

# Getting Started with Rodin 0.9.2: The *SetProject* Development

R. Banach  
School of Computer Science,  
University of Manchester,  
Manchester, M13 9PL, U.K.

N.B. Occasionally, the *Edit* or *Pretty Print* window (see below) may stop working. If this happens Save (see below), exit Rodin, and restart. This usually solves the problem. If not, restart Windows XP.

## 1 First Steps

### 1.1 Windows XP

Log in to Windows. Select *Rodin* > *Rodin* from the *All Programs* menu. When Rodin starts it offers to build a workspace. **If** this is in the *All Programs* directory structure or other system-owned area, do not accept the suggestion (since you have no permission to create files there). Instead use the *Browse* button to create a directory (called eg. *workspace*) on your P: drive (or other suitable place).

When *Rodin* starts, you see the *Welcome* tab. The lower half of this has **Important Installation Notes**. The first time Rodin is used, follow the procedure described there to install the Atelier-B Provers. On subsequent occasions, check that the provers are there. If not, install them (after its first appearance, the *Welcome* tab can be recovered from the top level *Help* menu).

### 1.2 Linux

N.B. The linux installation has been removed! Log in to `linux`. In a shell, at the command prompt, type (or set up an alias for yourself for):

```
/opt/rodin/rodin/rodin &
```

Rodin starts up. Rodin will offer you a workspace. Choose it or an alternative.

### 1.3 Once you're up and running ...

The main Rodin Welcome tab then appears. Assuming the Atelier-B Provers have been installed, click the *delete* cross in the tab to reveal the Rodin main screen.

The Rodin main screen is divided into four main areas: the *Event-B Explorer* window on the left (showing the components of the current project); the *Outline* window on the right (showing the project outline); the *Rodin Problems* window at the bottom (giving feedback, error messages etc.); and, most importantly, the *Main* window in the middle (which is where most of the interaction with the user takes place during development).

## 2 Creating the *SetProject* Project

The Rodin main screen should be completely empty (aside from the above structure). Only the *Event-B Explorer* window is active. Hover the mouse over its active icons, and see that *Create new project* and *Create new component* are available actions. Left-click *Create new project*. A dialog box appears, containing a suggested name for the new project. Edit it to say '*SetProject*' and click *Finish*.

## 3 Building the *SetProject* Context

Thus far, *SetProject* is an empty shell. In the *Event-B Explorer* window, click on *Create new component*. A dialog box appears. It contains a text entry box with a *Browse* button beside it, a suggested name for the new component and two radio buttons for the kind of component: *Machine* or *Context*. Use the *Browse* button to choose the *SetProject* project. Enter the name '*MySet*' in the *Component name* box, click *Context* (for the type of component), and click *Finish*. Rodin analyses the input and creates the *MySet* Context component.

The *Main* window now becomes active, acquiring a tab at the top called *MySet*, with a C in a purple disk indicating that it's a Context component. This allows you to work on the *MySet* component. There is a row of tabs at the bottom of the main window *MySet* pane: *Pretty Print*, *Edit*, *Synthesis*, *Dependencies*. The *Edit* tab is open and shows the (so far empty) structure of the *MySet* component. Click the *Pretty Print* tab. You see the embryonic form of the *MySet* component in text form.

Unfortunately, *MySet* ought to be the name of the set to be introduced, not the name of the component. Click the  $\boxplus$  explore icon to the left of *SetProject* in the *Event-B Explorer* window to reveal the *MySet* component as a child of *SetProject*. Click the *MySet* child to highlight it. Right-click the highlighted *MySet* to reveal a menu of actions. Select *Delete* and click *Yes* in the confirmation dialog box. The *MySet* component disappears.

Following the procedure described before, create a new *Context* component called *MyCtx*. Check its embryonic form via *Pretty Print*.

