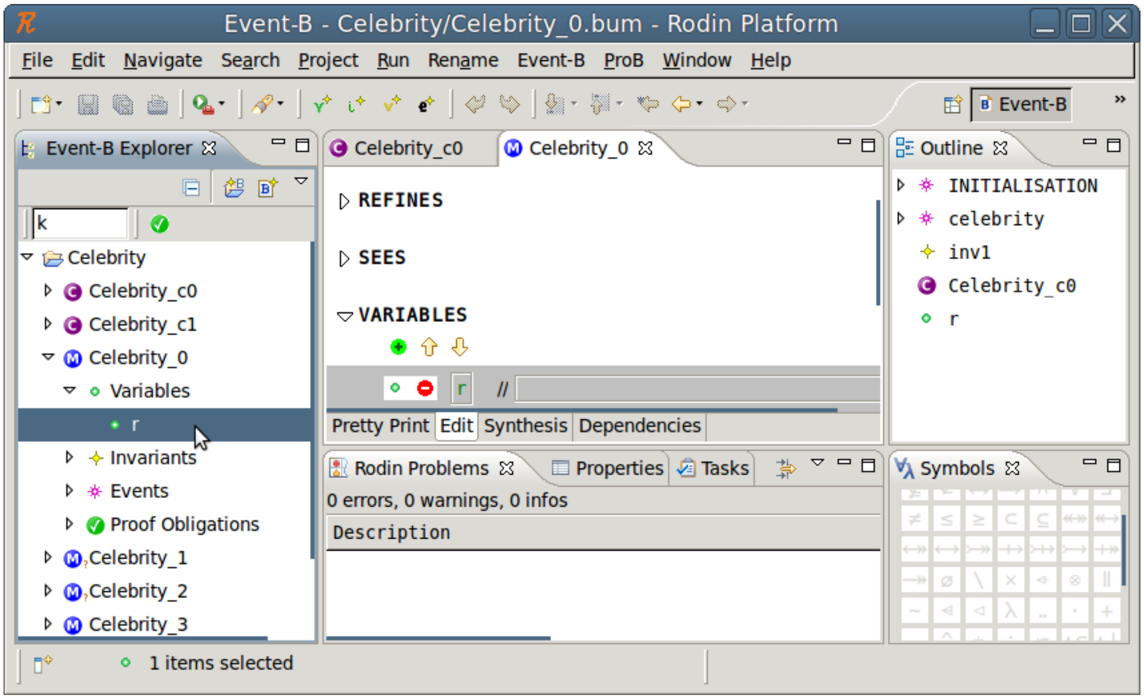

Figure 3.7: Double clicking on an element opens the Event-B editor and marks the corresponding position

do this, press the New Fast View button on the bar. It might be better to leave the Goal, the Problems and the Proof Control window at the bottom of the screen, as you may want them to stay open while editing. A good selection of tools to add to the Fast View bar may be:

- Project Explorer
- Obligation Explorer
- Search Hypothesis
- Cache Hypothesis
- Proof Tree
- Proof Information
- Progress Window

All of the windows that you cannot create directly when clicking on New Fast View can be found under Others/General. Once you are done, the window should look like in Figure 3.8. Click on "Save Perspective As" in the Window menu to save the perspective.

3.1.4 The Event-B Editor

Once a context or a machine is created, a window appears in the editing area as shown in figure 3.9.

By default, you are in the Edit mode which allows you to edit the modelling elements of the context which are the dependencies (keyword "EXTENDS"), the carrier sets (keyword "SETS"), the constants (keyword "CONSTANTS"), and the axioms (keyword "AXIOMS"). By pressing the triangle (▷) next to each

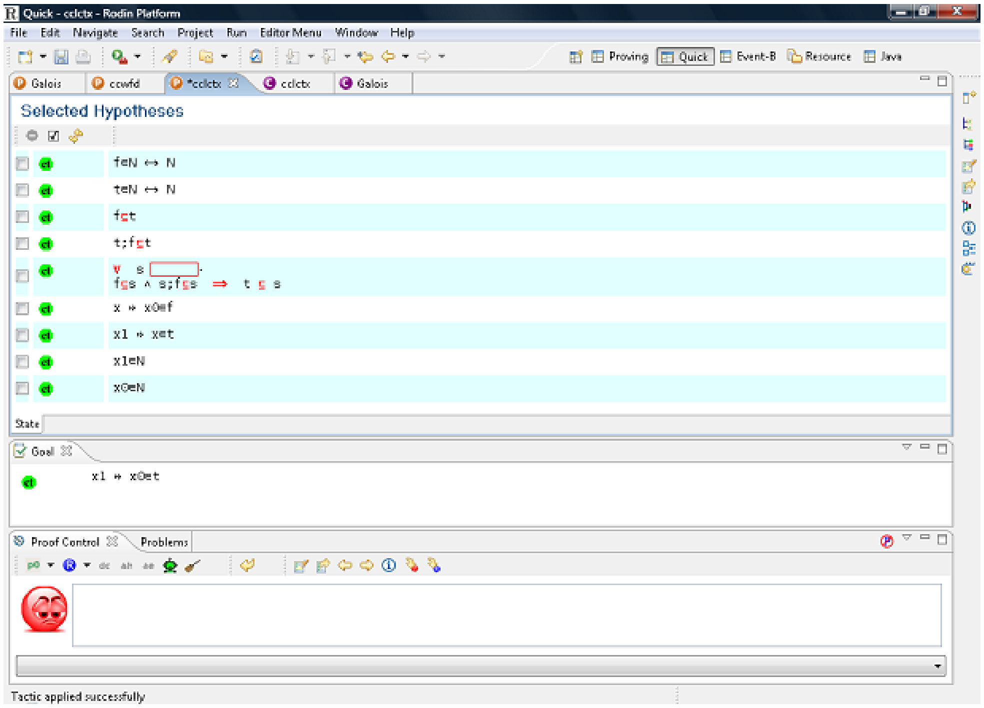

Figure 3.8: Our self-made Quick perspective

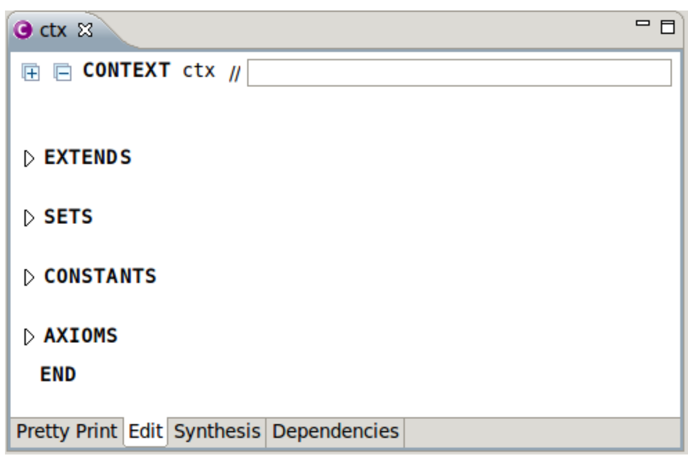

Figure 3.9: The Event-B editor

keyword, you can see the different modelling elements and also add, remove, or move them. Figure 3.10 shows what the section looks like after pressing the triangle next to the keyword "AXIOMS".

By pressing the button, you can add a new modelling element. For instance, clicking on the button next in the AXIOMS section will add a new axiom element. You can now enter a new axiom and a comment in the corresponding boxes as indicated in Figure 3.11.

To remove a modelling element, press the button. You can also move an modelling element up or down by selecting it and then pressing one of the two arrows (or).

It is also possible to add modelling elements by using wizards. You can active the different wizard by

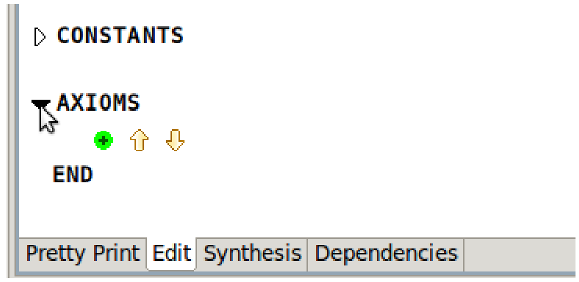

Figure 3.10: By pressing the triangle you can collapse/expand context sections

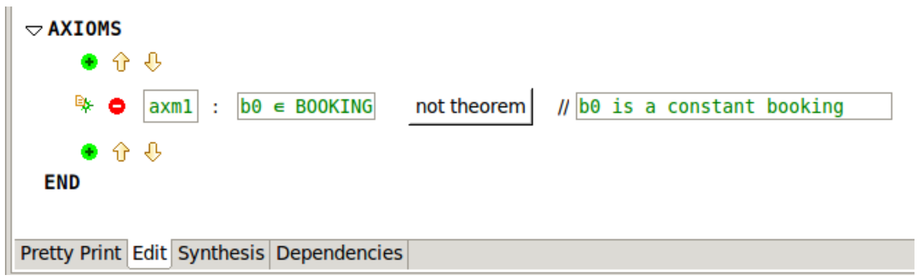

Figure 3.11: Adding a new modelling element

using the buttons in the tool bar as shown in Figure 3.12 and in Figure 3.13 or via the Event-B menu (keeping in mind that the wizards depend on the active file machine or context). The next sections explain how to use the wizards.

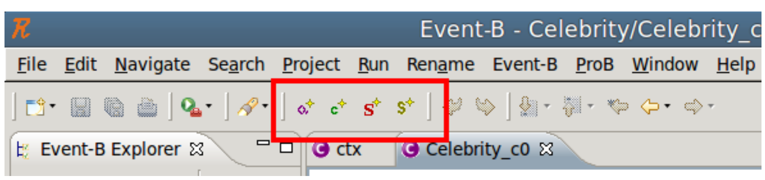

Figure 3.12: Wizards for context specific elements located in the tool bar

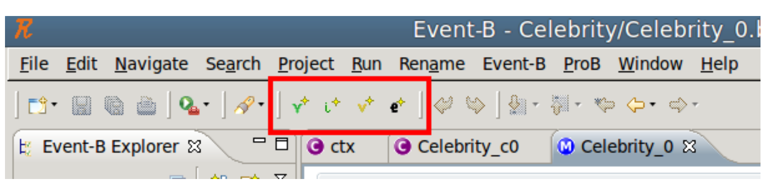

Figure 3.13: Wizards for machine specific elements located in the tool bar

New Carrier Sets Wizard

To activate the New Carrier Sets Wizard, press the button located in the tool bar. Pressing the button bring up the window shown in Figure 3.14.

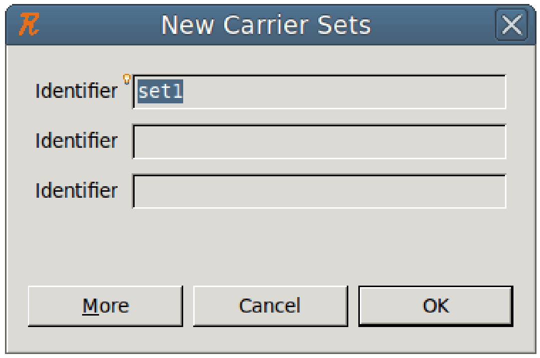

Figure 3.14: New Carrier Sets Wizard

You can enter as many carrier sets as you want by pressing the More button. When you are finished, press the OK button.

New Enumerated Set Wizard

To activate the New Enumerated Set Wizard, press the button located in the tool bar. Pressing the button bring up the window shown in Figure 3.15.

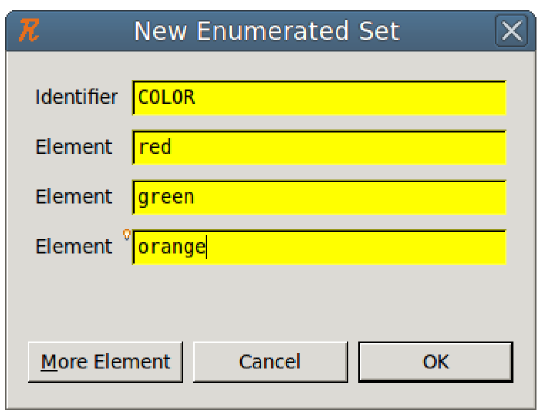

Figure 3.15: New Enumerated Set Wizard

Enter the name of the new enumerated set as well as the names of its elements. By pressing the More Elements button, you can enter additional elements. When you're finished, press the OK button. The benefit of using this wizard is that in addition to creating the set and its elements, the wizard automatically creates the axiom that is necessary for the context to work. For example, when you add the new carrier set `COLOUR` and the three constants `red`, `green`, and `orange`, the wizard automatically creates the following axiom $partition(COLOUR, \{red\}, \{green\}, \{orange\})$.

New Constants Wizard

To activate the New Constants Wizard, press the button located in the tool bar. Pressing the button will bring up the window shown in Figure 3.16.

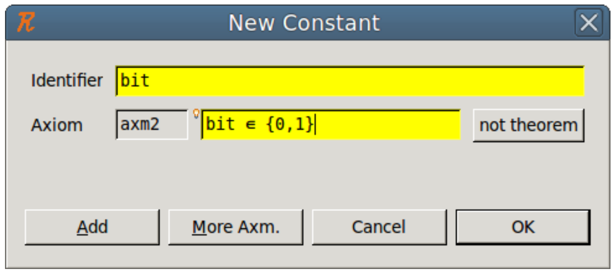

Figure 3.16: New Constants Wizard

You can then enter the names of a constant and an axiom which can be used to define the constant's type. By pressing the More Axm. button you can enter additional axioms. To add more constants, press the Add button. When you have finished, press the OK button.

New Axioms Wizard

To activate the New Axioms Wizard, press the button located in the tool bar. Pressing the button brings up the window shown in Figure 3.17.

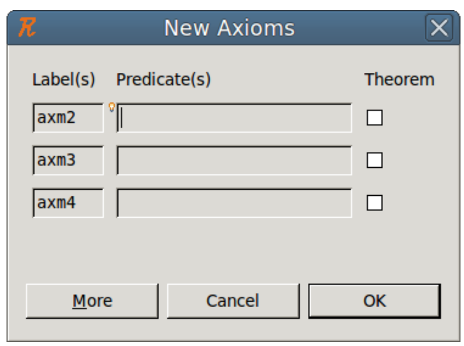

Figure 3.17: New Axioms Wizard

You can then enter the axioms you want. If more axioms are needed, press the More button. When you are finished, press the OK button.

Check the "Theorem" checkbox to indicate that the corresponding axiom should be a theorem.

New Variable Wizard

To activate the New Variable Wizard, press the button located in the tool bar. Pressing the button brings up the window shown in Figure 3.18.

You can then enter the names of the variable, what its state at initialization should be, and an invariant which defines its type. By pressing button More Inv., you can enter additional invariants. For adding more variables, press the Add button. When you're finished, press the OK button.

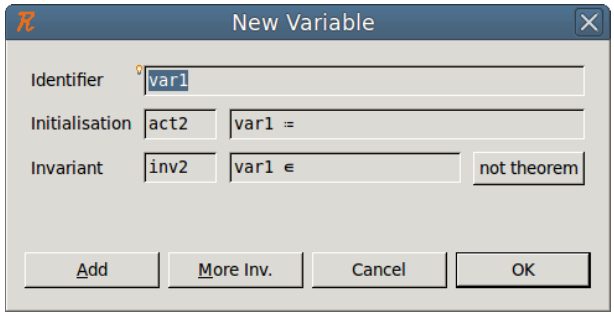

Figure 3.18: New Variable Wizard

New Invariants Wizard

To activate the New Invariants Wizard, press the button located in the tool bar. Pressing the button bring up the window shown in Figure 3.19.

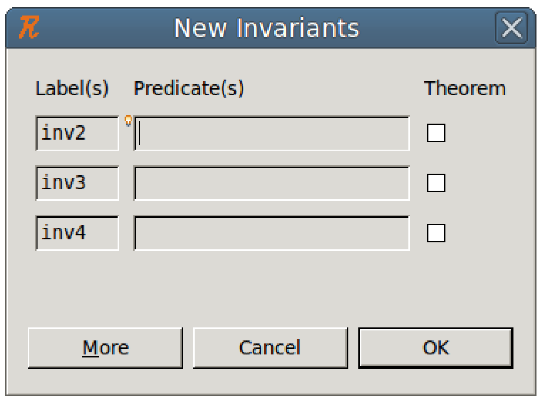

Figure 3.19: New Invariants Wizard

You can then enter the invariants you want. If more invariants are needed, press the More button. Check the Theorem checkbox to indicate that the corresponding invariant should be a theorem.

New Event Wizard

To activate the New Events Wizard, press the button located in the tool bar. Pressing this button brings up the window shown in Figure 3.20.

You can then enter the events that you want. As indicated, the following elements can be entered: name, parameters, guards, and actions. More parameters, guards and actions can be entered by pressing the corresponding buttons. If more events are needed, press the Add button. Press the OK button when you're finished.

Note that an event with no guard is considered to the guard *true*. Also, an event with no action is considered to have the action *skip*.

Dependencies (Context)

By selecting the Dependencies tab at the bottom of the Event-B editor, you obtain the window as shown in Figure 3.21.

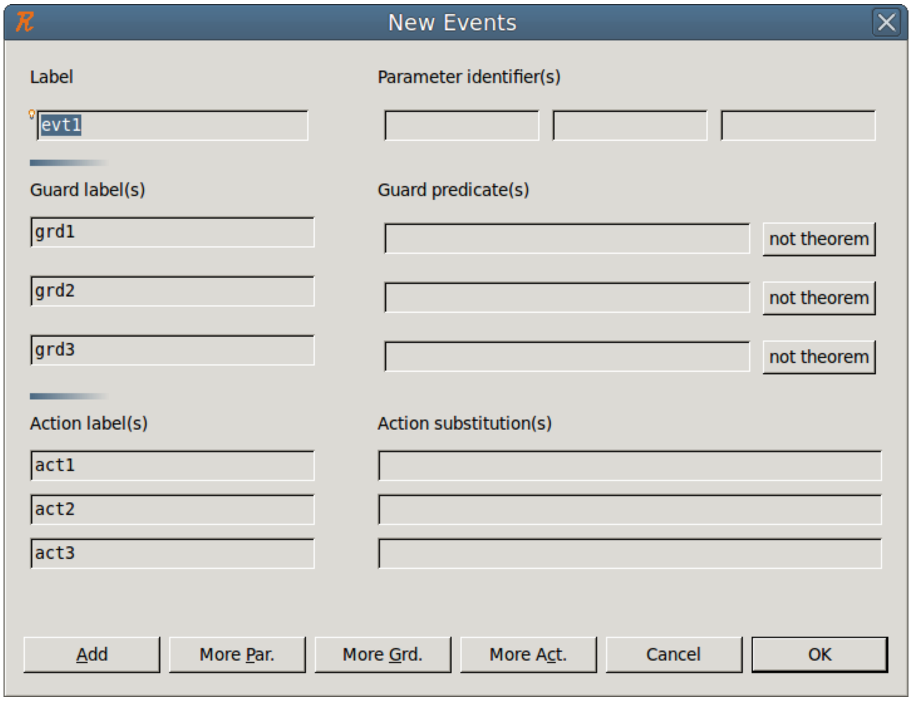

Figure 3.20: New Event Wizard

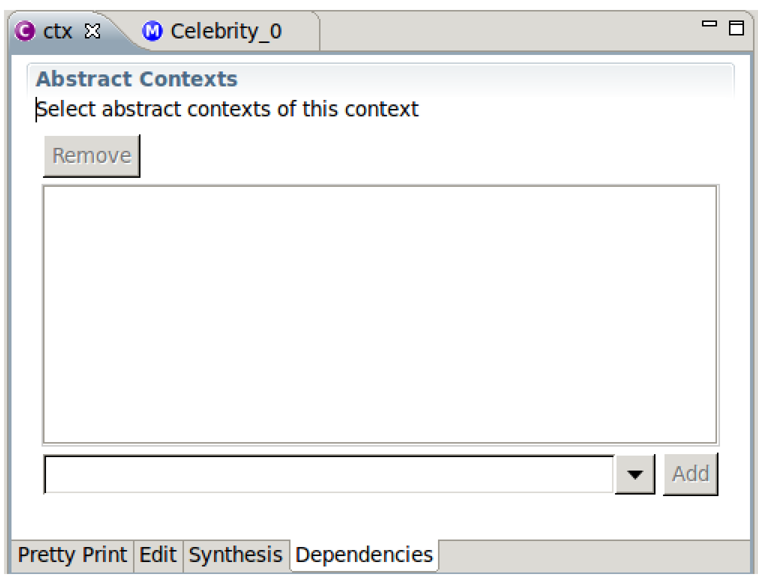

Figure 3.21: Dependencies tab of the Event-B editor

The dependencies tab allows you to control what other contexts that the current context is extending. To add the context that you want to extend, select the name of the context from the drop-down menu at the bottom of the window and then hit the Add button.

There is also another way to create a new context as an extension existing one. Select the context in the project window and then press the right mouse key. Then select Extend from the menu that opens up. This should bring up the window as shown in Figure 3.22.

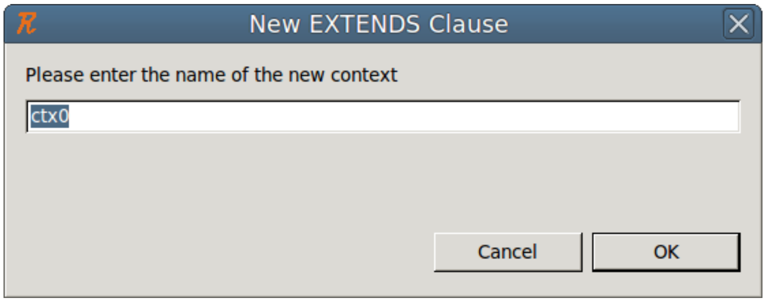

Figure 3.22: New EXTENDS Clause window

You can then enter the name of the new context which will automatically extend your chosen context.

Dependencies (Machine)

The Dependencies tab for machines is very similar to the one for contexts that is described in the previous section. The main difference is that there are two kinds of dependencies that can be established: machines on which the current machine depends are listed in the upper part and contexts on which the current machine depends are listed in the lower part.

Synthesis (Context)

Selecting the Synthesis tab brings up a global view of your context's elements (carrier set / constant / axiom / extended context) as demonstrated in Figure 3.23.

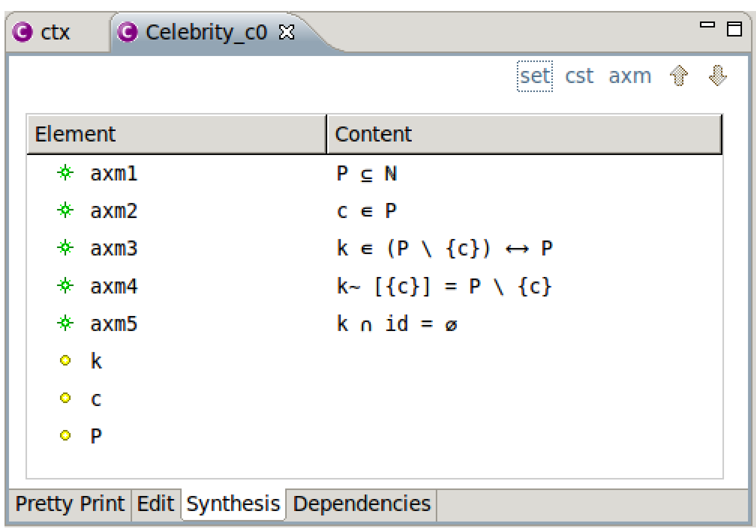

Figure 3.23: The Synthesis tab of the Event-B editor

By pressing the set, cst, or axm buttons in the upper right corner, you can filter out the carrier sets, constants or axioms of your context respectively.

If you select an element, you can change its priority by pressing the ⇧ button or the ⇩ button. You do this for axioms, carrier sets, constants and extended contexts.

Right clicking in this view will bring up a contextual menu that allows you to add a new carrier set, constant, axiom or extended context.

Synthesis (Machine)

The Synthesis tab for machines is very similar to the one of contexts (see above) except that you have a global view of your machine's elements (refined machine/seen context/variable/invariant/event/variant).

Pretty Print

Selecting the Pretty Print tab gives you a global view of your context or machine as if it had been entered through with an input text file as seen in Figure 3.24.

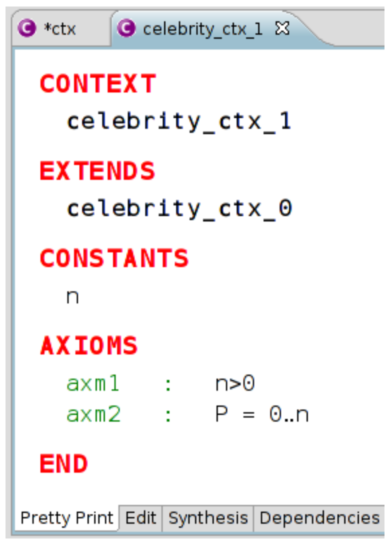

Figure 3.24: The Pretty Print tab of the Event-B editor

3.1.5 The Proving Perspective

When proof obligations (POs) (Section 3.2.6) are not discharged automatically, the user can attempt to discharge them interactively using the Proving Perspective. This page provides an overview of the Proving Perspective and its use. If the Proving Perspective is not visible as a tab on the top right-hand corner of the main interface, the user can switch to it via Window ⟩ Open Perspective.

The Proving Perspective consists of a number of views: the Proof Tree, the Goal, the Selected Hypotheses, the Proof Control, the Search Hypotheses, the Cache Hypotheses and the Proof Information. In the discussion that follows we look at each of these views individually. Figure 3.25 shows an overview of the Proving Perspective.

Loading a Proof

To work on an PO that has not yet been discharged, it is necessary to load the proof into the Proving Perspective. To do this, switch to the Proving Perspective. Select the project from the Event-B Explorer and select and expand the component (context or machine). Finally, select (double-click) the proof obligation of interest. A number of views will be updated with details of the corresponding proof.

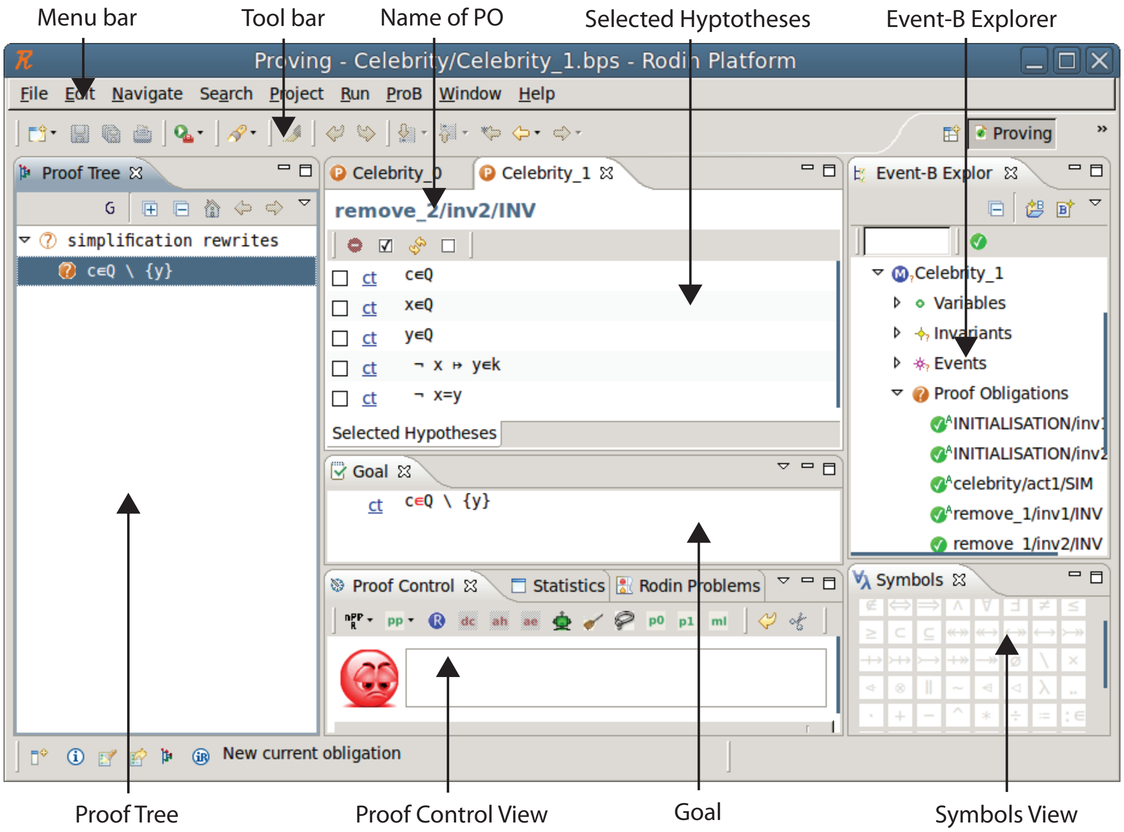

Figure 3.25: Overview of the Proving Perspective

Note that pressing the ✓ button on the top left hand side of the Event-B Explorer will remove all discharged proof obligations (PO's) from the view.

The Proof Tree

The proof tree view provides a graphical representation of each individual proof step as seen in figure 3.26.

The tree is made up of sequents. A line of the tree is shifted to the right when the corresponding node is a direct descendant of the node of the previous line. Each sequent is labeled with a comment which indicates which rule has been applied or which prover discharged the proof. By selecting a sequent in the proof tree, the hypotheses of the sequent are loaded to the Selected Hypotheses window, and the goal of the sequent is loaded to the Goal view.

Decoration The symbol to the left of the leaf indicates the state of the sequent:

- ✓ indicates that this sequent is discharged.
- ? indicates that this sequent is not discharged.
- R indicates that this sequent has been reviewed.

Internal nodes in the proof display the symbols in reverse colours. Note that a "reviewed" sequent is one that is not discharged yet by the prover. Instead, it has been "seen" by the user who decided to postpone

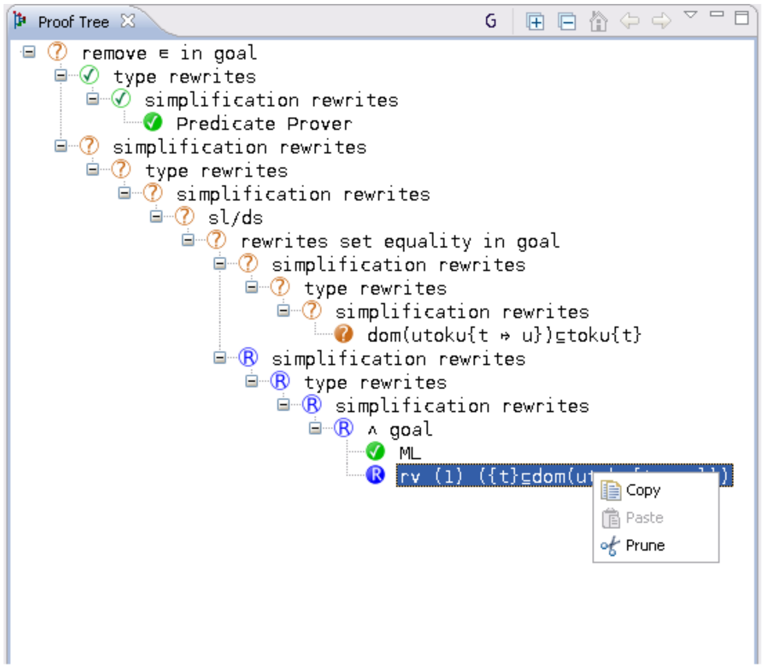

Figure 3.26: The Proof Tree

the proof. Marking sequents as "reviewed" is very convenient since the provers will ignore these leaves and focus on specific areas of interest. This allows the proof to be discharged interactively in a gradual fashion. In order to discharge a "reviewed" sequent, select it and prune the tree at that node: the node will become "brown" again (undischarged), and you can now try to discharge it.

Navigation within the Proof Tree There are three buttons on the top of the proof tree view:

- G allows you to see the goal of corresponding to each. node,
- allows you to fully expand the proof tree.
- allows you to fully collapse the tree; only the root stays visible.

Manipulating the Proof Tree

Hiding The button next to each node in the proof tree allows you to expand or collapse the subtree starting at that node.

Pruning The proof tree can be pruned at a selected node. This means that the subtree of the selected node is removed from the proof tree. The selected node becomes a leaf and displays the symbol . The proof activity can then be resumed from this node. After selecting a node in the proof tree, pruning can be performed in two ways:

- by right-clicking and then selecting Prune,
- by clicking on the button in the proof control view.

Note that after pruning, the post-tactic is not applied to the new current sequent. The post-tactic should be applied manually, if required, by clicking on the post-tactic button in the Proof Control view. This is especially useful when you want to redo a proof from the beginning. The proof tree can be pruned at its root node and then the proof can proceed again with invocation of internal or external provers or with an interactive proof.

Before pruning a particular node, the node (and its subtree) can be copied to the clipboard. If the new proof strategy subsequently fails, the copied version can be pasted back into the pruned node (see the next section).

Copy/Paste By selecting a node in the proof tree and then right-clicking with the mouse, you can copy the part of the proof tree starting at that sequent (including the node and its subtree). Pasting the node and subtree back in is done in a similar manner with a right mouse click on a proof node. This allows reuse of part of a proof tree in the same or even in another proof.

Goal and Selected Hypotheses

Each sequent in the proof tree have corresponding hypotheses and goals. A user will work with one selected sequent at a time by attempting various strategies in an effort to show that the goal is true. The Goal and Selected Hypotheses views provide information to the user about the currently selected sequent. Figure 3.27 shows an example.

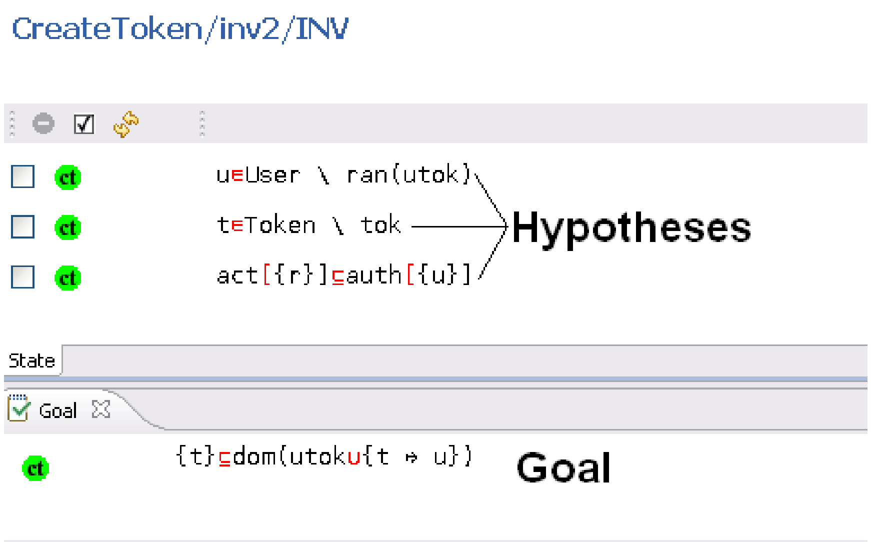

Figure 3.27: Open proof obligation

A hypothesis can be removed from the list of selected hypotheses by selecting the check the box situated next to it (you can click on several boxes at once) and then by clicking on the button at the top of the selected hypotheses window.

Note that the deselected hypotheses are not lost. You can find them again using the Search Hypotheses button in the Proof Control view. Other buttons are used as follows:

- ☑ - Select all hypotheses.
- - Invert the selection.
- next to the goal - Proof by contradiction 1: The negation of the goal becomes a selected hypothesis and the goal becomes "⊥".

- next to a selected hypothesis - Proof by contradiction 2: The negation of the hypothesis becomes the goal and the negated goal becomes a selected hypothesis.

A user wishing to attempt an interactive proof has a number of proof rules available, and these may either rewrite a hypothesis/goal or infer something from it. In the Goal and the Selected Hypotheses views, various operators may appear in red coloured font. The red font indicates that some interactive proof rule(s) are applicable and may be applied as a step in the proof attempt. When the mouse hovers over such an operator, a number of applicable rules may be displayed; the user may choose to apply one of the rules by clicking on it. Figure 3.28 shows an example.

Other proof rules can be applied when green buttons appear in the Goal and Selected Hypotheses views. Examples are proof by contradiction and conjunction introduction .

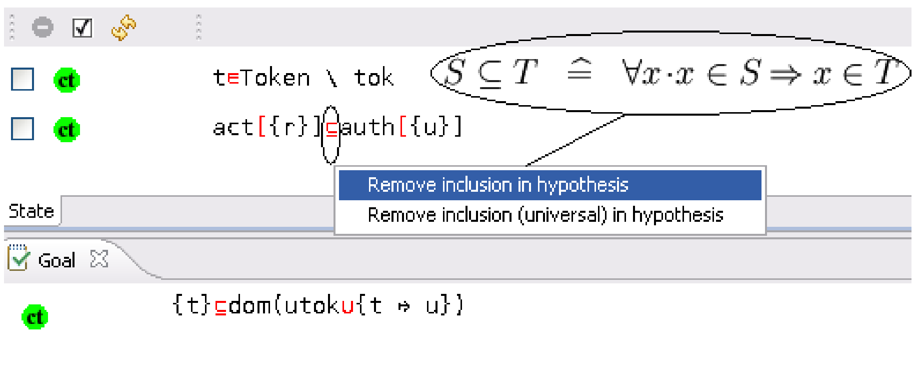

Figure 3.28: Applying a rule

To instantiate a quantifier, the user enters the desired expression in the box behind the quantifier and clicks on the quantifier (the red ∃) as demonstrated in figure 3.29.

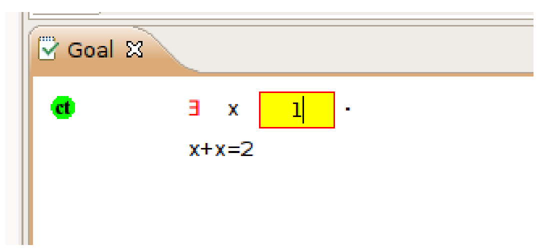

Figure 3.29: Instantiating a quatifier

The Proof Control View

The Proof Control view contains the buttons with which you can perform an interactive proof. An overview of this proof can be seen in Figure 3.30.

The following buttons are available in the Proof Control view:

- invokes the new predicate prover. A drop-down list indicates alternative strategies.
- indicates that a node has been reviewed. In an attempt by the user to carry out sequents in a certain order, they might decide to postpone the task of discharging some sequents until a later stage. To do

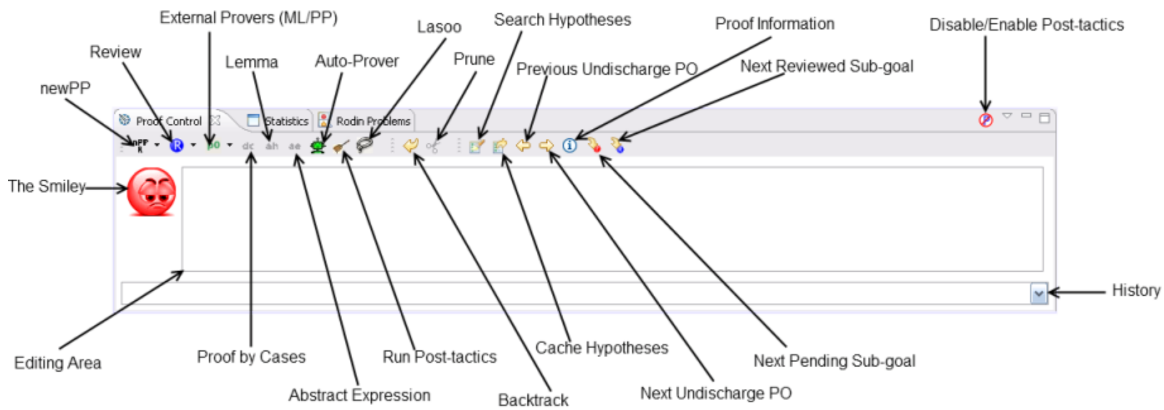

Figure 3.30: The Proof Control View

this, the sequent can be marked as reviewed by choosing the correct node and clicking on this button. This indicates that by visually checking the sequent, the user is convinced that they can discharge it later, but they do not want to do it right now.

- external provers can be invoked from the drop-down list to test sequents using different forces.
- begins a proof by cases. The proof is split into two branches. In the first branchf, the predicate supplied by the user is added to the Selected Hypotheses, and the user attempts to discharge this branch. In the second branch, the predicate supplied by the user is negated and added to the Selected Hypotheses. The user then attempts to discharge this branch.
- adds a new hypothesis. The predicate in the editing area should be proved by the user. It is then added as a new selected hypothesis.
- adds an abstract expression. The expression in the editing area is given a fresh name.
- invokes the auto-prover which attempts to discharge the goal. The auto-prover is applied automatically on all proof obligations after a the machine or context is saved. Using this button, you can invoke the auto-prover within an interactive proof.
- executes the post-tactics.
- loads the hidden hypotheses that contain identifiers in common with the goal and with the selected hypotheses into the Selected Hypotheses window
- backtracks from the current node (i.e., prunes its parent).
- prunes the proof tree from the node selected in the proof tree.
- finds hypotheses containing the character string in the editing area and displays them in the Search Hypothesis view.
- displays the Cache Hypotheses view. This view displays all hypotheses that are related to the current goal.

- loads the preceding undischarged proof obligation.
- loads the next undischarged proof obligation,
- displays information regarding the current proof obligation in the corresponding window. This information corresponds to the elements that directly took part in the generation of the proof obligation (events, invariant, etc.).
- moves to the next pending node of the current proof tree,
- loads the next reviewed node of the current proof tree.

You can also disable/enable post-tactics which allows you to decide if post-tactics should run after each interactive proof step. In addition, you can open the preferences for post-tactics. For this, open the menu of the Proof Control view via the little triangle on the top right corner of the view.

The Smiley The smiley can be one of three different colours: (1) red indicates that the proof tree contains one or more undischarged sequents, (2) blue indicates that all undischarged sequents of the proof tree have been reviewed, (3) green indicates that all sequents of the proof tree are discharged.

The Editing Area The editing area allows the user to supply parameters for proof commands. For example, when the user attempts to add a new hypothesis (by clicking on the ah button), the new hypothesis should have been written by the user in the editing area.

ML/PP and Hypotheses

ML ml (mono-lemma) prover appears in the drop-down list under the button (pp) as M0, M1, M2, M3, and ML. The different configuration (e.g., M0) refer to the proof force of the ML prover. All hypotheses are passed to ML.

PP pp (predicate prover) appears in the drop-down list under the button (pp) as P0, P1, PP.

- p0 uses all selected hypotheses (the ones in Selected Hypotheses window).
- p1 performs a lasoo operation on the selected hypotheses and the goal and uses the resulting hypotheses.
- pp uses all hypotheses.

3.1.6 Auto Prover

The auto-prover can be configured by means of a preference page, which can be obtained as follows: press the "Window" button on the top tooolbar. On the coming menu, press the "Preferences" button. On the coming menu, press the "Event-B" menu, then the "Sequent Prover", and finally the "Auto-Tactic" button. This yields the window in Figure 3.31.

On the left part you can see the ordered sequence of individual tactics composing the auto-prover, whereas the right part contains further tactics you can incorporate in the left part. By selecting a tactic you can move it from on part to the other or change the order in the left part.

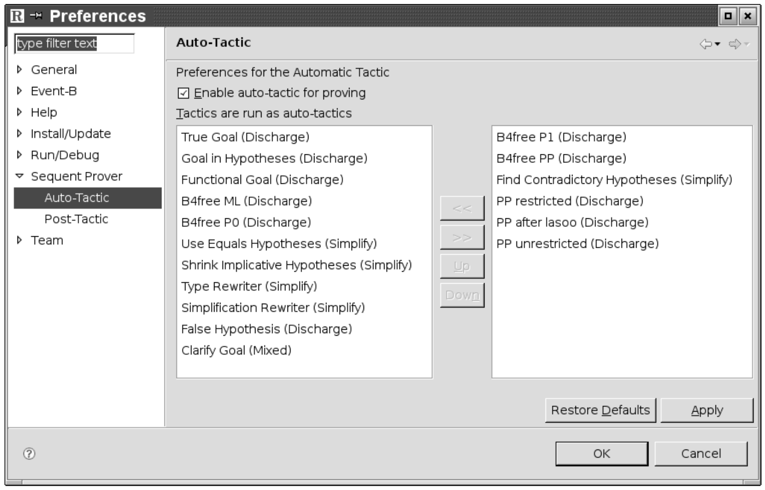

Figure 3.31: Auto Prover Preferences

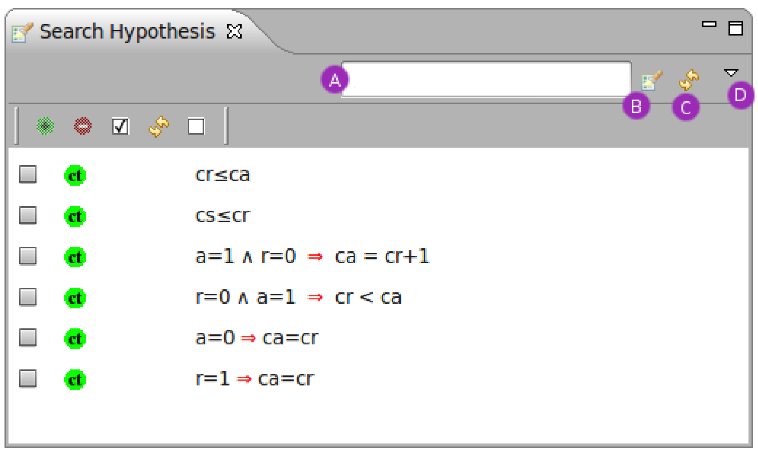

Figure 3.32: The Search Hypotheses View

The Search Hypotheses View

By typing a string in the Proof Control view and pressing the Search Hypotheses ( ) button, a window is provided which contains all the hypotheses that have a character string in common with the one entered by the user in the editing area. For example, if we search for hypotheses involving the character string "cr", then after pressing the Search Hypothesis ( ) button on the Proof Control view, we obtain the windows as shown in Figure 3.32.