Go to the toolbar just above the *Main* window. On the right there is a collection of six icons. Hover the mouse over them. You should see (left to right): *New Theorems Wizard*, *New Axioms Wizard*, *New Constants Wizard*, *New Enumerated Set Wizard*, *New Carrier Sets Wizard*, *Generate Latex document*. The first five of these also correspond to the items to be found in the *Event-B* menu in the top level menu bar, just above. Click on *New Enumerated Set Wizard*. A dialog box appears. It contains a suggested name for the new enumerated set, which you should replace with 'MySet', and a series of slots for elements; fill in elements 'a', 'b', 'c', click *More Element* and add 'd' too. Click *OK*. See all the new stuff that appears in the *Pretty Print* window. If you look at the *Edit* window then nothing much seems to have changed ... until you click on the explore icons to the left of SETS, CONSTANTS, AXIOMS, whereupon you will see the new stuff that was visible in the *Pretty Print* window, but this time in editible form — N.B. do not edit anything! Note also the  $\boxplus$  and  $\boxminus$  icons to the left of the construct name at the top of the *Edit* window — these reveal/conceal all the inner structure of the whole construct in one click. Note also how the *MyCtx* name in the tab has acquired an asterisk against it, showing that it's an unsaved component.

Click on *New Carrier Sets Wizard*. Accept its suggestion (*set1*) for a single new carrier set by clicking in the string and typing and then deleting some character (so that the string turns yellow — while it remains blue or uncoloured no action will be taken). See the effect via the *Pretty Print* window — only the set itself appears, and since no elements are named, no axioms are generated for them. In the *Edit* window the SETS section opened and the new set is visible. Various icons are visible in such a section, such as: crosshairs to allow an item to be selected, arrows to allow a selected item to be moved up and down the list, green  $\oplus$  icons to allow new elements to be introduced at the beginning or end of the list (so that's another way of introducing new stuff), and red  $\ominus$  icons to allow elements to be deleted. Click the red  $\ominus$  icon against *set1*. It disappears.

Go to the *File* menu in the top level menu bar. Click *Save*. (Alternatively click the standard *Save* icon in the main toolbar). If you are lucky and the system is slow, you will see Rodin briefly indicate the saving activity that ensues via a progress bar in the bottom right hand corner of the tool. As Rodin saves the component, the asterisk against the *MyCtx* name in the main window *MyCtx* pane, which indicates that it is unsaved, disappears.

## 4 Building the *SetProject* Abstract Machine

In the *Project explorer* window, click on the *SetProject* entry to highlight it. Then click on *Create new component*. A dialog box appears. Note that it has *SetProject* already filled in. Click *Machine*, and enter the name 'MyMchA', and click *Finish*. Rodin creates the *MyMchA* Machine component. The main window acquires a new pane. It has a *MyMchA* tab, and an M in a blue disk (denoting a Machine component). The focus changes to the *Edit* window of *MyMchA*. Click on *Pretty Print* to see the embryonic form of *MyMchA*. It just contains an initialisation, which is an 'ordinary' event, whose action is set to *skip*.

To enable *MyMchA* to use the concepts in *MyCtx*, it must SEE it. Click on the *Dependencies* tab (of *MyMchA*). The *Dependencies* tab contains two parts: *Abstract Machine* (to tell Rodin which abstraction the new machine refines ... in our case none, so ignore it), and *Seen Contexts* (which is what we want). In the *Seen Contexts* part, click on the down-arrow to the right of the empty input line near the bottom (you may have to scroll the whole *Dependencies* tab window down to reveal this). A menu of potential seen contexts appears. Click on *MyCtx* (which is the only thing in the menu). *MyCtx* appears in the *Seen Contexts* window, and an asterisk appears against the *MyMchA* name in the tab at the top to show that *MyMchA* is unsaved. Click on *Pretty Print* to see the embryonic form of *MyMchA* evolve to now encompass the reference to *MyCtx*. The *Edit* window now shows the seen context too, under the SEES section.

At the bottom of the main window *MyMchA* pane, the tabs have changed. Note that we now have: *Pretty Print*, *Edit*, *Synthesis*, *Dependencies*.

Above the main window, the Wizards have changed. Hover with the mouse to confirm: *New Variant Wizard*, *New Invariants Wizard*, *New Variable Wizard*, *New Theorems Wizard*, *New Event Wizard*, *Generate Latex document*. Observe that the first five also correspond to the items in the *Event-B* menu in the top level menu bar. Click *New Variable Wizard*. A dialog box appears, containing three input areas: *Name*, *Initialisation*, *Invariant*. Replace the *Name* of the variable with ‘x’, observing how the other references to the name change in parallel with it. Next, go to the *Initialisation* line. It has two parts. The left hand one is the *name* of the initialisation line. This is useful for Rodin internal bookkeeping. Although you may change it, there’s no need — Rodin has a well thought out internal convention for naming things, that it applies automatically. The right hand one is the initialisation action itself. Click on the end of the line (to get the focus there, ready to type), and then type ‘{}’ You note that ‘∅’ appears — so when you type in the ascii form of B AMN, it is automatically and instantaneously converted to the display form. You should now see:

Initialisation

Now go to the *Invariant* line, type ‘POW MySet’ ... ‘ $\mathbb{P}$  MySet’ appears. You should now see:

Invariant

Click on *OK*. Note how the changes are immediately reflected in the *Pretty Print* window. Change to the *Edit* window.

Go to the *File* menu in the top level menu bar. Click *Save* (or ... etc.). Rodin *saves and analyses* your work whenever you click *Save*. In this case, you see some sections of the *Edit* window light up in red, and some errors appear in the *Problems* window at the bottom. The first one says something about LPAR missing. Double click on the error. Suddenly the red highlighted sections of the *Edit* window open up, and the focus pinpoints *MySet* in *inv1* of the INVARIANTS section. That’s where that particular problem lies. Type a pair of parentheses around ‘MySet’ in ‘inv1’, which should now look like ‘x ∈  $\mathbb{P}$  (MySet)’. If you click on *Pretty Print* you can see the change reflected immediately. However the errors are still there in the *Problems* window. Click *Save* again in the *File* menu. This now commits the changes. The amended structure is re-analysed, and you see that the errors disappear.

## 5 Adding an Event

It is time to make *MyMchA* do something beyond mere initialisation. Click on the *New Event Wizard*. A dialog box appears. In *Label* replace ‘evt1’ with ‘AddE1’. In *Parameter Identifier(s)* type ‘e1’ in the first box. In the *Guard predicate(s)* column, in the entry belonging to ‘grd1’, type ‘MySet’ after the ‘e1 ∈ ’ which appeared the instant you typed the name ‘e1’. In the *Action substitution(s)* part, in the second column, type ‘x := \/{e1}’ in the entry belonging to ‘act1’. Note how it instantly changes to ‘x := ∪ {e1}’. Click *OK*. Click *Pretty Print* to see how *AddE1* has now appeared in the EVENTS part. Click on the *Synthesis* tab to see the same data that *Pretty Print* shows you, but in abstract syntax tree form. Click on the explore icons on the left to reveal/conceal the deeper structure. *Save* once more. A syntax error appears in the *Problems* window. Double click on it to be taken to the line you input: ‘x := ∪ {e1}’ in the *Edit* window. Change this by adding an extra ‘x’ between the ‘:=’ and the ‘∪’ so that it reads ‘x := x ∪ {e1}’. *Save* once more. Now there should be no errors.

Go to the top level menu bar, and drop down the *Window* menu near the right hand end. Hover over *Open Perspective*, and click on *Proving*. The windows (in a counterclockwise sense) round the periphery of Rodin change to: *Proof Tree*, *Proof Control* and *Goal, Event-B Explorer*. The first three don’t contain anything. The system so far is trivial enough that Rodin was able to prove everything needed by itself, behind the scenes, leaving no visible trace. At the right hand end of the toolbar containing the Wizards is a green tick icon and the word *Proving*. It used to say *Event-B*. To its immediate left is a grey box icon, whose tooltip says *Open Perspective*. Click it. A menu appears, containing *Proving*, *Resource* and *Other*.... Click *Other*. Select *Event-B* in the dialog box that appears and click *OK*. The perspective reverts to the previous one. (A similar effect can be achieved by clicking on the double guillemet at the extreme right of this toolbar, and then selecting *Event-B* once more.)

## 6 Making the Prover Show its Face

So far there has been no visible activity from the prover. Let’s provoke it now. In the *Event-B* perspective, in the main window, click on the *MyMchA* tab to highlight it. Click the *New Invariants Wizard*. In the box for

'inv2', type ' $d : x$ '. Marvel how it changes to ' $d \in x$ ' before your very eyes. Click *OK*. Check how it appears in *Pretty Print* and *Edit*. Click *Save* once more.