This view also integrates a "quick search" area (A), that allows us to search quickly hypotheses involving short character strings such as "cr", a search hypothesis button (B) that behaves the same as the button of the proving window, a refresh button (C) that updates the window manually for more control, and a drop down menu (D) to set the preferences of the view up.

By pressing return key or the button (B) (once a short string has been entered into the input area (A)), hypotheses can be searched quickly as if we used the Proof Control as described before.

The drop down menu (D) allows some preferences about the searched hypotheses to be set.

If we change preferences for the search, we might need to manually "update" the view with the button

(C). By selecting the "Consider hidden hypotheses in search" option, we can review all hypotheses that have been unselected in the Selected Hypotheses window.

To move hypotheses to the Selected Hypotheses window, select the wanted hypothesis (using the check boxes) and press the button. Adding these hypotheses to the selected hypotheses means that they will be visible to the prover. They can then be used during the next interactive proof phase.

To remove hypotheses from the Search Hypotheses window, use the button. This button also appears above the selected hypotheses and allows the user to remove any hypothesis from the Selected Hypotheses window.

The other button, situated to the left each hypotheses, is the button. Clicking on this will attempt a proof by contradiction. The effect is the same as if the hypothesis were in the Selected Hypotheses window.

The Cache Hypotheses Window

This window allows the user to keep track of recently manipulated (i.e., used, removed, or selected) hypotheses for any PO. For example, when the user applies a rewrite to a hypothesis, a new hypothesis (after the rewriting) is selected, and the original hypothesis is deselected and put in the Cache Hypotheses window.

Similar operations (to that of the Selected Hypotheses and Search Hypotheses windows) such as removing, selecting and proof by contradiction (ct) are also available for the cached hypotheses. Interactive proof steps (e.g., rewriting) can also be carried out from the Cache Hypotheses window.

Proof Information View

This view displays information so that the user can relate a proof obligation to the model. For example, typical information for an event invariant preservation includes the event as well as the invariant in question. For instance in Figure 3.33, the hyperlinks CreateToken and inv2 can be used to navigate to the containing machine.

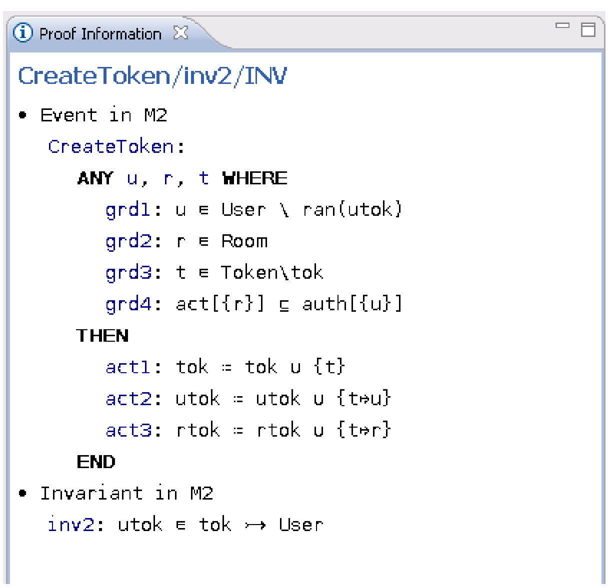

Figure 3.33: Proof Information View

Rule Details View

This view displays the information relating to a given proof tree node onto which a rule was applied. A command is available when right-clicking on a proof tree node in order to reveal the Rule Details view (See Figure 3.34).

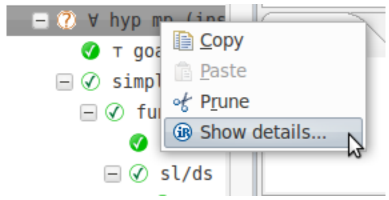

Figure 3.34: Open Rule Details View

By default, this view is a fast view. If the window does not appear, seek the button (identified by the view's icon) at the bottom left of the workbench to make this view visible.

Figure 3.35 gives an overview of the Rule Details View.

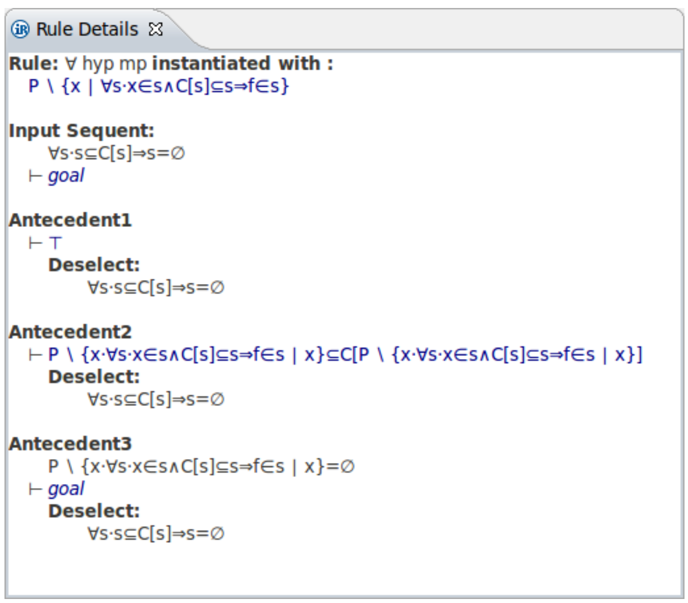

Figure 3.35: Rule Details View

A quick summary of the applied rule contents is provided. In this figure, we display the contents of the rule named ∀ hyp mp, where an input has been used to instantiate an hypothesis. One can quickly see the input that was used to instantiate the rule (the line below Rule: ∀ hyp mp instantiated with:), and the hypothesis that was considered by this rule (the line below Input Sequent:). Furthermore, it is possible to view the antecedents created by this rule in details (i.e. child proof tree nodes) and the actions performed on the hypotheses (selection, deselection, etc.).

Auto-tactic and Post-tactic

The auto-tactic automatically applies a combination (i.e. ordered list) of rewrite tactics, inference tactics and external provers to newly generated proof obligations. However, they can also be invoked by the user by clicking on the button in the Proof Control view. Note that the automatic application of the auto-prover can be quickly toggled with the Prove Automatically menu item available from the Project menu (See Figure 3.36).

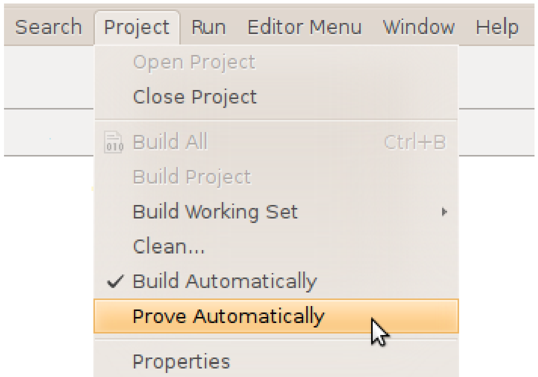

Figure 3.36: Toggle auto-prover via project menu

The post-tactic is also a combination of rewrite tactics, inference tactics and external provers and is applied automatically after each interactive proof step. However, it can also be invoked manually by clicking on the button in the Proof Control view.

Note that the post-tactic can be disabled quickly by clicking on the little arrow (marked with an A) of the Proof Control view (right upper corner) and then on Disable post-tactic option (B) as shown in figure 3.37.

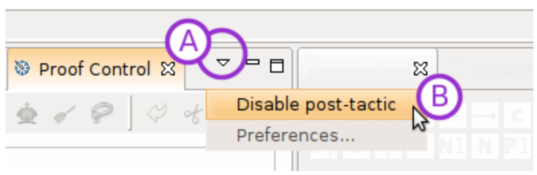

Figure 3.37: Proof Control view menu

Principles The ordered list of rewrite tactics, inference tactics and external provers that should be applied is called a profile. There are two default profiles. One is for auto-tactics and one is for post-tactics. These default profiles are immutable in time but can be duplicated for further modification by the user. The user can edit a profile and select it to run as automatic or post tactic. The idea is to have a set of available tactic profiles to be used as needed. Moreover, these edited profiles are shipped with projects if defined at the project level or can be imported or exported if defined at a workspace level. This makes it easy to share the profiles.

There are two ways to run the automatic or post tactics:

- Manually by clicking on the button or the button in the Proof Control view to launch the auto-tactic (with the selected auto-tactic profile) and the post-tactic (with the selected post-tactic profile) respectively.
- Automatically if such preference is activated. (Auto-tactic will then run after each proof step and post-tactic will run after each step and rebuild). Post-tactics and auto-tactics only need to be activated in order to run automatically (see previous sections on how to activate auto- and post-tactic).

The user can separately define tactic profiles and assign them to post and auto tactics. Therefore, there are two tabs in the Auto/Post Tactic preference page for each of these choices. These tabs will be described in Section 3.1.7.

3.1.7 Preferences

Rodin provides several options to set preferences the Event-B editor. You can access the preferences via Window 〉 Preferences 〉 Event-B in the menu bar. The following subsections describe the different preference option.

Appearance

This section provides settings for the Event-B editor appearance.

Colours and Fonts The colour and fonts preference page allows you to set the text and comment colour of the Event-B editor. Furthermore, it allows to toggle on or off the borders of the different fields in the Event-B editor. Figure 3.38 shows the Colours and Fonts preference page.

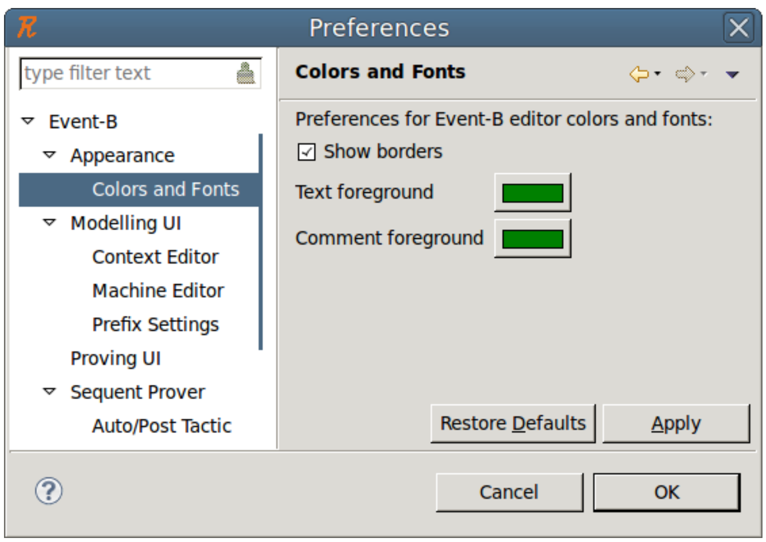

Figure 3.38: Colours and Fonts preferences

Context and Machine Editor

The context and machine editor preference pages allows you to customize the visible tabs of the context and machine editor.

Prefix Settings

This page describes the mechanism used to set element prefixes and perform renaming using dedicated actions for both machines and contexts. Note that prefixes are used for automatic renaming when elements should be alphanumerically ordered as well as when new elements are created.

Figure 3.39 shows that modifying prefixes on the workspace level or on the project level will affect the names used at creation of new Event-B elements. One can see that the prefixes for variables and invariants, which were originally set to “var” or “inv”, have been replaced by “variable” and “invariant”. New elements are then named using those prefixes.

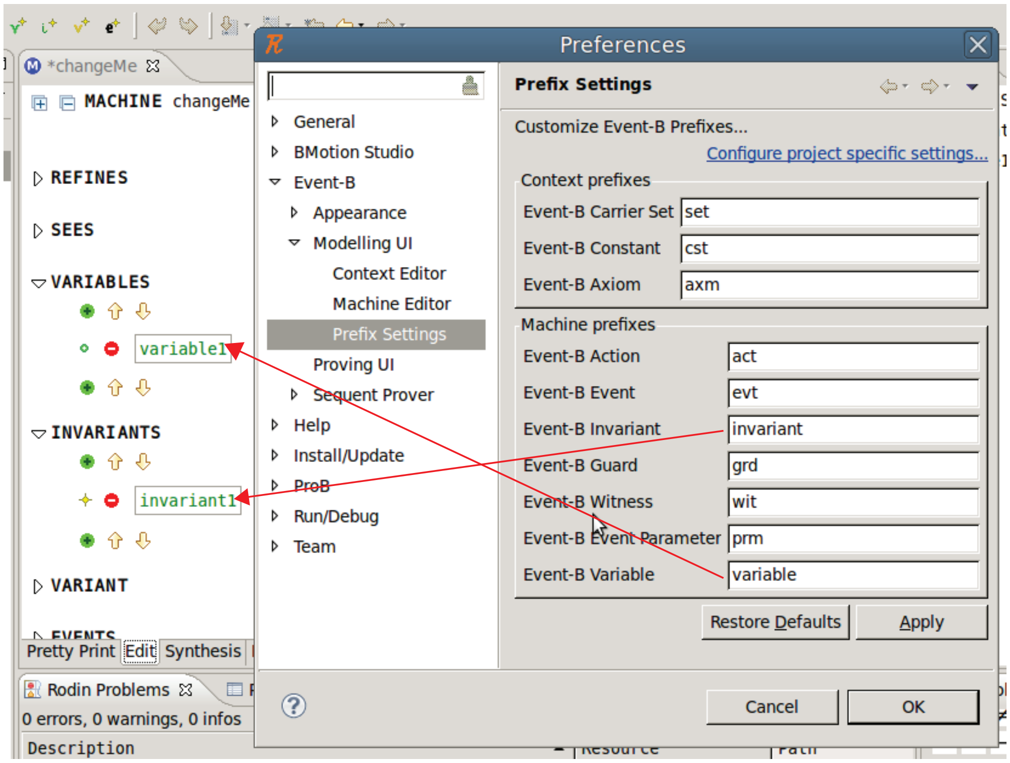

Figure 3.39: Prefix Settings

How to set prefixes Prefix settings can be accessed through in two different ways depending on the scope of their application: Window ⟩ Preferences ⟩ Event-B ⟩ Modelling UI ⟩ Prefix settings or directly via, Rename ⟩ Customize prefixes....

Project specific settings The user can select profiles locally for a project. To do so, you can select the Prefix Settings property page available via right-click on a project and select the Properties item. You can also click on the Configure project specific settings link on the Prefix Settings preference page. In this case, one will have to choose the project for which the prefixes should be set up. This is allowed via a specific project selection dialog. After the project selection, a dialog for prefix settings opens for the selected project.

A window (see Figure 3.40) appears, where a page handling prefixes settings allows a user to customize prefixes for a chosen project. On this page, the user can toggle the button Enable project specific settings.

- If this button is enabled, the prefixes used are those which are specified at this project level.
- If this button is not enabled, the prefixes used are those which are defined at the workspace level.

Sequent Prover / Auto/Post Tactic

Preferences for the selected auto and post tactic profile This section describes the Auto/Post Tactic tab of the Auto/Post Tactic preference page.

There are two scopes to set up preferences for the auto and post tactics: the workspace level and the project level. Note that if the automatic application of tactics is decided only at the workspace level, this option will also be set for the project level.

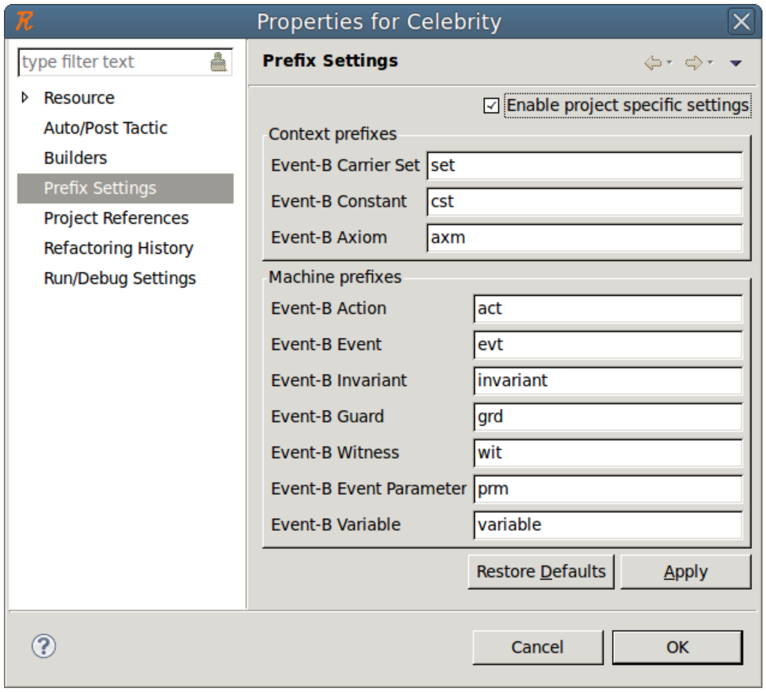

Figure 3.40: Project specific prefix settings

To access these preferences, open the "Auto/Post Tactic" preference page that can be found after Window ⟩ Preference ⟩ Sequent Prover.

Figure 3.41 shows the Auto/Post Tactic preference page.

The buttons 1 and 2 are activating/deactivating the automatic use of auto- and post-tactics. Here you can also choose the profile that should be used for auto- and post-tactics. Note that there is always a profile selected, and this profile can be changed regardless of whether the tactics are automatically launched or not because there is always a way to manually launch auto- and post-tactics. On the previously references figure, the Default Auto Tactic Profile is used for the automatic tactic, and the Default Post Tactic Profile is used for the post-tactic.

Preferences for available profiles This section describes the Profile tab of the Auto/Post Tactic preference page.

Figure 3.42 shows the contents of the profile tab. There are two visible lists: a list of profiles on the left and the tactics or provers that compose these profiles (Profile Details). Here one can see the contents of the default Auto Tactic Profile.

There are 4 buttons available to the user:

- New: to create a new profile "from scratch",
- Edit: to edit an existing (editable) profile,
- Remove: to remove a profile definitively,
- Duplicate: to duplicate a selected profile for further slight modification.

Default profiles can not be edited nor removed. That is why they this option appears in gray on the previously referenced figure.

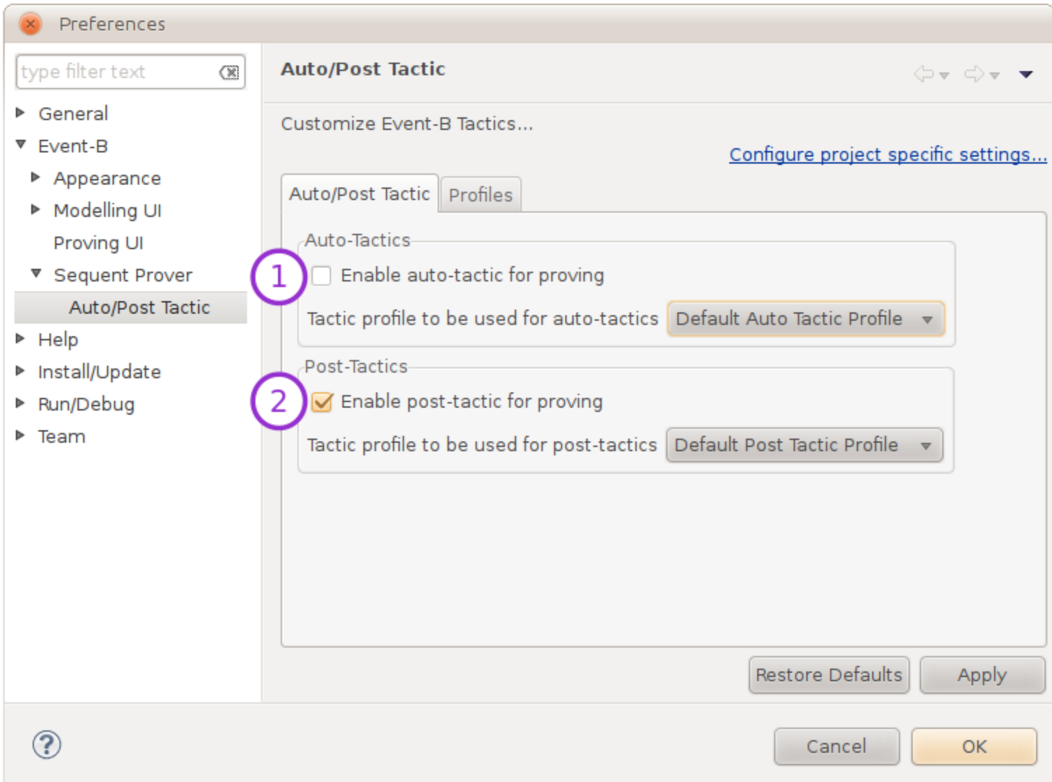

Figure 3.41: The "Auto/Post Tactic" preference page

Figure 3.43 shows the dialog available to edit or create a profile. For instance, here we create a profile named "MyFirstTacticProfile".

The list on the right represents the available and unselected tactics. The list of the left displays the profile contents and shows the selected tactics that will be applied from the top to the bottom. The user can choose a tactic from the list on the left and hit the >> button to select it or unselect some tactics from the list of the right using the << button. The user can re-order the priority of the selected tactic using the Up and Down button. By clicking on Finish, the profile will be saved and available for use in the auto and post tactics.

Project specific settings The user can select profiles locally for each project. To do so, select the Auto/Post Tactic property page available via right-click on a project and select the Properties item, or click the Configure project specific settings link on the Auto/Post Tactic preference page. Figure 3.44 shows what this Auto/Post Tactic tab looks like.

Note that the enablement of automatic use of post and auto tactics is decided at the workspace level. Figure 3.45 shows the Profiles tab of the Auto/Post Tactic page for a project specific setting. At the project level, there is a contextual menu available via right click from the list of defined profiles.

This contextual menu offers two options to the user:

- Import Workspace Profiles retrieve all the defined profiles in the workspace.

- Export to Workspace Profiles push a selected profile up in the list of workspace profiles.