Think about what is going on. Since  $x$  is initialised to  $\emptyset$ , there is no reason why  $d \in x$  should be a valid invariant. However there is no sign that anything is amiss in the *Event-B* perspective. Change to the *Proving* perspective. Still not much sign of anything wrong. In the *Event-B Explorer* window, explore *MyCtx*. At the bottom of the list is a green disk icon with a white tick inside next to *Proof Obligations*. This signifies that there is nothing wrong here. Now explore *MyMchA*. At the bottom of the list is a red disk icon with a white question mark inside next to *Proof Obligations*. This signifies that there *is* something wrong here. Explore this. You should see a subtree of two obligations marked red.

At the top of the *Event-B Explorer* window is a green disk with a tick inside it. Clicking it toggles the display of proved 'non-trivial' obligations. However, all the proved obligations so far (and there are indeed some) are trivial enough that Rodin does not show them. So clicking the green tick in fact achieves nothing except some spurious manipulation of explore icons.

Click on the red obligation called INITIALISATION/inv2/INV. There is a brief flurry of activity as Rodin sets up a new proof attempt, the main visible consequences of which are that a new main window pane has been created, with tab *MyMchA* and containing a P in a brown disk, indicating a *Proof Attempt* component. The new main window is for listing the *Selected Hypotheses* for the current state of the proof attempt in progress. Something also appears in the *Proof Tree*, *Proof Control* and *Goal* windows.

The *Goal* window shows the current goal, namely  $\perp$ . This is obviously unprovable (i.e. it denotes false). So the *Proof Control* window below has a red frown in it denoting a failed proof attempt. The *Proof Tree* window on the left shows how we arrived at the current state of affairs. It has a two level proof tree, the leaf of which is the  $\perp$  goal, and which is highlighted. Above  $\perp$ , is a branch that says 'simplification rewrites'. Click on it. The *Goal* window changes to ' $d \in \emptyset$ ', which was the original invariant, and the predecessor goal of  $\perp$ . Now Rodin has a rewrite rule that says every element is *not* a member of the empty set ( $d$  included), so it is not surprising that when the rule is applied, a contradiction is derived. The empty *Selected Hypotheses* main window shows that no extra information was required in order to apply this rule, so it was applied automatically. In the *Proof Tree* window, there are in addition various icons at the top, chiefly: the  $\textcircled{G}$  (goal) icon, and the  $\oplus$  and  $\ominus$  icons which allow expansion and contraction of the proof tree.

In the *Proof Control* window locate the  $\textcircled{i}$  icon and click it. The *Proof Information* window appears, containing the relevant fragments from the source that generated the mathematical goal, i.e. ' $d \in \emptyset$ '. In this case, these are the texts of the INITIALISATION event and of the inv2 invariant. Clicking the delete cross in the *Proof Information* window's tab causes it to disappear, exposing the window that it overwrote when it appeared. Since this proof obligation is clearly unprovable, click the delete cross in the main window P tab. The proof attempt is deleted.

Now click on the red obligation called AddEl/inv2/INV. Rodin sets up a new proof attempt as above. The name of the PO indicates what kind of a statement we are trying to prove. It is a PO coming from an invariant (INV), specifically originating from invariant inv2, which in turn originates from event AddEl. As such it seeks to establish that the invariant holds for the updated state of the event. Click the  $\textcircled{i}$  icon in the *Proof Control* window. The *Proof Information* window that appears contains the source material that generates the statement whose truth we are seeking to establish. We see the source of the AddEl event, and the invariant in question (which is ' $d \in x$ '). It is thus not surprising that the goal is ' $d \in x \cup \{el\}$ '. What *is* perhaps surprising, is that Rodin has not managed to deal with it unaided.<sup>1</sup> If you click on the root goal of the proof tree (in the *Proof Tree* window), you see the same goal, so not a lot has been accomplished.


Note the red ' $\in$ ' in the goal statement in the *Goal* window. Red symbols in the goal window indicate places in the goal statement where some work can be done. Hover the mouse over it. The tooltip says 'Remove membership in goal'. This means: rewrite the goal to reduce the ' $\in$ ' to simpler logical elements. Click the red ' $\in$ '.

Immediately, the situation improves. A lot of stuff turns green. In particular the red frown is now a green smiley — this says that the proof attempt has succeeded. The proof tree is now full of green ticks (if you expand it). In particular, its root now says 'remove  $\in$  in goal', which is what the click just did.