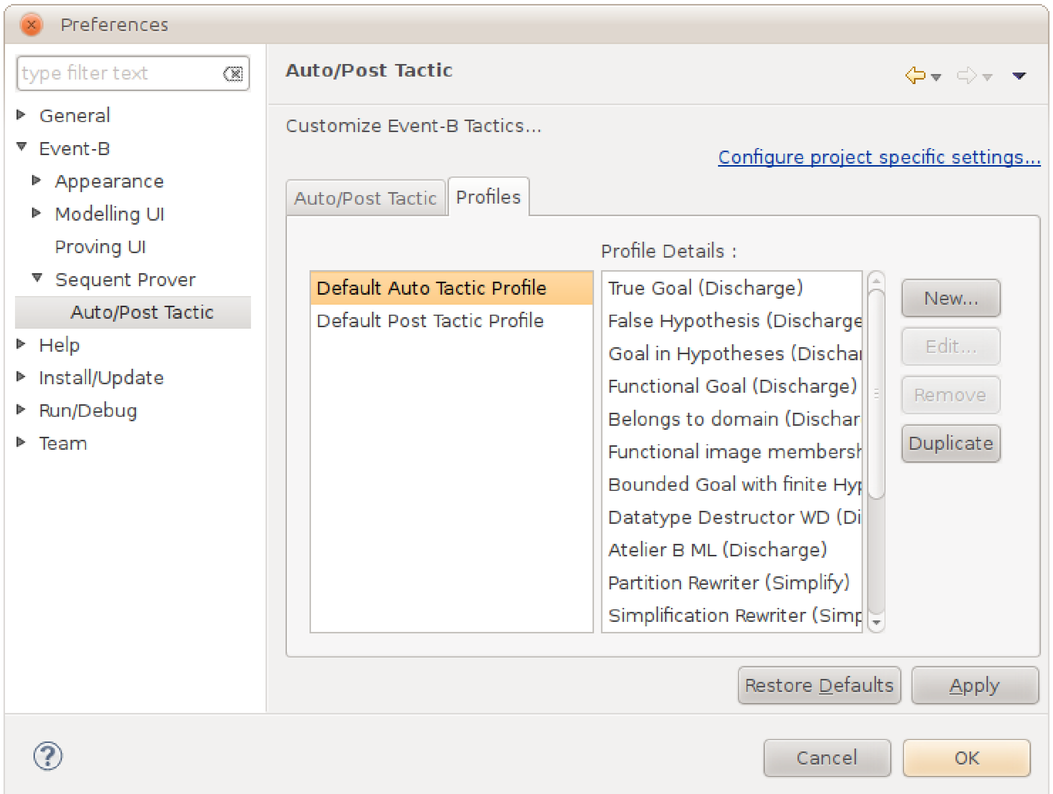

Figure 3.42: Selecting a profile for the Auto-Tactics

Preferences for the automatic tactics

Introduction The purpose of this section is to give a more detailed preferences to the user so he can build his own automated tactics. More precisely, the user should have a way to specify which parameters have to be passed to the reasoners and have a way to construct complex proof strategies.

User Documentation Here is the documentation about the current implementation of the Auto-tactic and Post-tactic preferences.

Tactic Combinators Tactic combinators can be used to construct complex proof strategies.

Historically, one combinator has existed since the beginning of auto tactic preferences: the "loop on all pending". It takes one or more tactics and loops them over every pending child until all tactic fail. Until Rodin 2.3 was released, it was the only combinator in Rodin. It is used on the configurable list of auto and post tactics. Rodin 2.3 is easier to configure because there are several other combinators and auto tactic editors.

The following is a list of combinators present by default.

One may notice the absence of child-specific combinator (i.e. combinators that apply tactic T1 on the first child, T2 on the second child, etc.) even though this kind of combinator exists in other provers. The reason is that we are concerned mainly auto tactics and these are tactics that are attempted in a general context. Provers with child-specific combinators are used to make manual proof because they require proof-specific adaptation.

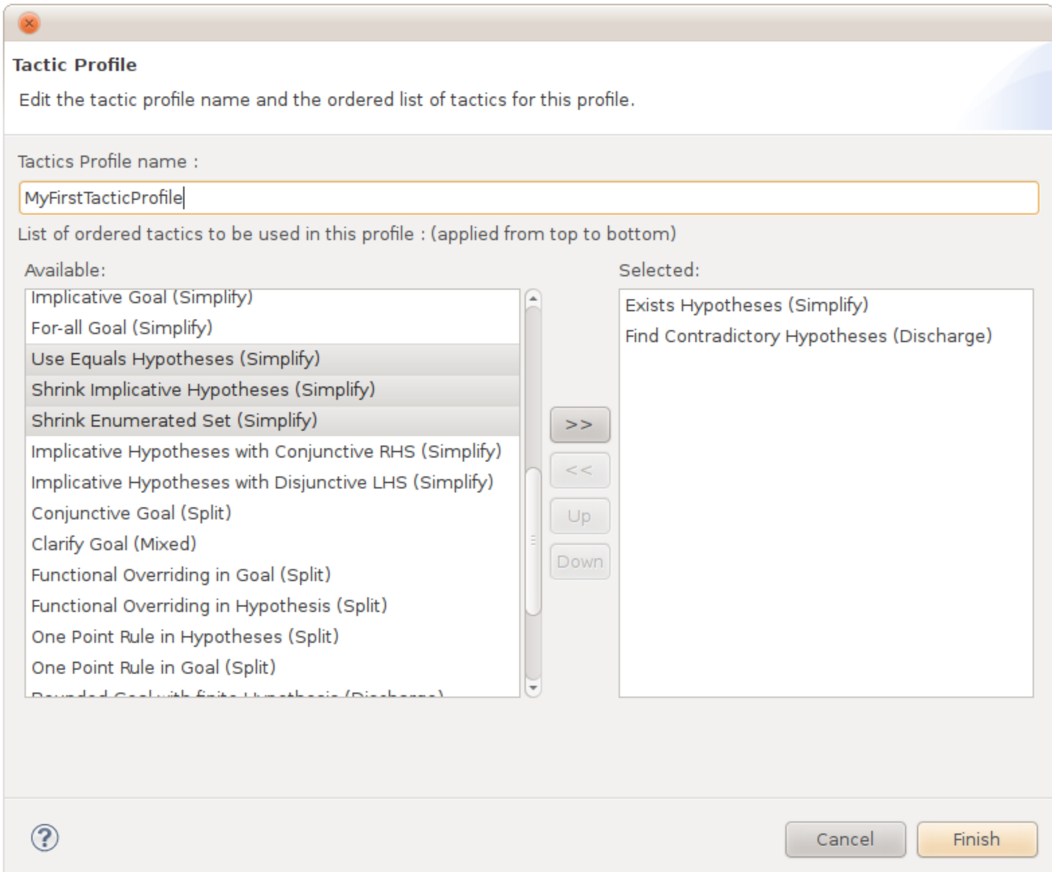

Figure 3.43: Selecting a profile for the Auto-Tactics

Composers A composer combinator applies its given tactic(s) to the given node. The given node may be open or closed. It succeeds if at least 1 tactic application is successful.

Name	Arity	Description	Stops when
Sequence	1..n	applies given tactics in given order	all tactics have been applied
Compose until Success	1..n	applies given tactics in given order	a tactic application succeeds
Compose until failure	1..n	applies given tactics in given order	a tactic application fails
Loop	1	applies given tactic repeatedly	the child tactic application fails

Selectors A selector combinator applies its given tactic to the set of nodes it selects. Selected nodes are computed from the given node. The given node may be open or closed. It succeeds if the tactic application is successful for at least 1 selected node.

Name	Arity	Selects
On all pending	1	all pending children of the given node (the given node itself if it is open)

Post Actions A post actions applies its given tactic to the given node. The given node must be open (otherwise it fails). Then it performs a specific treatment which is guarded by a trigger condition.

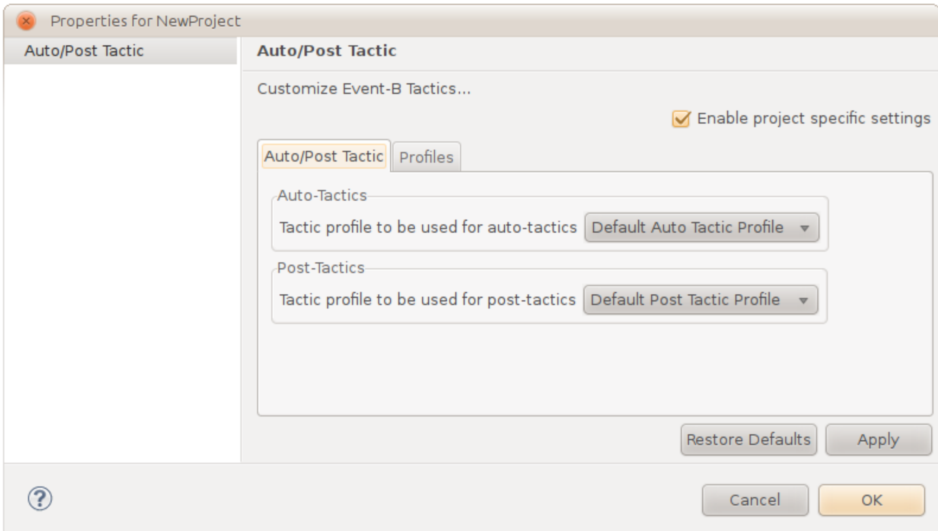

Figure 3.44: Auto/Post Tactic Tab for project specific settings for Auto/Post Tactic

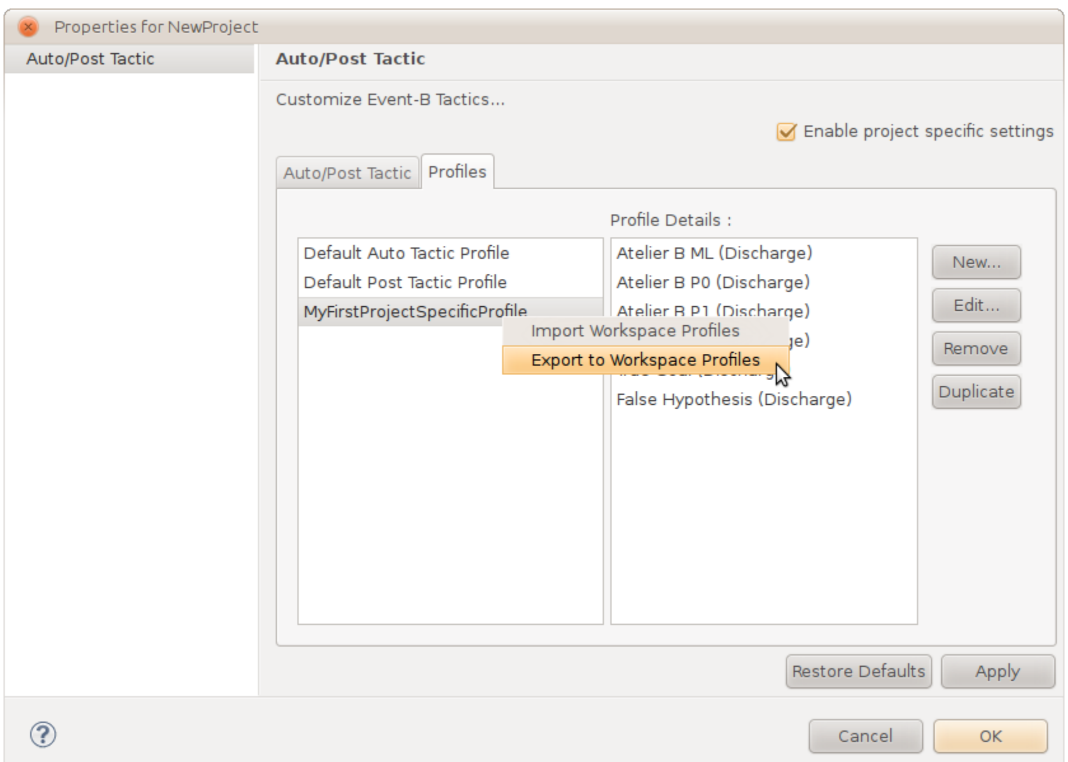

Figure 3.45: Profiles Tab for project specific settings for Auto/Post Tactic

Name	Arity	Trigger Condition	Post Action
Attempt	1	the given node still has pending children (subtree not closed)	prune proof tree at given node

Loop on All pending $loopOnAllPending(T_1 \ldots T_n) \;\widehat{=}\; loop(onAllPending(composeUntilSuccess(T_1 \ldots T_n)))$

Other Ideas timeout: a post action of arity 1 (with duration as input): limits the time allocated for the tactic that it is applied to (fails after time has gone out)

limitDepth: a post action of arity 1 (with depth as input): limits the proof tree depth for the tactic that it is applied to (prevents tree from growing beyond a given depth)

3.2 Event-B's modeling notation

In Event-B, we have two types of components: contexts and machines. Here we describe briefly the different elements of a context or machine. We do not use a specific syntax for describing the components because the syntax is dependent on the editor that is used. The default editor requires a structure which contains the elements that are described here.

Proof obligations are generated to guarantee certain properties of the modeled system. We will explain here which proof obligations are generated and we will list the goal and hypotheses that can be used when performing the proof for each one. This is done in the form:

	Description
Name	Label of the proof obligation (label refers to the label of the respective axiom/invariant/guard/etc.)
Goal	Goal that should be proved

Please note that Rodin often does not show a proof obligation if it is obviously true. If you expect a proof obligation that Rodin does not show, you can enforce that the proof obligation is showed by changing the model temporarily in a way that the proof cannot be proven. E.g. you can introduce a division by zero to see the corresponding well-definedness condition.

We first describe context and machines and how their consistency is proven. There are several locations where proof obligations due to well-definedness conditions or predicates marked as theorems are raised. We summarized the proof obligations in separate sections (3.2.4 resp. 3.2.5).

3.2.1 About the notation that we use

We denote a sequence of identifiers with $\mathbf{x} = x_1, \ldots, x_n$ and $\mathbf{x}' = x'_1, \ldots, x'_n$. As a convention, we use

- **c** for constants
- **v** and **w** for variables of an abstract resp. concrete machine
- **t** and **u** for parameters of an abstract resp. concrete machine
- A for axioms
- I and J for the invariants of the abstract machines resp. concrete machine
- G and H for the guards of the abstract events resp. concrete event

3.2.2 Contexts

A context describes the static part of a model. It consists of

- Carrier sets
- Constants
- Axioms
- Extended contexts

Carrier Sets

A new data type can be declared by adding its name – an identifier – to the *Sets* section. The identifier must be unique, i.e. it must not have been already declared as a constant or set in an extended context. The identifier is then implicitly introduced as a new constant (see below) that represents the set of all elements of the type.

A common pattern for declaring enumerated sets (sets where all elements are explicitly given) is to use the partition operator. If we want to specify a set S with elements $e_1, \ldots, e_n$, we declare S as a set, $e_1, \ldots, e_n$ as constants and add the axiom $\mathit{partition}(S, e_1, \ldots, e_n)$.

Extending a context

Other contexts can be extended by adding their name to the *Extends* section.

The resulting context consists of all constants and axioms of all extended contexts and the extending context itself. Thus for a context or machine that extends or sees the contexts, it makes no difference where a constant or axiom is declared.

Extending two contexts which declare a constant or set using the same identifier will result in an error.

Constants and axioms

Constants can be declared by adding their unique name (an identifier) to the *Constants* section. An axiom must also be in place from which the type of the constant can be inferred. We denote the sequence of all constants with $\mathbf{c}$.

An axiom is a statement that is assumed as true in the rest of the model. Each axiom consists of a label and a predicate $A(\mathbf{c})$.

Axioms can be marked as theorems, the generated proof obligation can be found in 3.2.5. The validity of a theorem results from the axioms declared before.

The well-definedness of axioms must be proven if an axiom contains a well-definedness condition (see 3.2.4 for the proof obligation).

3.2.3 Machines

A machine describes the dynamic behavior of a model by means of variables whose values are changed by events.

There are two basic things that must be proven for a machine:

1. The machine must be consistent, i.e. it should never reach a state which violates the invariant.

2. The machine is a correct refinement, i.e. its behavior must correspond to any machines that it refines.

Refinement and Abstract machines

A machine can refine at most one other machine. We refer to the refined machine as the abstract machine and refer to the refinement as the concrete machine. More generally, a machine M_0 can be refined by machine M_1, M_1 refined by M_2 and so on. The most concrete refinement would be M_n.

Basically, a refinement consists of two aspects:

1. The concrete machine's state is connected to the state of the abstract machine. To do this, an invariant is used to relate abstract and concrete variable. This invariant is called a *gluing invariant*.

2. Each abstract event can be refined by one or more concrete events.

The full invariant of the machine consists of both abstract and concrete invariants. The invariants are accumulated during refinements.

How to use Refinement: Refinement can be used to subsequently add complexity to the model - this is called superposition refinement (or horizontal refinement). It can also be used to add detail to data structures – this is called data refinement (or vertical refinement). We've seen both types of refinement in the tutorial (Chapter 2).

Seeing a context

If the machine sees a context, the sets and constants declared in the context can be used in all predicates and expressions. The conjunction of axioms $A(\mathbf{c})$ can be used as hypotheses in the proofs.

Variables and invariants

Variables can be declared by adding their unique name (an identifier) to the *Variables* section. The type of the variables must be inferable by the invariants of the machine. We denote the variables of the abstract machines $M_1, \ldots, M_{n-1}$ with a $\mathbf{v}$ and the variables of the concrete machine with a $\mathbf{w}$.

An invariant is a statement that must hold at each state of the machine. Each invariant i consists of a label and a predicate $I_i(\mathbf{c}, \mathbf{v}, \mathbf{w})$. An invariant can refer to the constants as well as the variables of the concrete and all abstract machines.

We write $I(\mathbf{c}, \mathbf{v})$ to denote the conjunction of all invariants of the abstract machines and $J(\mathbf{c}, \mathbf{v}, \mathbf{w})$ for the conjunction of the machine's invariant.

Invariants that are marked as theorems derive their correctness from the preservation of other invariants, so their preservation does not need to be proven. The proof obligation can be found in 3.2.5.

If an invariant contains a well-definedness condition, a corresponding proof obligation is generated (see 3.2.4).

Common variables between machines With some restrictions, the abstract variables $\mathbf{v}$ and concrete variables $\mathbf{w}$ can have variables in common. If a variable v is declared in a machine M_i, it can be re-used in the direct refinement M_{i+1}. In that case, it is assumed that the values of the abstract and concrete variable are always equal. To ensure this, special proof obligations are generated (see below in section 3.2.3). Once a variable disappears in a refinement, i.e. is not declared in machine M_{i+2}, it cannot be re-introduced in a later refinement.

Events

A possible state change of a machine is defined by an event. The condition under which an event can be executed is given by a guard. The event's action describes how the new and old state relate to each other.

Events occur atomically (i.e. one event happens at a time) leading to a new state. Two events never happen at the same time. Time is also not factored into the execution of the event.

An event has the following elements:

- Name
- Parameters
- Guards
- Witnesses
- Actions
- Status (ordinary, convergent or anticipated): The status is used for termination proofs (see section 3.2.3 for details).

An event can refine one or more events of an abstract machine. To keep things simple, we will first consider events with only one refined event. If there are several refinement steps – e.g. event E_1 is refined by E_2 and E_2 by $E3$, we call E_1 and E_2 the abstract events and E_3 the concrete event. Likewise, if we refer to the parameters of the abstract events, we mean all the parameters of E_1 and E_2.

Parameters An event may have an arbitrary number of *parameters*. Their names must be unique, i.e. there must be no constant or variable with the same name. The types of the parameters must be inferable by the guards of the event. We denote the parameters of all abstract events with $\mathbf{t}$ and the parameters of the concrete event with $\mathbf{u}$.

Similarly to variables, an event can have parameters in common with the event it refines. If the refined event has a parameter t which is not a parameter of the refinement, a witness $W_t(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{w}',\mathbf{u},t)$ for the abstract parameter must be specified (see below – 3.2.3).

Guards Each *guard* consist of a label and a predicate $H(\mathbf{c},\mathbf{w},\mathbf{u})$. Variables or parameters of abstract machines are not accessible in a guard.

We write $G(\mathbf{c},\mathbf{v},\mathbf{t})$ for the conjunction of all guards of all abstract events.

Like axioms and invariants, guards can be marked as theorems, too. The proof obligation can be found in 3.2.5. If the guard contains WD-conditions, a proof obligation is generated See 3.2.4 for the proof obligation.

Actions An action is composed of a label and an assignment. Section 3.3.8 gives an overview of how they are assigned. They can be put into two groups: deterministic and non-deterministic assignments. Each assignment affects one or more concrete variables.

If an event has more than one action, they are executed in parallel. An error will occur if a new value is assigned to a variable in more than one action. All variables to which no new value is assigned keep the same value in new and old state.

We now define the before-after-predicate $\mathcal{T}$ of the actions together. Let $Q_1,\ldots,Q_n$ be the before-after-predicate of the event's assignments. Let $x_1,\ldots,x_k$ be the variables that are assigned by any action of the event. And $y_1,\ldots,y_l$ be the variables of the concrete machine that are *not* modified by any of the event's actions (i.e. $\mathbf{w} = x_1,\ldots,x_k,y_1,\ldots,y_l$). Then the before-after-predicate of the concrete event is $\mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}') = Q_1 \wedge \ldots \wedge Q_n \wedge y_1 = y_1' \wedge \ldots \wedge y_l = y_l'$.

Please note that Rodin optimizes proof obligations when a before-after-predicate is a hypothesis. x' is replaced directly by x when x is not changed by the event and replaced by $E(\mathbf{c},\mathbf{w},\mathbf{u})$ when $E(\mathbf{c},\mathbf{w},\mathbf{u})$ is assigned to x deterministically.

Witnesses Witnesses are composed of a label and a predicate that are is to establish a link between the values of the variables and parameters of the concrete and abstract events. Most of the time, this predicate is a simple equality.

Unlike other elements in Event-B that have a label, the label of a witness has a meaning and cannot be chosen arbitrarily.

If the user does not specify a witness, Rodin uses the default witness $\top$.

Witnesses are necessary in the following circumstances:

- The abstract event has a parameter p that is not a parameter of the concrete event. In this case, the label of the witness must be p, and the witness has the form $W_p(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{w}',\mathbf{u},p)$.
- The abstract event assigns non-deterministically (using $:\in$ or $:|$) a value to a variable x that is not a variable of the concrete machine. In this case, the label of the witness must be x', the witness has the form $W_{x'}(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{w}',\mathbf{t},\mathbf{u},x')$.

We denote the conjunction of all witnesses of an event with $W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{v}',\mathbf{w}',\mathbf{t},\mathbf{u})$.

The feasibility of the witness must be proven, i.e. that there is actually a value such that the predicate holds.

	Witness feasibility for a parameter p
Name	eventlabel/p/WFIS
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{w}',\mathbf{u},p)$

	Witness feasibility for an abstract variable x
Name	eventlabel/x'/WFIS
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{w}',\mathbf{t},\mathbf{u},x')$

A witness may contain well-definedness conditions, see 3.2.4 for the corresponding proof obligation.

Initialization The initialization of a machine is given by a special event called *INITIALISATION*. Unlike other events, the initialization must not contain guards and parameters. The actions must not make use of variable values before the initialization event occurs. All variables must have a value assigned to them. If there is no assignment for the variable x, Rodin assumes a default assignment of the form $x :| \top$.

Ensuring machine consistency The following proof obligations are generated for events:

The assignment of each action must be well-defined when the event is enabled, see 3.2.4 for the corresponding proof obligation.

If the event's guard is enabled, every action must be feasible. This is trivially true in the case of the deterministic assignments. For a non-deterministic assignment a, the feasibility $\mathcal{F}(a)$ must be proven. The feasibility operator $\mathcal{F}$ is defined in the reference section regarding non-deterministic assignments.

	Action feasibility
Name	eventlabel/actionlabel/FIS
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u})$ $\Rightarrow \mathcal{F}(a)$

For each invariant I_i with the label invlabel that contains a variable affected by the assignment, it must be proven that the invariant still holds with the new values.

	Invariant preservation
Name	eventlabel/invlabel/INV
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u})$ $\Rightarrow I_i(\mathbf{c},\mathbf{v},\mathbf{w}')$

Rodin simplifies this proof obligations by replacing x' with x for variables that are not changed and x' by $E(\mathbf{c},\mathbf{w},\mathbf{u})$ for variables that are assigned by an deterministic ($x := E$) assignment.

There are special cases regarding initialisation:

	Action feasibility (in the initialisation)
Name	INITIALISATION/actionlabel/FIS
Goal	$A(\mathbf{c}) \Rightarrow \mathcal{F}(a)$

	Invariant establishment
Name	INITIALISATION/invlabel/INV
Goal	$A(\mathbf{c}) \Rightarrow I_i(\mathbf{c},\mathbf{v},\mathbf{w}')$

Ensuring a correct refinement

An event can refine one or more events of the abstract machine. We first consider the refinement of only one event. For refining more than one event (i.e. merging events), please see below in Section 3.2.3.

If an event does not refine an abstract event, it is implicitly assumed that it refines *skip*, the event that is always enabled (i.e. its guard is $\top$) and does nothing (i.e. all variables keep their values).

Guard strengthening A concrete event must only be enabled if the abstract event is enabled. This condition is called *guard strengthening*. For each abstract guard G_i with label guardlabel, the following proof obligation is generated:

	Guard strenghtening
Name	eventname/guardlabel/GRD
Goal	$A(\mathbf{c}) \land I(\mathbf{c},\mathbf{v}) \land J(\mathbf{c},\mathbf{v},\mathbf{w}) \land H(\mathbf{c},\mathbf{w},\mathbf{u}) \land$ $W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{v}',\mathbf{w}',\mathbf{t},\mathbf{u}) \land \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow G_i(\mathbf{c},\mathbf{v},\mathbf{t})$

Action simulation If an abstract event's action i (with before-after predicate $Q_i(\mathbf{c},\mathbf{v},\mathbf{t},\mathbf{v}')$) assigns a value to a variable that is also declared in the concrete machine, it must be proven that the abstract event's behaviour corresponds to the concrete behaviour. The behaviour of the concrete event is given by the concrete before-after-predicate $\mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$, the relevant abstract behaviour by the before-after-predicate $Q_i(\mathbf{c},\mathbf{v},\mathbf{t},\mathbf{v}')$. The relation between abstract and concrete event is specified by the witnesses.

	Action simulation
Name	eventname/actionlabel/SIM
Goal	$A(\mathbf{c}) \land I(\mathbf{c},\mathbf{v}) \land J(\mathbf{c},\mathbf{v},\mathbf{w}) \land H(\mathbf{c},\mathbf{w},\mathbf{u}) \land$ $W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{v}',\mathbf{w}',\mathbf{t},\mathbf{u}) \land \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow Q_i(\mathbf{c},\mathbf{v},\mathbf{t},\mathbf{v}')$

When the assignments are deterministic or the witnesses are equations, the proof obligation can usually be simplified by Rodin.

Preserved variables If x is a variable of both the abstract and concrete machine and the concrete event assigns a value x but the abstract event does not, it must be proven that the variable's value does not change. Let $Q_i(\mathbf{c},\mathbf{w},\mathbf{u},\mathbf{w}')$ be the before-after-predicate of the action that changes x.

	Equality of a preserved variable x
Name	eventname/x/EQL
Goal	$A(\mathbf{c}) \land I(\mathbf{c},\mathbf{v}) \land J(\mathbf{c},\mathbf{v},\mathbf{w}) \land H(\mathbf{c},\mathbf{w},\mathbf{u}) \land Q_i(\mathbf{c},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow x' = x$

Merging events

Refining more than one abstract event by a single event is called *merging* of events. To merge events, two conditions must be taken into account.

- If two abstract events declare the same parameter, they must be of the same type.
- All abstract events must have identical actions.

Instead of the guard strengthening proof obligation, the following proof obligation is created with $G_1,\ldots,G_n$ being the abstract guards of the merged events and $\mathbf{t}_1,\ldots,\mathbf{t}_n$ their parameters.

	Guard strengthening (merge)
Name	eventlabel/MRG
Goal	$A(\mathbf{c}) \land I(\mathbf{c},\mathbf{v}) \land J(\mathbf{c},\mathbf{v},\mathbf{w}) \land H(\mathbf{c},\mathbf{w},\mathbf{u}) \land$ $W(\mathbf{c},\mathbf{v},\mathbf{w},\mathbf{v}',\mathbf{w}',\mathbf{t},\mathbf{u}) \land \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow G_1(\mathbf{c},\mathbf{v},\mathbf{t}_1) \lor \ldots \lor G_n(\mathbf{c},\mathbf{v},\mathbf{t}_n)$

The other proof obligations are the same than refining a single event. Also the same rules for defining witnesses apply.

Extending events

Instead of refining another event, an event can *extend* it. In this case the refined event will implicitly have all the parameters, guards and actions of the refined event. It can have additional parameters, guards and actions. The ability to extend an event is just syntactic sugar – the same effect can be achieved by manually copying the parameters, guards and actions.

This is especially usefull when additional features are gradually introduced into a model by refinement, which is also called "superposition refinement".

Termination

Event-B makes it possible to prove the termination of events in a model. Termination means that a chosen set of events are enabled only a finite number of times before an event that is not marked as terminating occurs. To support proofs for termination, a *variant* $V(\mathbf{c}, \mathbf{w})$ can be specified in a model. A variant is an expression that is either numeric ($V(\mathbf{c}, \mathbf{w}) \in \mathbb{Z}$) or a finite set ($V(\mathbf{c}, \mathbf{w}) \in \mathbb{P}(\alpha)$, where α is an arbitrary type).

Events can be marked as

ordinary when the event may occur arbitrarily often and does not underly any restrictions regarding the variant.

convergent when the event must decrease the variant.

anticipated when the event must not increase the variant.

Informally, termination is proven by stating that the convergent events strictly decrease the variant which has a lower bound. If a model contains a convergent event, a variant must be specified. If only anticipated events are declared, it is sufficient to assume a default constant variant such that all anticipated events do not increase the variant. When an event is marked as anticipated, one must just prove that the event does not increase the variant. The proof of termination is then delayed to the refinements of the anticipated event. A refinement of an anticipated event must be either anticipated or convergent.

If the convergence of an event is proven, the convergence its refinements is also guaranteed due to the guard strengthening.

A variant must be well-defined, the corresponding well-definedness proof obligations can be found in 3.2.4. The following other proof obligations are generated:

Numeric variant If the variant is numeric an anticipated or convergent event must only be enabled when the variant is non-negative.

	Numeric variant is a natural number
Name	eventlabel/NAT
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c}, \mathbf{v}) \wedge J(\mathbf{c}, \mathbf{v}, \mathbf{w}) \wedge G(\mathbf{c}, \mathbf{v}, \mathbf{t}) \wedge H(\mathbf{c}, \mathbf{w}, \mathbf{u})$ $\Rightarrow V(\mathbf{c}, \mathbf{w}) \in \mathbb{N}$

A convergent event must decrease the variant

	Decreasing of a numeric variant (convergent event)
Name	eventlabel/VAR
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c}, \mathbf{v}) \wedge J(\mathbf{c}, \mathbf{v}, \mathbf{w}) \wedge$ $G(\mathbf{c}, \mathbf{v}, \mathbf{t}) \wedge H(\mathbf{c}, \mathbf{w}, \mathbf{u}) \wedge \mathcal{T}(\mathbf{t}, \mathbf{u}, \mathbf{w}, \mathbf{u}, \mathbf{w}')$ $\Rightarrow V(\mathbf{c}, \mathbf{w}') < V(\mathbf{c}, \mathbf{w})$

and an anticipated event must not increase the variant.

	Decreasing of a numeric variant (anticipated event)
Name	eventlabel/VAR
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge$ $G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow V(\mathbf{c},\mathbf{w}') \leq V(\mathbf{c},\mathbf{w})$

Set variant If the variant is a set t must be proven that the set is always finite:

	Decreasing of a variant (anticipated event)
Name	FIN
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \Rightarrow \text{finite}(V(\mathbf{c},\mathbf{w}))$

A convergent event must remove elements from the set

	Decreasing of a set variant (convergent event)
Name	eventlabel/VAR
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge$ $G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow V(\mathbf{c},\mathbf{w}') \subset V(\mathbf{c},\mathbf{w})$

and an anticipated event must not add elements.

	Decreasing of a set variant (anticipated event)
Name	eventlabel/VAR
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge$ $G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}')$ $\Rightarrow V(\mathbf{c},\mathbf{w}') \subseteq V(\mathbf{c},\mathbf{w})$

3.2.4 Well-definedness proof obligations

There are several locations where well-definedness proof obligations can occur. In the reference of the mathematical notation the well-definedness conditions of each operator are defined by the $\mathcal{L}$-operator (see 3.3.1).

For well-definedness conditions the order of axioms, invariants and guards is important. Trivial ($\top$) well-definedness conditions are usually not shown in Rodin.

Axioms For an axiom A_w, let A_b denote (the conjunction of) all axioms declared in extended contexts and the axioms already declared in the current context before the axiom in question.

	Well-definedness of an axiom
Name	label/WD
Goal	$A_b(\mathbf{c}) \Rightarrow \mathcal{L}(A_w(\mathbf{c}))$

.

Invariants For an invariant J_w, let J_b denote (the conjunction of) all the model's invariants declared before the theorem.

	Well-definedness of an invariant
Name	label/WD
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J_b(\mathbf{c},\mathbf{v},\mathbf{w}) \Rightarrow \mathcal{L}(J_w(\mathbf{c},\mathbf{v},\mathbf{w}))$

Guards For an invariant H_w, let H_b denote (the conjunction of) all the model's invariants declared before the theorem.

	Well-definedness of a guard
Name	eventname/guardlabel/WD
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge H_b(\mathbf{c},\mathbf{w},\mathbf{u}) \Rightarrow \mathcal{L}(H_w(\mathbf{c},\mathbf{w},\mathbf{u}))$

Witnesses A witness W may contain well-definedness conditions.

	Well-definedness of a witness
Name	eventlabel/witnesslabel/WWD
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge \mathcal{T}(\mathbf{t},\mathbf{u},\mathbf{w},\mathbf{u},\mathbf{w}') \Rightarrow \mathcal{L}(W)$

Actions The assignment a of each action with the label actionlabel of an event must be well-defined.

	Well-definedness of an action
Name	eventlabel/actionlabel/WD
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge G(\mathbf{c},\mathbf{v},\mathbf{t}) \wedge H(\mathbf{c},\mathbf{w},\mathbf{u}) \Rightarrow \mathcal{L}(\ a\ )$

Variants A variant $V(\mathbf{c},\mathbf{w})$ must be well-defined.

	Well-definedness of a variant
Name	VWD
Goal	$A(\mathbf{c}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \Rightarrow \mathcal{L}(V(\mathbf{c},\mathbf{w}))$

3.2.5 Theorems

Axioms, invariants and guards can be marked as theorems. In that case it must be proven that the validity of the theorems results from the axioms, invariants or guards declared before the theorem.

Sometimes an axiom/invariant/guard marked as theorem is also called a *derived* axiom/invariant/guard.

Axioms For an axiom A_{thm}, let A_b denote (the conjunction of) all axioms declared in extended contexts and the axioms already declared in the current context before the axiom in question.

	An axiom as theorem
Name	label/THM
Goal	$A_b(\mathbf{c}) \Rightarrow A_{thm}(\mathbf{c})$

Invariants For an invariant I_{thm}, let I_b denote (the conjunction of) all the model's invariants declared before the theorem.

	An invariant as theorem
Name	label/THM
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge I_b(\mathbf{c},\mathbf{v},\mathbf{w}) \Rightarrow I_{thm}(\mathbf{c},\mathbf{v},\mathbf{w})$

Guards For an invariant H_{thm}, let H_b denote (the conjunction of) all the event's guards declared before the theorem.

	A guard as theorem
Name	label/THM
Goal	$A(\mathbf{c}) \wedge I(\mathbf{c},\mathbf{v}) \wedge J(\mathbf{c},\mathbf{v},\mathbf{w}) \wedge H_b(\mathbf{c},\mathbf{w},\mathbf{u}) \Rightarrow H_{thm}(\mathbf{c},\mathbf{w},\mathbf{u})$

3.2.6 Generated proof obligations

We give a brief overview about which proof obligations are generated. This should help the user to identify why a proof obligation generated by identifying its label. For further information we refer to the relevant reference section.

generated in contexts		
well-definedness of an axiom	label/WD	3.2.4
axiom as theorem	label/THM	3.2.5
generated for machine consistency		
well-definedness of an invariant	label/WD	3.2.4
invariant as theorem	label/THM	3.2.5
well-definedness of a guard	event/guardlabel/WD	3.2.4
guard as theorem	event/guardlabel/THM	3.2.5
well-definedness of an action	event/actionlabel/WD	3.2.4
feasibility of a non-det. action	event/actionlabel/FIS	3.2.3
invariant preservation	event/invariantlabel/INV	3.2.3
generated for refinements		
guard strengthening	event/abstract_guard_label/GRD	3.2.3
action simulation	event/abstract_action_label/SIM	3.2.3
equality of a preserved variable	event/variable/EQL	3.2.3
guard strengthening (merge)	event/MRG	3.2.3
well definedness of a witness	event/identifier/WWD	3.2.4
feasibility of a witness	event/identifier/WFIS	3.2.3
generated for termination proofs		
well definedness of a variant	VWD	3.2.4
finiteness for a set variant	FIN	3.2.3
natural number for a numeric variant	event/NAT	3.2.3
decreasing of variant	event/VAR	3.2.3

3.2.7 Visibility of identifiers

This table shows which identifiers can be used in predicates or expressions.

Sets Sets that are defined in the context (in case of an axiom) or in a seen context. If a context extends another context, the sets of the extended context are treated as if they are defined in the extending context.

Constants Like the sets, constants that are defined in the context (in case of an axiom) or in a seen context.

Concrete Variables Variables that are defined in the machine itself. This does *not* include variables of refined machines.

Abstract Variables Variables that are defined in an abstract machine.

Concrete Parameters Parameters that are defined in the event itself. This does *not* include parameters of refined events.

Abstract Parameters Parameters that are defined in an abstract event.

	sets	constants	concrete variables	abstract variables	concrete parameters	abstract parameters
axiom	×	×				
invariant	×	×	×	×		
variant	×	×	×	×		
guard	×	×	×	×	×	
witness*	×	×	×	×	×	×
action*	×	×	×	×	×	

* Witnesses and actions have additional identifiers in their scope. See 3.2.3 for details on witnesses resp. 3.3.8 for the scope of the different assignments in actions.

3.3 Mathematical Notation

3.3.1 Introduction

Data types

In Event-B we have 3 kinds of basic data types:

- $\mathbb{Z}$ is the set of all integers.
- BOOL is the set of Booleans. It has two elements $\mathrm{BOOL} = \{\mathrm{TRUE}, \mathrm{FALSE}\}$.
- User defined carrier sets. These are defined in the *Sets* section of a context. Carrier sets are never empty. There is no other assumption made about carrier sets unless it is stated explicitly as an axiom.

From all data types α, β, two other kinds of data types can be constructed:

- $\mathbb{P}(\alpha)$ contains the sets of elements of α.
- $\alpha \times \beta$ contains the pairs where the first element is of type α and the second element of type β.

Expressions that are constructed by the rules above are called *type expressions*.

A note about the notation We use the Greek letters α, β, γ, ... to represent arbitrary data types. For an expression E, we write $E \in \alpha$ to state that E is of type α. In the following descriptions of Event-B's mathematical construct we describe the types of all constructs and their components.

For example, we will describe the maplet $E \mapsto F$ which is defined as $E \mapsto F \in \alpha \times \beta$ with $E \in \alpha$ and $F \in \beta$. We do not make any restrictions on the types α and β.

For predicates, we just describe the data types of their components. The predicate itself does not have a type. For example, consider the components' types of the equality of two expressions $E = F$: $E \in \alpha$ and $F \in \alpha$. By stating that E and F are both of type α, we express that both expressions must have the same type but do not make any further assumptions about their types.

Well-definedness

A predicate which describes the condition under which an expression or predicate in Event-B can be safely evaluated is the well-definedness condition. An example with integer division makes this clear: The expression $x \div y$ makes only sense when $y \neq 0$.

Well-definedness conditions are usually used for the well-definedness proof obligations.

In Rodin, the $\mathcal{L}$-operator defines which well-defined condition a predicate or expression has. When applied to the above example, integer division can be formatted as follows: $\mathcal{L}(x \div y) \mathrel{\widehat{=}} y \neq 0$.

In the following sections, we state for each mathematical construct what the well-definedness conditions are. In many cases, this is just the conjunction of the well-definedness conditions for the different syntactical parts of a construct.

Free identifiers

Free identifiers in predicates and expressions are those identifiers which are used but not introduced by quantifiers. More formally, we define the set of free identifiers *Free*(E) of an expression or predicate E recursively as follows:

Expression / Predicate	**Free identifiers**
Identifier x	$\{x\}$
Integer n	$\varnothing$
$\top$ $\bot$ BOOL TRUE FALSE $\varnothing$ id prj_1 prj_2 $\mathbb{Z}$ $\mathbb{N}$ $\mathbb{N}_1$	$\varnothing$
$\neg A$ $\mathrm{bool}(A)$ $\mathbb{P}(A)$ $\mathbb{P}_1(A)$ $\mathrm{finite}(A)$ $\mathrm{card}(A)$ $\mathrm{union}(A)$ $\mathrm{inter}(A)$ A^{-1} $\mathrm{dom}(A)$ $\mathrm{ran}(A)$ $-A$ $\min(A)$ $\max(A)$	$\mathit{Free}(A)$
$A \wedge B$ $A \vee B$ $A \Rightarrow B$ $A \Leftrightarrow B$ $A = B$ $A \neq B$ $A \in B$ $A \notin B$ $A \subseteq B$ $A \not\subseteq B$ $A \subset B$ $A \not\subset B$ $A \cup B$ $A \cap B$ $A \setminus B$ $A \times B$ $A \leftrightarrow B$ $A \trel B$ $A \srel B$ $A \strel B$ $A \lhd B$ $A \domsub B$ $A \rhd B$ $A \ransub B$ $A ; B$ $A \circ B$ $A \ovl B$ $A \parallel B$ $A \otimes B$ $A[B]$ $A \pfun B$ $A \rightarrow B$ $A \pinj B$ $A \rightarrowtail B$ $A \psur B$ $A \twoheadrightarrow B$ $A \tbij B$ $A(B)$ $A + B$ $A - B$ $A \cdot B$ $A \div B$ $A \bmod B$ $A \,\hat{}\, B$ $A \fcomp B$	$\mathit{Free}(A) \cup \mathit{Free}(B)$
$\{\, E_1, \ldots, E_n \,\}$ $\mathrm{partition}(E_1, \ldots, E_n)$	$\mathit{Free}(E_1) \cup \ldots \cup \mathit{Free}(E_n)$
$\forall \mathit{ids} \cdot P$ $\exists \mathit{ids} \cdot P$	$\mathit{Free}(P) \setminus \mathit{ids}$
$\{\, \mathit{ids} \cdot P \mid E \,\}$ $\bigcup \mathit{ids} \cdot P \mid E$ $\bigcap \mathit{ids} \cdot P \mid E$	$(\mathit{Free}(P) \cup \mathit{Free}(E)) \setminus \mathit{ids}$
$\{\, E \mid P \,\}$ $\bigcup E \mid P$ $\bigcap E \mid P$	$\mathit{Free}(P) \setminus \mathit{Free}(E)$

Structure of the subsections

The following reference subsections will have the form the form:

math. Symbol — `ASCII representation` — Name of the operator
... — ... — ...

Description A short description of what the operator does

Definition A more formal definition

Types A description of the types of all arguments and, if the operation is an expression, the expression's type

WD A description of the well-definedness conditions using the $\mathcal{L}$ operator

Feasibility Non-deterministic assignments may have feasibility conditions. These are used in the proof obligations of an event (Section 3.2.3).

Example We add examples to some constructs to clarify their use.

3.3.2 Predicates

Logical primitives

$\top$ — `true` — True
$\bot$ — `false` — False

Description The predicates $\top$ and $\bot$ are the predicates that are true and false respectively.

WD $\mathcal{L}(\top) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\bot) \mathrel{\widehat{=}} \top$

Logical operators

$\wedge$	—	`&`	—	Conjunction
$\vee$	—	`or`	—	Disjunction
$\Rightarrow$	—	`=>`	—	Implication
$\Leftrightarrow$	—	`<=>`	—	Equivalence
$\neg$	—	`not`	—	Negation

Description These are the usual logical operators.

Definition The following truth tables give an overview:

P	Q	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

P	$\neg P$
$\bot$	$\top$
$\top$	$\bot$

Types All arguments are predicates.

WD Please note that the operators $\wedge$ and $\vee$ are not commutative, because their well-definedness conditions distinguish between the first and second argument. Therefore, if their arguments have well-definedness conditions, the order matters. E.g. $x \neq 0 \wedge y \div x = 3$ is always well-defined, but $y \div x = 3 \wedge x \neq 0$ still has the well-definedness condition $x \neq 0$.

$\mathcal{L}(P \wedge Q) \mathrel{\widehat{=}} \mathcal{L}(P) \wedge (P \Rightarrow \mathcal{L}(Q))$
$\mathcal{L}(P \vee Q) \mathrel{\widehat{=}} \mathcal{L}(P) \wedge (P \vee \mathcal{L}(Q))$
$\mathcal{L}(P \Rightarrow Q) \mathrel{\widehat{=}} \mathcal{L}(P) \wedge (P \Rightarrow \mathcal{L}(Q))$
$\mathcal{L}(P \Leftrightarrow Q) \mathrel{\widehat{=}} \mathcal{L}(P) \wedge \mathcal{L}(Q)$
$\mathcal{L}(\neg(P)) \mathrel{\widehat{=}} \mathcal{L}(P)$

Quantified predicates

$\forall$	—	`!`	—	Universal quantification
$\exists$	—	`#`	—	Existential quantification

Description The universal quantification $\forall x_1, \ldots, x_n \cdot P$ is true if P is satisfied for all possible values of $x_1 \ldots, x_n$. A usual pattern for quantification is $\forall x_1 \ldots, x_n \cdot P_1 \Rightarrow P_2$ where P_1 is used to specify the types of the identifiers.

The existantial quantification $\forall x_1 \ldots, x_n \cdot P$ is true if there is a valuation of $x_1 \ldots, x_n$ such that P is satisfied.

The types of all identifiers $x_1 \ldots, x_n$ must be inferable by P. They can be referenced in P.

Types The quantifiers and the P are predicates.

WD $\mathcal{L}(\forall x_1 \ldots, x_n \cdot P) \mathrel{\widehat{=}} \forall x_1 \ldots, x_n \cdot \mathcal{L}(P)$
$\mathcal{L}(\exists x_1 \ldots, x_n \cdot P) \mathrel{\widehat{=}} \forall x_1 \ldots, x_n \cdot \mathcal{L}(P)$

Equality

$=$	—	`=`	—	equality
$\neq$	—	`/=`	—	inequality

Description Checks if both expressions are (not) equal.

Definition $E \neq F \mathrel{\widehat{=}} \neg(E = F)$

Types $E = F$ and $E \neq F$ are predicates with $E \in \alpha$ and $F \in \alpha$, i.e. E and F must have the same type.