In fact, behind the scenes, removing the ' $\in$ ' rewrote the goal to  $d \in x \vee d \in \{el\}$  — this is now a disjunction between one of the hypotheses and something else, and Rodin's prover is smart enough to figure out the rest. You can confirm this by clicking the  $\oplus$  explore icon against the root of the proof tree, (if you didn't expand it

---

1. An earlier release of Rodin did indeed manage to dispose of this by itself! However, for illustrative purposes, it's better that it doesn't.

before). The new leaf says ‘ $\forall$  goal in hyp’ (theses). Repeatedly click the  icon in the *Proof Tree* window’s header bar. It reveals/conceals the rewriting of the goal as just discussed.

Save the current proof attempt in the usual manner. Look in the *Event-B Explorer* window. Depending on the current setting of the green ‘display proved non-trivial obligations’ tick, the **AddEl/inv2/INV** obligation may or may not be visible. Clicking the tick reveals/conceals it, and when it is exposed, it now has a green tick against it showing that it has been successfully discharged.

You will not be surprised to learn that there is a lot more to the business of discharging proof obligations than we have seen thus far (as you might guess from the many icons not yet discussed in the *Proof Control* window), but what we have seen will do for now.

Go back to the *MyMchA* tab. Select the ‘ $d \in x$ ’ invariant in the *Edit* tab and delete it. Save. *MyMchA* goes back to normal. In particular, you can check that there are now no unproved obligations to be seen in the *Event-B Explorer* window.

## 7 Refining the *SetProject* Abstract Machine



Let’s develop the system further. Before clicking more buttons, let’s overview the basic idea of what we are going to do.

At the abstract level, i.e. the level of *MyMchA*, the idea was that the state variable,  $x$ , was an updatable set, and that the only action that you could perform on it, was to add another element to it. Unfortunately, sets as such, are not directly implementable, so one thing we could do is to translate the abstract notions of *MyMchA* into something closer to what can be achieved on an actual machine. The essence of this is going to be that we represent the set as a natural number,  $y$ , such that every distinct prime number in the unique prime factorisation of  $y$  represents the presence of a specific element of *MySet* in  $x$ . Since there are four elements in *MySet*, we need four different prime numbers to represent them inside  $y$ . Let’s choose the four smallest 2, 3, 5, 7, for  $a, b, c, d$ , respectively. Adding an element of *MySet* to  $x$  now corresponds to multiplying  $y$  by the relevant factor. And the set that a given number (built up in this way) represents, is revealed by which of the prime numbers 2, 3, 5, 7, divide it exactly (i.e. without remainder). Note that just as you can add the same element to  $x$  several times without changing  $x$  (after the first time), so you can multiply a number by the same prime several times without changing which primes divide it exactly (after the first time). We will come back to this detail in due course.

Go to the *File* menu in the top level menu bar and select *New*, then *Event-B Component*. If the *SetProject* project is not highlighted, then you will have to either go back and highlight it, or specify it as the container of the new component. Create a new *Machine* component called *MyMchC*. In the *Dependencies* tab that opens, in the *Abstract Machine* part, select *MyMchA* as the *Abstract machine* for *MyMchC* by clicking the down-arrow at the end of the input line, and clicking on the menu that appears. Similarly select *MyCtx* in the *Seen Contexts* part.

Aside: In many Event-B developments, a refinement step (as we have just embarked on) amounts to simply adding more stuff to a machine. In such a case you want a convenient way of creating a refining machine with all the abstract stuff already in it. This can be done in Rodin by selecting the abstract machine in the *Event-B Explorer* window, right clicking, and choosing *Refine* in the menu. This invites you to input a name for the new (concrete) machine, and then copies the contents of the abstract machine into it, ready for further editing. In the case of our case study, this would be counterproductive since we don’t want any of this stuff inside the refining machine, so this technique would just cause us to have to delete it all — the more low level technique described above for creating *MyMchC* is more efficient in this case. End of Aside.