WD $\mathcal{L}(E = F) \mathrel{\widehat{=}} \mathcal{L}(E) \wedge \mathcal{L}(F)$
$\mathcal{L}(E \neq F) \mathrel{\widehat{=}} \mathcal{L}(E) \wedge \mathcal{L}(F)$

Membership

$\in$	—	`:`	—	set membership
$\notin$	—	`/:`	—	negated set membership

Description Checks if an expression e denotes an element of a set S.

Definition $e \notin S \mathrel{\widehat{=}} \neg(e \in S)$

Types $e \in S$ and $e \notin S$ are predicates with $e \in \alpha$ and $S \in \mathbb{P}(\alpha)$.

WD $\mathcal{L}(e \in S) \mathrel{\widehat{=}} \mathcal{L}(e) \wedge \mathcal{L}(S)$
$\mathcal{L}(e \notin S) \mathrel{\widehat{=}} \mathcal{L}(e) \wedge \mathcal{L}(S)$

3.3.3 Booleans

BOOL	—	`BOOL`	—	Boolean values
TRUE	—	`TRUE`	—	Boolean true
FALSE	—	`TRUE`	—	Boolean false
bool	—	`bool`	—	Convert a predicate into a Boolean value

Description BOOL is a pre-defined carrier set that contains the constants TRUE and FALSE.

bool(P) denotes the Boolean value of a predicate P. If P is true, the expression is TRUE, if P is false, the expression is FALSE.

Definition $\mathrm{partition}(\mathrm{BOOL}, \{\mathrm{TRUE}\}, \{\mathrm{FALSE}\})$
$\mathrm{bool}(P) = \mathrm{TRUE} \Leftrightarrow P$

Types $\mathrm{BOOL} \in \mathbb{P}(\mathrm{BOOL})$
$\mathrm{TRUE} \in \mathrm{BOOL}$
$\mathrm{FALSE} \in \mathrm{BOOL}$
$\mathrm{bool}(P) \in \mathrm{BOOL}$ with P being a predicate.

WD $\mathcal{L}(\mathrm{BOOL}) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\mathrm{TRUE}) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\mathrm{FALSE}) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\mathrm{bool}(P)) \mathrel{\widehat{=}} \mathcal{L}(P)$

3.3.4 Sets

Set comprehensions

$\{\ ids \cdot P \mid E\ \}$	—	`{ids.P|E}`	—	Set comprehension
$\{\ E \mid P\ \}$	—	`{E|P}`	—	Set comprehension (short form)

Description *ids* is a comma-separated list of one ore more identifiers whose type must be inferable by the predicate P. The predicate P and E can contain references to the identifiers *ids*.

The set comprehension $\{\ x_1, \ldots, x_n \cdot P \mid E\ \}$ contains all values of E for values of $x_1, \ldots, x_n$ where P is true.

$\{\ E \mid P\ \}$ is a short form for $\{\ \mathit{Free}(E) \cdot P \mid E\ \}$ where $\mathit{Free}(E)$ denotes the list of free identifiers occurring in E (see Section 3.3.1)).

Definition $\{\ E \mid P\ \} = \{\ \mathit{Free}(E) \cdot P \mid E\ \}$

Types With $x_1 \in \alpha_1, \ldots, x_n \in \alpha_n$ and $E \in \beta$:
$\{\ x_1, \ldots, x_n \cdot P \mid E\ \} \in \mathbb{P}(\beta)$
$\{\ E \mid P\ \} \in \mathbb{P}(\beta)$

WD $\mathcal{L}(\{\ x_1, \ldots, x_n \cdot P \mid E\ \}) \quad \widehat{=} \quad \forall x_1, \ldots, x_n \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))$
$\mathcal{L}(\{\ E \mid P\ \}) \quad \widehat{=} \quad \forall \mathit{Free}(E) \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))$

Example The following set comprehensions contain all the first 10 squares numbers:
$\{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$
$= \{\ x \cdot x \in 1 \mathbin{..} 10 \mid x \,\hat{}\, 2\}$
$= \{\ x \mid \exists y \cdot y \in 1 \mathbin{..} 10 \land x = y \,\hat{}\, 2\ \}$
$= \{\ x \,\hat{}\, 2 \mid x \in 1 \mathbin{..} 10\ \}$

Basic sets

$\varnothing$	—	`{}`	—	Empty set
$\{\mathit{exprs}\}$	—	`{`*exprs*`}`	—	Set extension

Description *exprs* is a comma-separated list of one or more expressions of the same type.

The empty set $\varnothing$ contains no elements. The set extension $\{E_1, \ldots, E_n\}$ is the set that contains exactly the elements $E_1, \ldots, E_n$.

Definition $\varnothing = \{\ x \mid \bot\ \}$
$\{E_1, \ldots, E_n\} = \{\ x \mid x = E_1 \lor \ldots \lor x = E_n\}$

Types $\varnothing \in \mathbb{P}(\alpha)$, where α is an arbitrary type.
$\{E_1, \ldots, E_n\} \in \mathbb{P}(\alpha)$ with $E_1 \in \alpha, \ldots, E_n \in \alpha$

WD $\mathcal{L}(\varnothing) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\{E_1, \ldots, E_n\}) \mathrel{\widehat{=}} \mathcal{L}(E_1) \land \ldots \land \mathcal{L}(E_n)$

Subsets

$\subseteq$	—	`<:`	—	subset
$\not\subseteq$	—	`/<:`	—	not a subset
$\subset$	—	`<<:`	—	strict subset
$\not\subset$	—	`/<<:`	—	not a strict subset

Description $S \subseteq T$ checks if S is a subset of T, i.e. if all elements of S occurr in T. $S \subset T$ checks if S is a subset of T and S does not equal T. $S \not\subseteq T$ resp. $S \not\subset T$ are the respective negated variants.

Definition $S \subseteq T \mathrel{\widehat{=}} \forall e \cdot e \in S \Rightarrow e \in T$
$S \not\subseteq T \mathrel{\widehat{=}} \neg(S \subseteq T)$
$S \subset T \mathrel{\widehat{=}} S \subseteq T \land S \neq T$
$S \not\subset T \mathrel{\widehat{=}} \neg(S \subset T)$

Types $S \square T$ is a predicate with $S \in \mathbb{P}(\alpha)$, $T \in \mathbb{P}(\alpha)$ and for each operator $\square$ of $\subseteq$, $\not\subseteq$, $\subset$, $\not\subset$.

WD $\mathcal{L}(S \square T) \mathrel{\widehat{=}} \mathcal{L}(S) \wedge \mathcal{L}(T)$ for each operator $\square$ of $\subseteq$, $\not\subseteq$, $\subset$, $\not\subset$.

Operations on sets

$\cup$ — `\/` — Union
$\cap$ — `/\` — Intersection
$\setminus$ — `\` — Set subtraction

Description The union $S \cup T$ denotes the set that contains all elements that are in S or T. The intersection $S \cap T$ denotes the set that contains all elment that are in both S and T. The set subtraction or set difference $S \setminus T$ denotes all elements that are in S but not in T.

Definition $S \cup T = \{\ x \mid x \in S \vee x \in T\ \}$
$S \cap T = \{\ x \mid x \in S \wedge x \in T\ \}$
$S \setminus T = \{\ x \mid x \in S \wedge x \notin T\ \}$

Types $S \square T \in \mathbb{P}(\alpha)$ with $S \in \mathbb{P}(\alpha)$ and $T \in \mathbb{P}(\alpha)$ and for each operator $\square$ of $\cup$, $\cap$, $\setminus$

WD $\mathcal{L}(S \square T) \mathrel{\widehat{=}} \mathcal{L}(S) \wedge \mathcal{L}(T)$ for each operator $\square$ of $\cup$, $\cap$, $\setminus$

Power sets

$\mathbb{P}$ — `POW` — Power set
$\mathbb{P}_1$ — `POW1` — Set of non-empty subsets

Description $\mathbb{P}(S)$ denotes the set of all subsets of the set S. $\mathbb{P}(S)$ denotes the set of all non-empty subsets of the set S.

Definition $\mathbb{P}(S) = \{\ x \mid x \subseteq S\ \}$
$\mathbb{P}_1(S) = \mathbb{P}(S) \setminus \{\varnothing\}$

Types $\mathbb{P}(\alpha) \in \mathbb{P}(\mathbb{P}(\alpha))$ and $\mathbb{P}_1(\alpha) \in \mathbb{P}(\mathbb{P}(\alpha))$ with $S \in \mathbb{P}(\alpha)$.

WD $\mathcal{L}(\mathbb{P}(S)) \mathrel{\widehat{=}} \mathcal{L}(S)$
$\mathcal{L}(\mathbb{P}_1(S)) \mathrel{\widehat{=}} \mathcal{L}(S)$

Finite sets

finite — `finite` — Finite set
card — `card` — Cardinality of a finite set

Description finite(S) is a predicate that states that S is a finite set. card(S) denotes the cardinality of S. The cardinality is only defined for finite sets.

Definition $\mathrm{finite}(S)\ \mathrel{\widehat{=}}\ \exists n, b \cdot n \in \mathbb{N} \wedge b \in S \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \mathinner{..} n$
$\mathrm{card}(S) = n\ \mathrel{\widehat{=}}\ \exists b \cdot b \in S \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \mathinner{..} n$

Types finite(S) is a predicate and $\mathrm{card}(S) \in \mathbb{Z}$ with $S \in \mathbb{P}(\alpha)$, i.e. S must be a set.

WD $\mathcal{L}(\mathrm{finite}(S)) \mathrel{\widehat{=}} \mathcal{L}(S)$
$\mathcal{L}(\mathrm{card}(S)) \mathrel{\widehat{=}} \mathcal{L}(S) \wedge \mathrm{finite}(S)$

Partition

partition — `partition` — Partitions of a set

Description partition$(S, s_1, \ldots, s_n)$ is a predicate that states that the sets $s_1, \ldots, s_n$ constitute a partition of S. The union of all elements of a partition is S and all elements are disjoint.

partition(S) is equivalent to $S = \varnothing$.

Definition $\text{partition}(S, s_1, \ldots, s_n) \mathrel{\widehat{=}} S = s_1 \cup \ldots \cup s_n \land \forall i, j \cdot i \neq j \Rightarrow s_i \cap s_j = \varnothing$

Types partition$(S, s_1, \ldots, s_n)$ is a predicate with $S \in \mathbb{P}(\alpha)$ and $s_i \in \mathbb{P}(\alpha)$ for $i \in 1 \mathinner{..} n$

WD $\mathcal{L}(\text{partition}(S, s_1, \ldots, s_n)) \mathrel{\widehat{=}} \mathcal{L}(S) \land \mathcal{L}(s_1) \land \ldots \land \mathcal{L}(s_n)$

Generalized union and intersection

union — `union` — Generalized union
inter — `inter` — Generalized intersection

Description union(S) is the union of all elements of S. inter(S) is the intersection of all elements of S. The intersection is only defined for non-empty S.

Definition $\text{union}(S) = \{\ x \mid \exists s \cdot s \in S \land x \in s\ \}$
$\text{inter}(S) = \{\ x \mid \forall s \cdot s \in S \Rightarrow x \in s\ \}$

Types union$(S) \in \mathbb{P}(\alpha)$ and inter$(S) \in \mathbb{P}(\alpha)$ with $S \in \mathbb{P}(\mathbb{P}(\alpha))$.

WD $\mathcal{L}(\text{union}(S)) \mathrel{\widehat{=}} \mathcal{L}(S)$
$\mathcal{L}(\text{inter}(S)) \mathrel{\widehat{=}} \mathcal{L}(S) \land S \neq \varnothing$

Quantified union and intersection

$\bigcup$ — `UNION` — Quantified union
$\bigcap$ — `INTER` — Quantified intersection

Description $\bigcup x_1 \ldots, x_n \cdot P \mid E$ is the union of all values of E for valuations of the identifiers $x_1 \ldots, x_n$ that fulfill the predicate P. The types of $x_1, \ldots, x_n$ must be inferable by P.

Analougously is $\bigcup x_1 \ldots, x_n \cdot P \mid E$ the intersection of all values of E for valuations of the identifiers $x_1 \ldots, x_n$ that fulfill the predicate P.

Like set comprehensions (Section 3.3.4), the quantified union and intersection have a short form where the free variables of the expression are quantified implicitly: $\bigcup E \mid P$ and $\bigcap E \mid P$.

Definition $\bigcup x_1 \ldots, x_n \cdot P \mid E = \text{union}(\{\ x_1 \ldots, x_n \cdot P \mid E\})$
$\bigcap x_1 \ldots, x_n \cdot P \mid E = \text{inter}(\{\ x_1 \ldots, x_n \cdot P \mid E\})$
$\bigcup E \mid P = \bigcup \mathit{Free}(E) \cdot P \mid E$
$\bigcap E \mid P = \bigcap \mathit{Free}(E) \cdot P \mid E$

Types With $E \in \mathbb{P}(\alpha)$ and P being a predicate:
$(\bigcup x_1 \ldots, x_n \cdot P \mid E) \in \mathbb{P}(\alpha)$
$(\bigcap x_1 \ldots, x_n \cdot P \mid E) \in \mathbb{P}(\alpha)$
$(\bigcup E \mid P) \in \mathbb{P}(\alpha)$
$(\bigcap E \mid P) \in \mathbb{P}(\alpha)$

WD $\mathcal{L}(\bigcup x_1 \ldots, x_n \cdot P \mid E) \mathrel{\widehat{=}} (\ \forall x_1 \ldots, x_n \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))\ )$
$\mathcal{L}(\bigcap x_1 \ldots, x_n \cdot P \mid E) \mathrel{\widehat{=}} (\ \forall x_1 \ldots, x_n \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))\ ) \land \exists x_1 \ldots, x_n \cdot \mathcal{L}(P)$
$\mathcal{L}(\bigcup E \mid P) \mathrel{\widehat{=}} (\ \forall \mathit{Free}(E) \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))\ )$
$\mathcal{L}(\bigcap E \mid P) \mathrel{\widehat{=}} (\ \forall \mathit{Free}(E) \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))\ ) \land \exists \mathit{Free}(E) \cdot \mathcal{L}(P)$

3.3.5 Relations

Pairs and Cartesian product

$\mapsto$ — `|->` — Pair
$\times$ — `**` — Cartesian product

Description $x \mapsto y$ denotes the pair whose first element is x and second element is y.

$S \times T$ denotes the set of pairs where the first element is a member of S and second element is a member of T.

Definition $S \times T = \{\ x \mapsto y \mid x \in S \land y \in Y\ \}$

Types $x \mapsto y \in \alpha \times \beta$ with $x \in \alpha$ and $y \in \beta$.
$S \times T \in \mathbb{P}(\alpha \times \beta)$ with $S \in \mathbb{P}(\alpha)$ and $T \in \mathbb{P}(\beta)$.

WD $\mathcal{L}(x \mapsto y) \mathrel{\widehat{=}} \mathcal{L}(x) \land \mathcal{L}(y)$
$\mathcal{L}(S \times T) \mathrel{\widehat{=}} \mathcal{L}(S) \land \mathcal{L}(T)$

Relations

$\leftrightarrow$ — `<->` — Relations
$\mathrel{\leftarrow\!\!\!\!\leftrightarrow}$ — `<<->` — Total relations
$\mathrel{\leftrightarrow\!\!\!\!\rightarrow}$ — `<->>` — Surjective relations
$\mathrel{\leftarrow\!\!\!\!\leftrightarrow\!\!\!\!\rightarrow}$ — `<<->>` — Total surjective relations

Description $S \leftrightarrow T$ is the set of relations between the two sets S and T. A relation consists of pairs where the first element is of S and the second of T. $S \leftrightarrow T$ is just an abreviation for $\mathbb{P}(S \leftrightarrow T)$.

A total relation is a relation which relates each element of S to at least one element of T.

A surjective relation is a relation where there is at least one element of S for each element of T such that both are related.

Definition $S \leftrightarrow T = \mathbb{P}(S \times T)$
$S \mathrel{\leftarrow\!\!\!\!\leftrightarrow} T = \{\ r \mid r \in S \leftrightarrow T \land \mathrm{dom}(r) = S\ \}$
$S \mathrel{\leftrightarrow\!\!\!\!\rightarrow} T = \{\ r \mid r \in S \leftrightarrow T \land \mathrm{ran}(r) = S\ \}$
$S \mathrel{\leftarrow\!\!\!\!\leftrightarrow\!\!\!\!\rightarrow} T = (S \mathrel{\leftarrow\!\!\!\!\leftrightarrow} T) \land (S \mathrel{\leftrightarrow\!\!\!\!\rightarrow} T)$

Types For $S \in \mathbb{P}(\alpha)$ and $T \in \mathbb{P}(\beta)$ and for each operator $\square$ of $\leftrightarrow$, $\mathrel{\leftarrow\!\!\!\!\leftrightarrow}$, $\mathrel{\leftrightarrow\!\!\!\!\rightarrow}$, $\mathrel{\leftarrow\!\!\!\!\leftrightarrow\!\!\!\!\rightarrow}$:
$S \mathbin{\square} T \in \mathbb{P}(\mathbb{P}(\alpha \times \beta))$

WD $\mathcal{L}(S \mathbin{\square} T) \mathrel{\widehat{=}} \mathcal{L}(S) \land \mathcal{L}(T)$ for each operator $\square$ of $\leftrightarrow$, $\mathrel{\leftarrow\!\!\!\!\leftrightarrow}$, $\mathrel{\leftrightarrow\!\!\!\!\rightarrow}$, $\mathrel{\leftarrow\!\!\!\!\leftrightarrow\!\!\!\!\rightarrow}$.

Domain and Range

dom — `dom` — Domain
ran — `ran` — Range

Description If r is a relation between the sets S and T, the domain $\mathrm{dom}(r)$ is the set of elements of S that are related to at least one element of T by r.

Likewise the range $\mathrm{ran}(r)$ is the set of elements of T where at least one element of S relates to by r.

Definition $\mathrm{dom}(r) = \{\ x \mid \exists y \cdot x \mapsto y \in r\ \}$
$\mathrm{ran}(r) = \{\ y \mid \exists x \cdot x \mapsto y \in r\ \}$

Types $\mathrm{dom}(r) \in \mathbb{P}(\alpha)$ and $\mathrm{ran}(r) \in \mathbb{P}(\beta)$ with $r \in \mathbb{P}(\alpha \times \beta)$.

WD $\mathcal{L}(\mathrm{dom}(r)) \mathrel{\widehat{=}} \mathcal{L}(r)$
$\mathcal{L}(\mathrm{ran}(r)) \mathrel{\widehat{=}} \mathcal{L}(r)$

Domain and Range Restrictions

$\lhd$	—	`<|`	—	Domain restriction
$⩤$	—	`<<|`	—	Domain subtraction
$\lhd$	—	`|>`	—	Range restriction
$⩤$	—	`|>>`	—	Range subtraction

Description The domain restriction $S \lhd r$ is a subset of the relation r that contains all pairs whose first element is in S. $S ⩤ r$ is the subset where the pair's first element is *not* in S.

Analogously the range restriction $r \rhd S$ is a subset that contains all pairs whose second element is in S and $r ⩥ S$ is the set where the pair's second element is not in S.

Definition
$s \lhd r = \{\ x \mapsto y \mid x \mapsto y \in r \wedge x \in s\}$
$s ⩤ r = \{\ x \mapsto y \mid x \mapsto y \in r \wedge x \notin s\}$
$r \rhd s = \{\ x \mapsto y \mid x \mapsto y \in r \wedge y \in s\}$
$r ⩥ s = \{\ x \mapsto y \mid x \mapsto y \in r \wedge y \notin s\}$

Types $s \lhd r \in \mathbb{P}(\alpha \times \beta)$ and $s ⩤ r \in \mathbb{P}(\alpha \times \beta)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\alpha)$
$r \rhd s \in \mathbb{P}(\alpha \times \beta)$ and $r ⩥ s \in \mathbb{P}(\alpha \times \beta)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\beta)$

WD $\mathcal{L}(s \mathbin{\Box} r) \mathrel{\widehat{=}} \mathcal{L}(s) \wedge \mathcal{L}(r)$ for each operator $\Box$ of $\lhd$, $⩤$, $\rhd$, $⩥$

Operations on relations

$;$	—	`;`	—	Relational forward composition
$\circ$	—	`circ`	—	Relational backward composition
$⩤\!\!+$	—	`<+`	—	Relational override
$\parallel$	—	`||`	—	Parallel product
$\otimes$	—	`><`	—	Direct product
$^{-1}$	—	`~`	—	Inverse

Description An element x is related by $r\,;S$ to an element y if there is an element z such that R relates x to z and S relates z to y.

$s \circ r$ can be written as an alternative to $r\,;s$. This reflects the fact that for two functions f and g holds $f(g(x)) = (f \circ g)(x)$.

The relational overwrite $r \mathbin{⩤\!\!+} s$ is equal to r except all entries in r whose first element is in the domain of s are replaced by the corresponding entries in s.

If a relation r relates an element x to z and s relates x to z, the parallel product $r \parallel s$ relates x to the pair $y \mapsto z$.

The direct product $r \otimes s$ relates a pair $x \mapsto y$ to a pair $m \mapsto n$ when r relates x to m and s relates y to n.

The inverse relation r^{-1} relates an element x to y if the original relation r relates y to x.

Definition
$r\,;s = \{\ x \mapsto y \mid \exists z \cdot x \mapsto z \in r \wedge z \mapsto y \in s\ \}$
$r \circ s = s\,;r$
$r \mathbin{⩤\!\!+} s = s \cup (dom(s) ⩤ r)$
$r \parallel s = \{\ x \mapsto (y \mapsto z) \mid x \mapsto y \in r \wedge x \mapsto z \in s\ \}$
$r \otimes s = \{\ (x \mapsto y) \mapsto (m \mapsto n) \mid x \mapsto m \in r \wedge y \mapsto n \in s\ \}$
$r^{-1} = \{y \mapsto x \mid x \mapsto y \in r\ \}$

Types $r \mathbin{;} s \in \mathbb{P}(\alpha \times \gamma)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\beta \times \gamma)$
$r \circ s \in \mathbb{P}(\gamma \times \alpha)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\beta \times \gamma)$
$r \mathbin{<\!\!+} s \in \mathbb{P}(\alpha \times \beta)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\alpha \times \beta)$
$r \parallel s \in \mathbb{P}(\alpha \times (\beta \times \gamma))$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\alpha \times \gamma)$
$r \otimes s \in \mathbb{P}(\ (\alpha \times \gamma) \times (\beta \times \delta)\ )$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $s \in \mathbb{P}(\gamma \times \delta)$
$r^{-1} \in \mathbb{P}(\beta \times \alpha)$ with $r \in \mathbb{P}(\alpha \times \beta)$

WD $\mathcal{L}(r \mathbin{\Box} s) \mathrel{\widehat{=}} \mathcal{L}(r) \land \mathcal{L}(s)$ for each operator $\Box$ of ;, $\circ$, $\mathbin{<\!\!+}$, $\parallel$, $\otimes$
$\mathcal{L}(r^{-1}) \mathrel{\widehat{=}} \mathcal{L}(r)$

Relational image

$[\ldots]$ — `[...]` — Relational image

Description The relational image $r[S]$ are those elements in the range of r that are mapped from S.

Definition $r[S] = \{\ x, y \cdot x \in S \land x \mapsto y \in r \mid y\ \}$

Types $r[S] \in \mathbb{P}(\beta)$ with $r \in \mathbb{P}(\alpha \times \beta)$ and $S \in \mathbb{P}(\alpha)$

WD $\mathcal{L}(r[S]) \mathrel{\widehat{=}} \mathcal{L}(r) \land \mathcal{L}(S)$

Constant relations

id — `id` — Identity relation
prj_1 — `prj1` — First projection
prj_2 — `prj2` — Second projection

Description id is the identity relation that maps every element to itself.

prj_1 is a function that maps a pair to its first element. Likewise prj_2 maps a pair to its second element.

id, prj_1 and prj_2 are generic definitions. Their type must be infered from the environment.

Definition $\mathrm{id} = \{\ x \mapsto x \mid \top\ \}$
$\mathrm{prj}_1 = \{\ (x \mapsto y) \mapsto x \mid \top\ \}$
$\mathrm{prj}_2 = \{\ (x \mapsto y) \mapsto y \mid \top\ \}$

Types $\mathrm{id} \in \mathbb{P}(\alpha \times \alpha)$ for an arbitrary type α.
$\mathrm{prj}_1 \in \mathbb{P}((\alpha \times \beta) \times \alpha)$ and $\mathrm{prj}_1 \in \mathbb{P}((\alpha \times \beta) \times \beta)$ for arbitrary types α and β.

WD $\mathcal{L}(\mathrm{id}) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\mathrm{prj}_1) \mathrel{\widehat{=}} \top$
$\mathcal{L}(\mathrm{prj}_2) \mathrel{\widehat{=}} \top$

Example The assumption that a relation r is irreflexive can be expressed by:
$r \cap \mathrm{id} = \varnothing$

Sets of functions

$\pfun$ — `+->` — Partial functions
$\rightarrow$ — `-->` — Total functions
$\pinj$ — `>+>` — Partial injections
$\rightarrowtail$ — `>->` — Total injections
$\psurj$ — `+->>` — Partial surjections
$\twoheadrightarrow$ — `-->>` — Total surjections
$\bij$ — `>->>` — Bijections

Description A partial function from S to T is a relation that maps an element of S to at most one element of T. A function is total if its domain contains all elements of S, i.e. it maps every element of S to an element of T.

A function is injective (is an injection) if two distinct elements of S are always mapped to distinct elements of T. It is also equivalent to say that the inverse of an injective function is a also a function.

A function is surjective (is a surjection) if for every element of T there exists an element in S that is mapped to it.

A function is bijective (is a bijection) if it is both injective and surjective.

Definition $S \mathrel{+\!\!\!\rightarrow} T = \{\ f \mid f \in S \leftrightarrow T \wedge (\forall e, x, y \cdot e \mapsto x \in f \wedge e \mapsto y \in f \Rightarrow x = y)\ \}$
$S \rightarrow T = \{\ f \mid f \in S \mathrel{+\!\!\!\rightarrow} T \wedge \mathrm{dom}(f) = S\ \}$
$S \mathrel{\rightarrowtail\!\!\!\!\!+} T = \{\ f \mid f \in S \mathrel{+\!\!\!\rightarrow} T \wedge f^{-1} \in T \mathrel{+\!\!\!\rightarrow} S\ \}$
$S \rightarrowtail T = (S \mathrel{\rightarrowtail\!\!\!\!\!+} T) \cap (S \rightarrow T)$
$S \mathrel{\twoheadrightarrow\!\!\!\!\!+} T = \{\ f \mid f \in S \mathrel{+\!\!\!\rightarrow} T \wedge \mathrm{ran}(f) = T\ \}$
$S \twoheadrightarrow T = (S \mathrel{\twoheadrightarrow\!\!\!\!\!+} T) \cap (S \rightarrow T)$
$S \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} T = (S \rightarrowtail T) \cap (S \twoheadrightarrow T)$

Types With $S \in \mathbb{P}(\alpha)$, $T \in \mathbb{P}(\beta)$ and for each operator $\Box$ of $\mathrel{+\!\!\!\rightarrow}$, $\rightarrow$, $\mathrel{\rightarrowtail\!\!\!\!\!+}$, $\rightarrowtail$, $\mathrel{\twoheadrightarrow\!\!\!\!\!+}$, $\twoheadrightarrow$, $\mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}$:
$S \,\Box\, T \in \mathbb{P}(\mathbb{P}(\alpha \times \beta))$

WD For each operator $\Box$ of $\mathrel{+\!\!\!\rightarrow}$, $\rightarrow$, $\mathrel{\rightarrowtail\!\!\!\!\!+}$, $\rightarrowtail$, $\mathrel{\twoheadrightarrow\!\!\!\!\!+}$, $\twoheadrightarrow$, $\mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow}$:
$\mathcal{L}(S \,\Box\, T) \mathrel{\widehat{=}} \mathcal{L}(S) \wedge \mathcal{L}(T)$

Function application

$(\ldots)$ — `(...)` — Function application

Description The function application $f(x)$ yields the value for x of the function f. It is only defined if x is in the domain of f and if f is a actually a function.

Definition $f(x) = y \mathrel{\widehat{=}} x \mapsto y \in f$

Types $f(x) \in \beta$ with $f \in \mathbb{P}(\alpha \times \beta)$ and $x \in \alpha$

WD $\mathcal{L}(f(x)) \mathrel{\widehat{=}} \mathcal{L}(f) \wedge \mathcal{L}(x) \wedge f \in \alpha \mathrel{+\!\!\!\rightarrow} \beta \wedge x \in \mathrm{dom}(f)$ with $\mathbb{P}(\alpha \times \beta)$ being the type of f.

Lambda

λ — `%` — Lambda

Description $(\lambda\, \mathcal{P} \cdot P \mid E\ )$ is a function that maps an "input" $\mathcal{P}$ to a result E such that P holds.

$\mathcal{P}$ is a pattern of identifiers, parentheses and $\mapsto$ which follows the following rules:

- An identifier x is a pattern.
- An identifier x, followed by an $\circ\!\!\circ$ operator is a pattern (See 3.3.7 for more details).
- A pair $p \mapsto q$ is a pattern when p and q are patterns.
- (p) is pattern when p is pattern.

In the simplest case, $\mathcal{P}$ is just an identifier.

Definition $(\lambda\, \mathcal{P} \cdot P \mid E\ ) = \{\ \mathcal{P} \mapsto E \mid P\ \}$

Types $(\lambda\, \mathcal{P} \cdot P \mid E\ ) \in \mathbb{P}(\ \alpha \times \beta\ )$ with $\mathcal{P} \in \alpha$, P being a predicate and $E \in \beta$.

WD $\mathcal{L}(\lambda\, \mathcal{P} \cdot P \mid E\,) \;\widehat{=}\; \forall Free(\mathcal{P}) \cdot \mathcal{L}(P) \land (P \Rightarrow \mathcal{L}(E))$

Example A function *double* that returns the double value of a natural number:
$double = (\lambda x \cdot x \in \mathbb{N} \mid 2 \cdot x)$

The dot product of two 2-dimensional vectors can be defined by:
$dotp = (\lambda\ (a \mapsto b) \mapsto (c \mapsto d) \cdot a \in \mathbb{Z} \land b \in \mathbb{Z} \land c \in \mathbb{Z} \land d \in \mathbb{Z} \mid a \cdot c + b \cdot d\ )$

3.3.6 Arithmetic

Sets of numbers

$\mathbb{Z}$	—	`INT`	—	Integers
$\mathbb{N}$	—	`NAT`	—	Natural numbers, starting with 0
$\mathbb{N}_1$	—	`NAT1`	—	Natural numbers, starting with 1
$..$	—	`..`	—	Range of numbers

Description The set of all integers is denoted by $\mathbb{Z}$. It contains all elements of the type. The two subsets $\mathbb{N}$ and $\mathbb{N}_1$ contain all elements greater than or equal to 0 and 1 respectively. The range of numbers between a and b is denoted by $a\,..\,b$.

Definition $\mathbb{N} = \{\ n \mid n \in \mathbb{Z} \land n \geq 0\ \}$
$\mathbb{N}_1 = \{\ n \mid n \in \mathbb{Z} \land n \geq 1\ \}$
$a\,..\,b = \{\ n \mid n \in \mathbb{Z} \land a \leq n \land n \leq b\ \}$

Types $\mathbb{Z} \in \mathbb{P}(\mathbb{Z})$
$\mathbb{N} \in \mathbb{P}(\mathbb{Z})$
$\mathbb{N}_1 \in \mathbb{P}(\mathbb{Z})$
$a\,..\,b \in \mathbb{P}(\mathbb{Z})$ with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$

WD $\mathcal{L}(\mathbb{Z}) \,\widehat{=}\, \top$
$\mathcal{L}(\mathbb{N}) \,\widehat{=}\, \top$
$\mathcal{L}(\mathbb{N}_1) \,\widehat{=}\, \top$
$\mathcal{L}(a\,..\,b) \,\widehat{=}\, \mathcal{L}(a) \land \mathcal{L}(b)$

Arithmetic operations

$+$	—	`+`	—	Addition
$-$	—	`-`	—	Subtraction or unary minus
$\cdot$	—	`*`	—	Multiplication
$\div$	—	`/`	—	Integer division
mod	—	`mod`	—	Modulo
$\frown$	—	`^`	—	Exponentation

Description These are the usual arithmetic operations.

Definition Addition, subtraction and multiplication behave as expected.

The division is defined in a way that $1 \div 2 = 0$ and $-1 \div 2 = 0$:
$a \div b = \max(\{\ c \mid c \in \mathbb{N} \land b \cdot c \leq a\ \})$ for $a \in \mathbb{N}$ and $b \in \mathbb{N}$
$(-a) \div b = -(a \div b)$
$a \div (-b) = -(a \div b)$

$a \bmod b = c \;\widehat{=}\; c \in 0\,..\,b-1 \land \exists k \cdot k \in \mathbb{N} \land k \cdot b + c = a$

Types With $a \in \mathbb{Z}$, $b \in \mathbb{Z}$ and for each operator $\square$ of $+$, $-$, $\cdot$, $\div$, mod:
$a \,\square\, b \in \mathbb{Z}$
$-a \in \mathbb{Z}$

WD $\mathcal{L}(a+b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b)$
$\mathcal{L}(a-b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b)$
$\mathcal{L}(-a) \mathrel{\widehat{=}} \mathcal{L}(a)$
$\mathcal{L}(a \cdot b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b)$
$\mathcal{L}(a \div b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b) \wedge b \neq 0$
$\mathcal{L}(a \bmod b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b) \wedge a \geq 0 \wedge b > 0$
$\mathcal{L}(a \,\widehat{}\, b) \mathrel{\widehat{=}} \mathcal{L}(a) \wedge \mathcal{L}(b) \wedge a \geq 0 \wedge b \geq 0$

Minimum and Maximum

min — `min` — Minimum
max — `max` — Maximum

Description $\min(I)$ resp. $\max(I)$ denotes the smallest resp. largest number in the set of integers I.

The minimum and maximum are only defined if such a number exists.

Definition $\min(I) = b \mathrel{\widehat{=}} b \in I \wedge (\forall x \cdot x \in I \Rightarrow b \leq x)$
$\max(I) = b \mathrel{\widehat{=}} b \in I \wedge (\forall x \cdot x \in I \Rightarrow b \geq x)$

Types $\min(I) \in \mathbb{Z}$ and $\max(I) \in \mathbb{Z}$ with $I \in \mathbb{P}(\mathbb{Z})$.

WD $\mathcal{L}(\min(I)) \mathrel{\widehat{=}} \mathcal{L}(I) \wedge I \neq \varnothing \wedge \exists b \cdot \forall x \cdot x \in I \Rightarrow b \leq x$
$\mathcal{L}(\max(I)) \mathrel{\widehat{=}} \mathcal{L}(I) \wedge I \neq \varnothing \wedge \exists b \cdot \forall x \cdot x \in I \Rightarrow b \geq x$

3.3.7 Typing

$\circ\!\!\circ$ — `oftype` — of type

Description $E \circ\!\!\circ\, \alpha$ is an expression that has exactly the value of E but its type is specified by the type expression α (Section 3.3.1).

E is restricted to expressions whose type does not depend on an argument of that expression. Namely these are the constant relations id, prj_1, prj_2 and the empty set $\varnothing$.

Another location where the operator can be used is the declaration of bound variables in quantifiers and patterns in lambda expressions. Each identifier can be followed by $\circ\!\!\circ$ and the identifier's type.

Definition $E \circ\!\!\circ\, \alpha = E$

Types $E \circ\!\!\circ\, \alpha \in \alpha$ with $E \in \alpha$

WD $\mathcal{L}(E \circ\!\!\circ\, \alpha) \mathrel{\widehat{=}} \mathcal{L}(E)$

Example The predicate $\varnothing = \varnothing$ is not correctly typed in Event-B because the types of $\varnothing$ are not inferable. A valid alternative would be:
$(\varnothing \circ\!\!\circ\, \mathbb{Z}) = \varnothing$

The predicate $\exists x, y \cdot x \neq y$ is not correctly typed because the types of x and y cannot be inferred: A valid alternative (for integers) is:
$\exists x \circ\!\!\circ\, \mathbb{Z}, y \cdot x \neq y$

The following lambda expression uses the $\circ\!\!\circ$ operator:
$(\lambda x \circ\!\!\circ\, \mathbb{Z} \mapsto y \circ\!\!\circ\, \text{BOOL} \mid x > 0 \cdot x + 1)$
An arguably more readably version without the use of $\circ\!\!\circ$ is:
$(\lambda x \mapsto y \mid x > 0 \wedge y \in \text{BOOL} \cdot x + 1)$

3.3.8 Assignments

We did not consider the scope of the expressions in the previous sections, i.e. we did not say which constants or variables can be used in an expression or predicate because this depends completely on the context where the expression is used.

The following paragraphs are about assignments. Depending on the type of assignment it may be important to specify which identifiers can be used. Thus we use the notion $E(x_1, \ldots, x_n)$ to specify which free identifiers may occur in the expression E.

Deterministic Assignments

$:=$ — `:=` — deterministic assignment

Description $x_1, \ldots, x_n :=, E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}) \ldots, E_n(\mathbf{c}, \mathbf{w}, \mathbf{u})$ assigns the expressions E_i to the variable x_i, with $i \in 1 \mathinner{..} n$. All x_i must be distinct identifiers that refer to variables of the concrete machine. $\mathbf{c}, \mathbf{w}, \mathbf{u}$ represent the sequence of all constants, variables of the concrete machine and parameters of the concrete event.

There is a special form of the assignment which does a relational overwrite:
$x(F(\mathbf{c}, \mathbf{w}, \mathbf{u})) := E(\mathbf{c}, \mathbf{w}, \mathbf{u})$.

Definition The before-after-predicate of $x_1, \ldots, x_n := E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}), \ldots, E_n(\mathbf{c}, \mathbf{w}, \mathbf{u})$ is
$x_1' = E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}) \wedge \ldots \wedge x_n' = E_n(\mathbf{c}, \mathbf{w}, \mathbf{u})$.

The assignment is equivalent to $x_1, \ldots, x_n :\mid x_1' = E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}) \wedge \ldots \wedge x_n' = E_n(\mathbf{c}, \mathbf{w}, \mathbf{u})$.

The special form is syntactic sugar for:
$x(E_1(\mathbf{c}, \mathbf{w}, \mathbf{u})) := E_2(\mathbf{c}, \mathbf{w}, \mathbf{u}) \quad \widehat{=} \quad x := x \lhd\!\!- \{\ E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}) \mapsto E_2(\mathbf{c}, \mathbf{w}, \mathbf{u})\ \}$

Types $x_i \in \alpha$ and $E_i(\mathbf{c}, \mathbf{w}, \mathbf{u}) \in \alpha$ for $i \in 1 \mathinner{..} n$.

WD $\mathcal{L}(\ x_1, \ldots, x_n := E_1(\mathbf{c}, \mathbf{w}, \mathbf{u}), \ldots, E_n(\mathbf{c}, \mathbf{w}, \mathbf{u})\ ) \quad \widehat{=} \quad \mathcal{L}(E_1(\mathbf{c}, \mathbf{w}, \mathbf{u})) \wedge \ldots \wedge \mathcal{L}(E_n(\mathbf{c}, \mathbf{w}, \mathbf{u}))$
$\mathcal{L}(\ x(E_1(\mathbf{c}, \mathbf{w}, \mathbf{u})) := E_2(\mathbf{c}, \mathbf{w}, \mathbf{u})\ ) \quad \widehat{=} \quad \mathcal{L}(E_1(\mathbf{c}, \mathbf{w}, \mathbf{u})) \wedge \mathcal{L}(E_2(\mathbf{c}, \mathbf{w}, \mathbf{u}))$

Non-deterministic assignments

$:\mid$ — `:|` — non-deterministic assignment with a before-after-predicate

Description $x_1, \ldots, x_n :\mid Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n')$ assigns to the variables $x_1 \ldots, x_n$ any value such that the the before-after-predicate Q is fulfilled. All x_i are identifier that refer to a variable of the concrete machine. $\mathbf{c}, \mathbf{w}, \mathbf{u}$ represent the sequence of all constants, variables of the concrete machine and parameters of the concrete event. $x_1, \ldots, x_n$ are in $\mathbf{w}$.

This is the most general form of assignment, all other assignments can be converted to this.

Definition The before-after-predicate is $Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n')$.

Types $x \in \alpha$ and $E(\mathbf{c}, \mathbf{w}, \mathbf{u}) \in \mathbb{P}(\alpha)$

WD $\mathcal{L}(\ x_1, \ldots, x_n :\mid Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n')\ ) \quad \widehat{=} \quad \mathcal{L}(x_1, \ldots, x_n :\mid Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n'))$

Feasibility $\mathcal{F}(\ x_1, \ldots, x_n :\mid Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n')\ ) \quad \widehat{=} \quad \exists x_1', \ldots, x_n' \cdot Q(\mathbf{c}, \mathbf{w}, \mathbf{u}, x_1', \ldots, x_n')$

$:\in$ — `::` — non-deterministic assignment of a set member

Description $x :\in E(\mathbf{c}, \mathbf{w}, \mathbf{u})$ assigns to the variable x any value of the set E. x is an identifier that refers to a variable of the concrete machine. $\mathbf{c}, \mathbf{w}, \mathbf{u}$ represent the sequence of all constants, variables of the concrete machine and parameters of the concrete event.

Definition The before-after-predicate is $x' \in E(\mathbf{c}, \mathbf{w}, \mathbf{u})$.
The assignment is equivalent to $x :\mid x' \in E(\mathbf{c}, \mathbf{w}, \mathbf{u})$.

Types $x \in \alpha$ and $E(\mathbf{c}, \mathbf{w}, \mathbf{u}) \in \mathbb{P}(\alpha)$

WD $\mathcal{L}(\ x :\in E(\mathbf{c}, \mathbf{w}, \mathbf{u})\ ) \quad \widehat{=} \quad \mathcal{L}(E(\mathbf{c}, \mathbf{w}, \mathbf{u}))$

Feasibility $\mathcal{F}(\ x :\in E(\mathbf{c}, \mathbf{w}, \mathbf{u})\ ) \mathrel{\widehat{=}} E(\mathbf{c}, \mathbf{w}, \mathbf{u}) \neq \varnothing$

3.4 Proving

In Section 3.2.6, we learned what proof obligations are generated by Rodin from an Event-B model. We validate the model by discharging proof obligations. This is what we call proving.

In this chapter we will:

- Explain proof rules
- Explain tactics
- Explain and describe provers
- Explain reasoners
- Describe how to perform automatic and manual proving
- Purge proofs for maintenance

3.4.1 Sequents

A sequent is a formal statement describing something we want to prove.

Sequents are of the following form

$H \vdash G$

where H is the set of hypotheses (predicates) and G is the goal that can be inferred from the predicates.

The above statement can be read as follows: Under the hypotheses H, prove the goal G.

3.4.2 Proof Rules

In its pure mathematical form, a proof rule is a tool to perform a formal proof and is denoted by:

$$\frac{A}{C}$$

where A is a (possibly empty) list of sequents (the antecedents of the proof rule) and C is a sequent (the consequent of the rule). We interpret the above proof rule as follows: The combination of the proofs of each sequent of A prove the sequent C.

Example: Consider the following proof rule:

$$\frac{E_1}{E_1 \vee E_2}$$

This says that if E_1 holds, then the statement $E_1 \vee E_2$ must hold as well. Thus, we can replace the sequent by the consequent.

Proof Rule Representation in Rodin

In Rodin, the representation for proof rules is more structured not only to reduce the space required to store the rule but, more importantly, to support proof reuse. A proof rule in Rodin contains the following:

used goal A used goal predicate.

used hypotheses The set of used hypotheses.

antecedents A list of antecedents (to be explained later).

reasoner The reasoner used to generate this proof rule (see reasoners (Section 3.4.6)).

reasoner input The input for the reasoner to generate this proof rule (see reasoners (Section 3.4.6).

Each antecedent of the proof rule contains the following information:

new goal A new goal predicate.

added hypotheses The set of added hypotheses.

With this representation, a proof rule in Rodin corresponding to a proof schema as follows:

$$\frac{H, H_u, H_{A_0} \vdash G_{A_0} \quad \ldots \quad H, H_u, H_{A_n-1} \vdash G_{A_n-1}}{H, H_u \vdash G_u}$$

Where:

- H_u is the set of used hypotheses
- G_u is the used goal
- H_{A_i} is the set of added hypotheses corresponding to the ith antecedent.
- G_{A_i} is the new goal corresponding to the ith antecedent.
- H is the meta-variable that can be instantiated.

Applying Proof Rules

Given a proof rule of the form mentioned above, the following describes how to apply this rule to an input sequent. If the process is successful, a list of output sequences is produced.

- The rule is applicable if the goal of the sequent is not exactly the same as the used goal or if any of the used hypotheses are not contained in the set of hypotheses of the input sequent.
- If the case is applicable, the antecedent sequents are returned. The goal of each antecedent sequent is the new goal. The hypotheses of each antecedent sequent are the union of the old hypotheses and added hypotheses of the corresponding antecedent.

The user interface for proving is explained in Section 3.1.5. The practical application of proof rules is explained in Section 2.9.4

.

3.4.3 Proof Tactics

Tactics provide an easier way to construct and manage proof search and manipulation. They provide calls to the underlying reasoners or other tactics to modify proofs.

A list of all proof tactics is maintained in the Rodin Wiki.[3] This list is really comprehensive — be sure to check it out!

Tactics can be applied as follows:

Automatic Rodin can automatically apply a number of tactics after each manual proof step.

Proof tree Pruning the proof tree is a tactic that can be applied from the proof tree through the context menu. Other tactics may be available there.

In sequents Some sequents have elements that are highlighted in red. Clicking on these elements brings up a menu with all applicable tactics so that they can be applied manually.

It may be useful to consider the following categories of tactics:

Basic Tactics

The tactics that change the proof tree only at the point of application.