Open the *New Variable Wizard* in *MyMchC* and create a variable  $y$ . Complete the initialisation line with ‘1’, and the invariant line with ‘ $\text{NAT}$ ’. No magic happens as you type, but *Pretty Print* has ‘ $y \in \mathbb{N}$ ’ in the **INVARIANTS** part. Open the *New Event Wizard*. Create an event called ‘**AddE1a**’, with *Parameter Identifier(s)* ‘ $e1$ ’. Complete the predicate line for ‘**grd1**’ by typing ‘ $\text{MySet}$ ’ and in the predicate line for ‘**grd2**’ type ‘ $e1 = a$ ’. In the input line for ‘**act1**’ type ‘ $y := y * 2$ ’. Click *OK*. If you like, check your handiwork in *Pretty Print*.

Go to the *Edit* tab. Locate the **EVENTS** section and open it via the  explore icon. Open the **AddE1a** event via its explore icon. Open the **REFINES** subsection via its explore icon. Click the green  disk. From the selectable menu that appears, select **AddEl** (i.e. *not* **AddE1a**).

What you have done is the following. When a machine is refined, we have to know which concrete events refine which abstract ones. The connection is established via the **REFINES** subsection that you just filled in. So the **AddE1a** event in *MyMchC* is declared as refining the **AddEl** event in *MyMchA*.

Go to the *Synthesis* tab. Select the **AddEla** event. Right click the lilac star against the **AddEla** event name, and select *Copy*. Find a clear area of the *Synthesis* pane, and right click, this time selecting *Paste*. Another instance of **AddEla** appears. Open up its structure via the  $\boxplus$  explore icon. Select the name **AddEla**. Change the ‘a’ to ‘b’ so the name becomes **AddElb**. Similarly change the ‘a’ to ‘b’ in **grd2**, and the ‘2’ to ‘3’ in **act1**. Observe that **AddElb** also refines **AddEl** since we didn’t change that bit.

In a similar manner *Paste* another copy of **AddEla** but now change ‘a’ to ‘c’ and ‘2’ to ‘5’, getting **AddElc**. Finally *Paste* yet another copy, and change ‘a’ to ‘d’ and ‘2’ to ‘7’, getting **AddEld**. Save *MyMchC*. If you now check the *Proving* perspective, you will see some proof obligations that have been successfully autoproved ... and (most likely) some that have not been autoproved.

If you go to the *Proving* perspective and open one of the unproved obligations, it amounts to something trivial, like ‘ $y * 2 \in \mathbb{N}$ ’. In the tool bar of the *Proof Control* window, there is an icon, which (on close inspection) looks vaguely like the head and shoulders of a green robot; its tooltip says *Auto provers : rerun the auto provers* (it’s next to the screwdriver icon). Click it. After a brief flurry of activity the goal is discharged.<sup>2</sup> Save the proof attempt. The relevant proof obligation in the *Event-B Explorer* window turns green.

Back to the main thread. A little thought shows that the four events **AddEla**, **AddElb**, **AddElc**, **AddEld**, take care (at the concrete level, and in terms of the numerical representation) of the various possibilities offered by the abstract event **AddEl**. So we have the connection between the events in *MyMchA* and *MyMchC*, and have communicated these to Rodin via the *REFINES* details. However, the logical connection between the structures of the state spaces in *MyMchA* and *MyMchC* is still very weak, resulting in feeble proof obligations. Let’s improve this now.

## 8 Adding Gluing Invariants

What is missing in the development so far is that Rodin has no idea that presence of **a** in **x** is supposed to correspond to exact divisibility of **y** by 2; **b** in **x** corresponds to exact divisibility of **y** by 3; and similarly for **c** and 5, and **d** and 7. Facts such as these ought to be invariants. In *Event-B*, such invariants, which express joint properties of an abstract machine and its more concrete refinement, are deemed to live in the concrete machine.

In *MyMchC*, go to the *Edit* tab and open the **INVARIANTS** section. Click the green  $\oplus$  icon at the bottom to add a new invariant called ‘*inv2*’. Overwrite the **true** string with the string ‘ $a : x \Rightarrow y \bmod 2 = 0$ ’. Try to contain your astonishment as it instantly becomes ‘ $a \in x \Rightarrow y \bmod 2 = 0$ ’. Select the string ‘ $a \in x \Rightarrow y \bmod 2 = 0$ ’ that you just typed, and type *Ctrl-C*. Click the green  $\oplus$  icon at the bottom again to create ‘*inv3*’. Select the **true** string and type *Ctrl-V*, and see the string appear in the input line of the new invariant. Alter the **a** to **b**, and the 2 to 3. Click the green  $\oplus$  again and alter the **a** to **c**, and the 2 to 5. Click the green  $\oplus$  again and alter the **a** to **d**, and the 2 to 7. Save *MyMchC*. Saving now takes rather longer than previously.