- Prune - A direct application of the pruning facility providing by the proof tree. The tactic is successful if the input node are not pending.
- Rule Application Tactics - Tactics of this class provide a wrapper around a proof rule (See Proof Rules). The tactic is successful if the proof rule is successfully applied to the input node.
- Reasoner Application Tactics - Tactics of this class provide a wrapper around a reasoner (See Reasoner). The tactic is successful if the reasoner is successfully applied to the input node.

Tactical Tactics

The tactics that are constructed from existing tactics. They indicate different strategic or heuristic decisions.

- Apply on All Pending - A tactic to apply a specific sub-tactic to all pending nodes at the point of application. The tactic is successful if the sub-tactic is successful on one of the pending nodes.
- Repeating - A tactic that repeats a specific sub-tactic at the point of application until it fails. The tactic is successful if a sub-tactic is successful at least once.
- Composing Sequential - A tactic to compose a list of sub-tactics that can be applied to the point of application. The tactic is successful if one of the sub-tactics is successful.

More complex proof strategy can be constructed by combining the above tactical tactics.

3.4.4 Provers

In the end, provers perform the actual work. Rodin comes with one prover installed (New PP). It is strongly recommended that you install the third-party provers from Atelier B (as described in Section 3.4.4) in order to add the PP and ML provers. More provers may be available as plugins.

We will now give a very brief overview of the existing provers by pointing out their strengths/weaknesses.

[3] http://wiki.event-b.org/index.php/Rodin_Proof_Tactics

PP

We recommend trying the PP prover first because it is sound and does a pretty good job.

Names in the proof control: P0, P1, PP

Names in the proof tree: PP

Names in the preferences: Atelier B P0, Atelier B P1, Atelier B PP

Input: In the configuration “P0”, all selected hypotheses and the goal are passed to PP. In the configuration “P1”, one lasso operation is applied to the selected hypotheses and the goal and the result is passed to PP. In the configuration “PP”, all the available hypotheses are passed to PP.

How the Prover Proceeds: The input sequent is translated to classical B and fed to the PP prover of Atelier B. PP works in a manner similar to newPP but with support for equational and arithmetic reasoning.

Some Strengths:

- PP has limited support for equational and arithmetic reasoning.

Some Weaknesses:

- PP does not output a set of used hypotheses.
- PP is unaware of some set theoretical axioms.
- PP has similar problems to New PP with regard to well-definedness.
- If unnecessary hypotheses are present, they may prevent PP from finding a proof even when the proof obligation obviously holds.

ML

The ML prover can be quite helpful when the proofs involve arithmetic.

Names in the proof control: M0, M1, M2, M3, ML

Names in the proof tree: ML

Names in the preferences: Atelier B ML

Input: All visible hypotheses are passed to ML. The different configurations refer to the configuration (proof force) of the ML prover.

How the Prover Proceeds: ML applies a mix of forward, backward and rewriting rules in order to discharge the goal (or detect a contradiction among hypotheses).

Some Strengths:

- ML has limited support for equational and arithmetic reasoning.
- ML is more resilient to unnecessary hypotheses than newPP and PP.

Some Weaknesses:

- ML does not output a set of used hypotheses.
- Not all set theoretical axioms are part of ML.

New PP

New PP is unsound. There have been several bug reports. Some have been fixed, but at this point we do not recommend New PP for inexperienced users.

Names in the proof control: nPP R., nPP with a lasso symbol, nPP

Names in the proof tree: Predicate Prover

Names in the preferences: PP restricted, PP after lasso, PP unrestricted

Input: In the configuration "restricted", all selected hypotheses and the goal are passed to New PP. In the configuration "after lasso", a lasso operation is applied to the selected hypotheses and the goal and the result is passed to New PP. The lasso operation selects any unselected hypothesis that have a common symbol with the goal or a hypothesis that is currently selected. In the configuration "unrestricted", all the available hypotheses are passed to New PP.

How the Prover Proceeds: First, all function and predicate symbols that are different from "$\in$" and not related to arithmetic are translated away. For example A $\subseteq B$ is translated to $\forall x \cdot x \in A \Rightarrow x \in B$. Then New PP translates the proof obligation to CNF (conjunctive normal form) and applies a combination of unit resolution and the Davis Putnam algorithm.

Some Strengths:

- New PP outputs a set of "used hypotheses". If an unused hypotheses changes, the old proof can be reused.
- New PP has limited support for equational reasoning.

Some Weaknesses:

New PP is unsound. There have been several bug reports. Most notably, see [1].

What is the [1] here refering to?

- New PP does not support arithmetic; hence, $\vdash_{\mathcal{L}} 1 = 1$ is discharged, but $\vdash_{\mathcal{L}} 1+1 = 2$ is not. Note that arithmetic reasoning (when the formula is not ground) is a long standing challenge.
- New PP is unaware of set theoretical axioms. In particular, $\vdash_{\mathcal{L}} \exists A \cdot \forall x \cdot x \in A \Leftrightarrow x \in B \vee x \in C$ is not recognized because the union axiom is not available within New PP. Roughly spoken, New PP can only reuse sets that already appear in the formula, but it is unable to introduce new sets. Note that set theoretical reasoning is perceived as a hard problem.
- If unnecessary hypotheses are present, they may prevent New PP from finding a proof even when the proof obligation obviously holds. We therefore advise you to unselect unnecessary hypotheses.
- New PP does not take well-definedness into account: Lemma $\vdash_{\mathcal{L}} b \in f^{-1}[\{f(b)\}]$ is not discharged. In fact, this sequent has exactly the same translation as $\vdash_{\mathcal{L}} b \in \mathrm{dom}(f)$, which is not provable.
- New PP tends to run out of memory if the input is large.

3.4.5 How to Use the Provers Effectively

It is very hard, in general, to predict whether a certain automatic prover can or cannot discharge a given proof obligation within a given amount of time. (This is also the case for many other automatic first order theorem provers.) Therefore applying the 11 configurations in a trial and error fashion is often frustrating.

The following guidelines may be useful:

- Add New PP restricted (PP restricted), P0 (Atelier B P0), and ML (Atelier B ML) to the auto-tactic. If the auto-tactic runs out of memory, remove New PP.
- If the model is small, add New PP after lasso (PP after lasso) and P1 (Atelier B P1) to the auto-tactic.

- Whenever you think that the current proof obligation should be discharged automatically, invoke the auto-tactic ( ) instead of some particular automatic prover.
- If the auto-tactic fails, it is usually best to simplify the proof obligation in some way. The most important ways of simplifying the proof obligation are:
  - Remove unnecessary hypotheses; add required hypotheses that have been missing.
  - Do some case splits.
  - Instantiate quantifiers.
  - Apply ae (abstract expression) to replace complicated expressions by fresh variables.
- You can also apply one of the automatic provers. They may be more successful than the auto-tactic because they have a longer timeout.
  - Try New PP before PP or ML because New PP proofs can be better reused, if the model changes.
  - The configurations that act on more than the selected hypotheses (New PP after lasso and unrestricted P1, PP and ML) become useless when the model grows.
- When everything fails, try to unfold the proof obligation manually, by clicking on the red links.
  - You may discover that some assumption was missing.
  - You may complete the proof.
  - If you observe that a valid proof obligation cannot be proved manually, please send a bug report. (Rodin Bug Tracker)

3.4.6 Reasoners

Reasoners apply to the sequent of a given proof tree node and provide a way to contribute to the provers. They are typically of more interest to the developer than the user.

A reasoner is (and has to be) quite “rough” : it takes a given sequent and produces a proof rule that will (if possible) apply to this given sequent. A tactic could be smarter. Indeed, a tactic can use several reasoners by applying them in loops, combining them, or even calling other tactics.

3.4.7 Purging Proofs

Proofs are stored in proof files. Each time a new proof obligation is generated by the tool, a corresponding (initially empty) proof is created. However, proofs are never removed automatically by the Rodin platform. As time passes and a model is worked out, obsolete proofs (e.g., proofs that do not have a corresponding proof obligation anymore) accumulate and clutter proof files.

The purpose of the proof purger is to allow the user to delete obsolete proofs.

Why proofs become obsolete

Proof obligations are named after the main elements related to it, such as events and invariants. Therefore, each time such an element is renamed manually, the corresponding proof obligations get a new name. However, the existing proof is not renamed, and a new proof gets created with the new name.

Consequently, after a lot of model editing, there are more and more obsolete proofs stored in proof files.

Selecting purge input

In any view, right-clicking an Event-B project or file will display a popup menu, with a Purge Proofs... option. If several files or projects (or both) are selected, purging will apply to all of them.

Firstly, the proof purger tries to find obsolete proofs in the selection. If no obsolete proof is found, a message will pop up informing the user that no proof needs to be purged. Otherwise, a new window will pop up displaying a list of all POs that are considered obsolete, i.e. all proofs that exist in some proof file and do not have any corresponding proof obligation.

Choosing proofs to delete

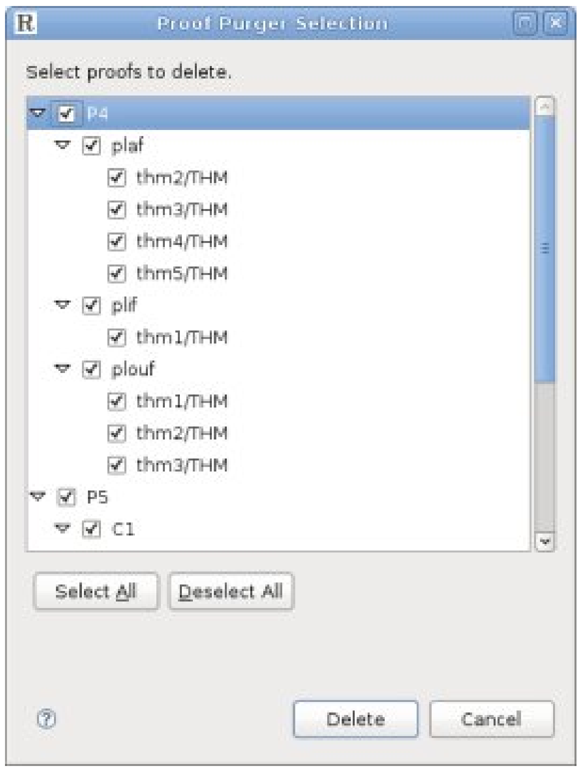

Figure 3.46: Proof Purger Selection Window

For the moment, nothing has been erased. The new window (see Figure 3.46) shows obsolete proofs and allows the user to choose among them and select the ones which should be deleted. One may wish to keep some of them knowing they might be useful in the future.

Once the selection has been decided, clicking the Delete button to actually delete the selected proofs from the proof files. Files that become empty will be deleted as well.

Caution

Proof purging should not be performed on models that are not in a stable state. For instance, it should not be applied to a model that has some errors or warnings issued by the type checker. This is because if there are errors and warnings, not all proof obligations are generated. Therefore, some proofs may have been considered wrongly as obsolete.

Chapter 4

Frequently Asked Questions

4.1 General Questions

4.1.1 What is Event-B?

Event-B is a formal method for system-level modelling and analysis. Key features of event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. More details are available in `http://www.event-b.org`.

4.1.2 What is the difference between Event-B and the B method?

Event-B (Section 2.2.4) is derived from the B method. Both notations have the same inventor, and share many common concepts (set-theory, refinement, proof obligations, ...) However, they are used for quite different purposes. The B method is devoted to the development of *correct by construction* software, while the purpose of Event-B is used to model full systems (including hardware, software and environment of operation).

Event-B and the B method use mathematical languages which are similar but do not match exactly (in particular, operator precedences are different).

4.1.3 What is Rodin?

The **Rodin Platform** is an Eclipse-based IDE for Event-B that provides support for refinement and mathematical proofs. The platform is open source, contributes to the Eclipse framework and can be extended with plugins.

4.1.4 Where does the Rodin name come from?

The Rodin Platform (Section 3.1) was initially developed within the European Commission funded Rodin project (IST-511599), where Rodin is an acronym for "Rigorous Open Development Environment for Complex Systems. Rodin is also the name of a famous French sculptor. One of his most famous works is the Thinker.

4.1.5 Where I can download Rodin?

Rodin is available for download at the Rodin Download page: `http://wiki.event-b.org/index.php/Rodin_Platform_Releases`

4.1.6 How to contribute and develop?

Glad to hear that you want to help! Please see the Developer FAQ page.

4.2 General Tool Usage

4.2.1 Do I lose my proofs when I clean a project?

No! This is a common misunderstanding of what a project clean does. A project contains two kinds of files:

- those you can edit: contexts, machines, proofs
- those generated by a project build: proof obligations, proof statuses (for each proof obligation, discharged or not discharged)

The cleaner just undoes what the builder does, i.e. it removes proof obligations and statuses, but it never modifies any proof.

A status may change from *discharged* to *not discharged* when the proof is no longer compatible with the corresponding proof obligation (e.g. when a hypothesis is changed), but **the proof itself is still there!** You can try to replay it.

Confusion may arise when automatic provers have been launched. The cleaner does not undo these automatic proofs (why would it ?!!). Once a proof is made, the platform does not modify or delete it by itself. Even obsolete proofs are preserved!

4.2.2 How do I install external plug-ins without using Eclipse Update Manager?

Although it is recommended that you install additional plug-ins into the Rodin platform using the Update Manager of Eclipse, this might not always be practical. In this case, you can install these plug-ins by emulating the operations normally performed by the Update Manager either manually or by using ad-hoc scripts.

The manual installation of plug-ins is described in *Installing external plug-ins manually*.

4.2.3 The builder takes too long

Generally, the builder spends most of its time attempting to prove POs. There are basically two ways to get it out of the way:

- the first one is to disable the automated prover in the Preferences panel.
- the second one is to mark a PO as reviewed if you do not want the auto-prover to attempt it anymore.

Note that if you disable the automated prover, you always can run it later on some files by using the contextual menu in the Event-B Explorer.

To disable the automated prover, open Rodin Preferences (menu Window ⟩ Preferences...). In the tree on the left-hand panel, select Event-B ⟩ Sequent Prover ⟩ Auto-tactic. Then, in the right-hand panel ensure that the checkbox labelled Enable auto-tactic for proving is disabled.

To review a proof obligation, just open it in the interactive prover and then click on the *review* button (this is a round blue button with a R in the proof control toolbar). The proof obligation should now labelled with the same icon in the Event-B explorer.

4.2.4 What are the ASCII shortcuts for mathematical operators

The ASCII shortcuts that can be used for entering mathematical operators are described in the following section: Rodin Keyboard User Guide ⟩ Getting Started ⟩ Special Combos.

This page is also available in the dynamic help system. The advantage of using dynamic help is that it is able to display the help page side-by-side with the other views and editors. To start the dynamic help, click Help ⟩ Dynamic Help, then click All Topics and select the page in the tree.

4.2.5 Rodin (and Eclipse) doesn't take into account the MOZILLA_FIVE_HOME environment variable

You have to add a property by appending the following code to your eclipse/eclipse.ini or rodin/rodin.ini file:

```
-Dorg.eclipse.swt.browser.XULRunnerPath=/usr/lib/xulrunner/xulrunner-xxx
```

4.2.6 No More Handles

On Windows platforms, it may happen that Rodin crashes, complaining that there are "no more handles". This is an OS specific limitation, described here and there. A workaround is provided at this site.

4.2.7 Software installation fails

The installation of software from update sites (Help ⟩ Install New Software...) sometimes fails with an error saying something like:

```
No repository found containing: osgi.bundle,org.eclipse.emf.compare,1.0.1.v200909161031
No repository found containing: osgi.bundle,org.eclipse.emf.compare.diff,1.0.1.v200909161031
...
```

This is an eclipse/p2 bug, referenced here.

The workaround is to:

- Go to Window ⟩ Preferences ⟩ Install/Update ⟩ Available Software Sites
- Remove all sites then add them back again, which can be achieved in the Available Software Sites preference page by:
- Select all update sites (by highlighting them all those that are checked)
- Export them
- Remove them
- Restart Rodin
- Go back to the preference page and import update sites back (from the previously exported file)

4.3 Modeling

4.3.1 Witness for Xyz missing. Default witness generated

A parameter has disappeared during a refinement. If this is intentional, you have to add a witness 3.2.3 telling how the abstract parameter is refined.

4.3.2 Identifier Xyz should not occur free in a witness

You refer to Xyz in a witness predicate where Xyz is a disappearing abstract variable or parameter which is not set as the witness label.

4.3.3 In INITIALISATION, I get Witness Xyz must be a disappearing abstract variable or parameter

The witness is for the after value of the abstract variable, hence you should use the primed variable. The witness label should be Xyz', and the predicate should refer to Xyz' too.

4.3.4 I've added a witness for Xyz but it keeps saying "Identifier Xyz has not been defined"

As specified in the 3.2.3 section, the witness must be labelled by the name Xyz of the concrete variable being concerned.

4.3.5 How can I create a new Event-B Project?

Please check Tutorial 2.4.2 to learn how to create a new Event-B project.

4.3.6 How to remove a Event-B Project?

In order to remove a project, first select it on the Project Explorer and then right click with the mouse. The contextual menu will appear on the screen as indicated in Figure 4.1.

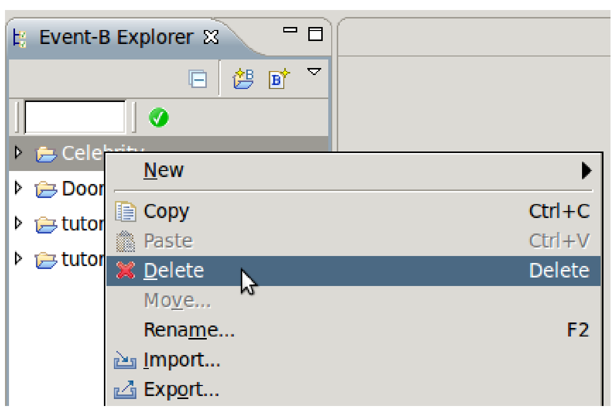

Figure 4.1: Removing a Event-B Project

You simply click on Delete and your project will be deleted (after you confirm it in the window that pops up). It is then removed from the Project Explorer.

4.3.7 How to export an Event-B Project?

Exporting a project is the operation by which you can construct automatically a "zip" file containing the entire project. Such a file is ready to be sent by mail. Once received, an exported project can be imported (next section). It then becomes a project like the other ones which were created locally. In order to export a project, first select it, and then click on File ⟩ Export... from the menubar as indicated in Figure 4.2.

The Export wizard will pop up. In this window, select General ⟩ Archive File and click the Next > button. Specify the path and name of the archive file into which you want to export your project and finally click

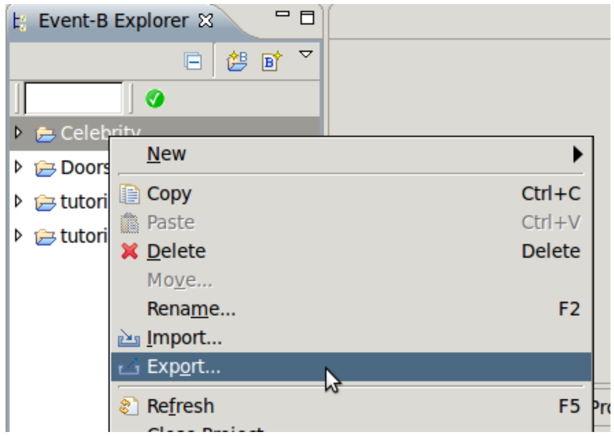

Figure 4.2: Export a Event-B Project

Finish. This menu sequence belongs to Eclipse (as well as the various options). For more information, refer to the Eclipse documentation.

4.3.8 How to import a Event-B Project?

A ".zip" file corresponding to a project which has been exported elsewhere can be imported locally. In order to do this, click on File 〉 Import from the menubar. In the import wizard, select General 〉 Existing Projects into Workspace and click Next >. Then, enter the file name of the imported project and finally click Finish. As with exporting, the menu sequence and layout are part of Eclipse.

The importation will fail if the name of the imported project (not the name of the file) is the same as the name of an existing local project. The moral of the story is that when exporting a project to a partner you need to modify its name in case your partner already has a project with that same name (which could be a previous version of the exported project). Changing the name of a project is explained in the next section.

4.3.9 How to change the name of a Event-B Project?

Select the project whose name you want to modify, and then click on File 〉 Rename.... Modify the name and click on OK. The name of your project will then have been modified accordingly.

4.3.10 How to create a Event-B Component

Please check Tutorial 2.4.2 to learn how to create a new Event-B component.

4.3.11 How to remove a Event-B Component

In order to remove a component, press the right mouse button on the component. In the context menu, select Delete. This component is removed from the Project Explorer.

4.3.12 How to save a Context or a Machine

Once a machine or context is (partially) edited, you can save it by using the save button as indicated in Figure 4.3.

Once a "Save" is done, three tools are called automatically, these are:

- the Static Checker

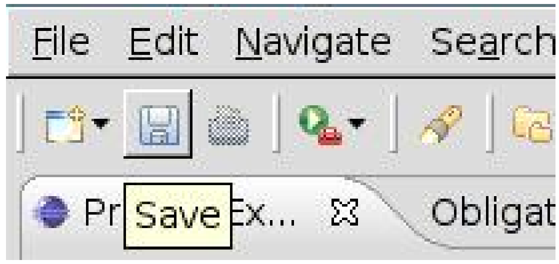

Figure 4.3: Save a context or a machine

- the Proof Obligation Generator (Section 3.2.6)
- the Auto-Prover (Section 3.1.6)

This can some time. A "Progress" window can be opened at the bottom right of the screen to see which tools are working (most of the time, it will be the auto-prover).

4.4 Proving

4.4.1 Help! Proving is difficult!

Yes, it is. Check out Section 3.4.5 to get started with using the provers.

4.4.2 How can I do a Proof by Induction?

This page about proof by induction will give you some starting tips.

4.4.3 Labels of proof tree nodes explained

- ah means *add hypothesis*,
- eh means rewrite with *equality from hypothesis* from left to right,
- he means rewrite with *equality from hypothesis* from right to left,
- rv tells us that this goal has been manually reviewed (see 3.1.5),
- sl/ds means *selection/deselection*,
- PP means *discharged by the predicate prover*,
- ML means *discharged by the mono lemma prover*

4.5 Usage Questions

4.5.1 Where did the GUI window go?

When you are looking for a particular view, and the view does not appear or if it appears in a different place than is usual, try clicking on Window ⟩ Reset Perspective.... This will reset the different views back to their default positions. If you can't find menu buttons from one of the views, try resizing the view in question to see if part of the menu has been hidden.

4.5.2 Where vs. When: What's going on?

You may have noticed that both in this tutorial, as well as in the tool, events sometimes use the keyword "when" and sometimes "where". The idea of this was to make the formal statements more intuitive. Unfortunately, this created more confusion than anything else.

The short answer is: "when" and "where" in events have exactly the same meaning, for all practical purposes.

The long answer is: In some contexts (but not all), the tool changes the keywords to make the meaning of the event more apparent. The distinguishing factor is the parameter: Without parameter, "when" is used, with a parameter, "where" is used.

To confuse things even more, this doesn't apply everywhere: The structural editor always shows "where", but the pretty print toggles between the two. The Event-B in this handbook has been generated with the LaTeXplugin, which also toggles the keyword.

Index