Aside: We could have strengthened the statements that we made to (say) biimplications eg. ‘ $a \in x \Leftrightarrow y \bmod 2 = 0$ ’ etc., but since the four abstract possibilities are mutually exclusive and exhaustive (for membership of **x**), and the four concrete possibilities are mutually exclusive and exhaustive (for exact divisibility of **y** by 2, 3, 5, 7), we judge that what we input is sufficient. The stronger biimplication alternative is yet another option in contrast to: on the one hand, the trivial relationship between the state spaces of the two machines that we started with; and on the other, the invariants we actually input. In general this illustrates a universal truth about this kind of formal development process. The weaker the statements we make about the relationship between the two machines, the less there is to prove, and consequently, the less that the tool can check about this relationship. Conversely, the stronger the statements we make about the relationship between the two machines, the more work there is to do to prove all these statements, but also, the more there is that the tool can check about the relationship. Note however that the choice of such invariants etc. is a judgement call, i.e. **it is a human decision**. No formal methodology can ever tell you *what* is the right decision to make regarding the properties that the system being developed ought to have. You have to take responsibility for that. The formal methodology can only ever help you pursue the consequences of your decisions in a correct manner. End of Aside.

Returning to *MyMchC*, if you check the *Proving* perspective, you will see that a large number of new proof obligations have been generated and attempted. A few have been successfully discharged, but not many.

## 9 Playing with the Prover Some More

The current release of the prover is not particularly good at arithmetic (though it’s slightly better than it was).

2. So why didn’t the autoprover manage to do it previously? The autoprover is organised in (a small number of) layers, each more powerful than its predecessor. It may be that the first couple of runs only called the more feeble layers.

Click on the unproved obligation (AddEla/inv3/INV). The *Goal* window says ' $(y*2) \bmod 3 = 0$ '. The fact that you multiplied the number  $y$  by 2 offers no guarantee that the result is exactly divisible by 3! However there are some potentially helpful hypotheses in the *Selected Hypotheses* window above. Notably, one hypothesis says that ' $y \bmod 3 = 0$ ' provided that ' $b \in x$ ' holds ... (or what it literally says is that ' $b \in x \Rightarrow y \bmod 3 = 0$ '). Now ' $y \bmod 3 = 0$ ' is enough to guarantee that ' $(y*2) \bmod 3 = 0$ ' by arithmetic, so let us see if we can convince ourselves that ' $b \in x$ ' holds. We have a hypothesis that ' $b \in x \cup \{a\}$ ', so since  $a$  and  $b$  are certainly not the same,  $b$  must be an element of  $x$ . Let us see if we can persuade Rodin of at least this much, which would reduce the proof to pure arithmetic, (which Rodin won't be able to proceed much further with).

Click the red ' $\in$ ' in ' $b \in x \cup \{a\}$ ' in the *Selected Hypotheses* window. It turns into ' $b \in x \vee b = a$ '. Hover the mouse over the red ' $\vee$ ' in ' $b \in x \vee b = a$ ', and you see that you have a choice of two options. Click **Proof by cases**. The proof branches into two subproofs. If you look in the *Proof Tree* window, you see that the upper branch is highlighted; so that is the subproof that is being displayed in the *Selected Hypotheses*, *Goal*, and *Proof Control* windows. The goal says ' $(y*2) \bmod 3 = 0$ ' as before, and now we have a hypothesis ' $y \bmod 3 = 0$ ' as we hoped for. So we have reduced this branch to pure arithmetic. While Rodin can do a certain amount as regards arithmetic which involves just plus, minus and comparison, the inclusion of multiplication and/or division makes arithmetic statements involving them undecidable in general, curtailing what Rodin might achieve, even in the ideal case. Since our statements involve modulus, it is not so surprising that Rodin doesn't make much headway (even though we ourselves can figure it out pretty easily). Anyway, we have come as far as we will be able to go with this branch.

In the *Proof Tree* window, the lower branch has a  $\boxplus$  explore icon against it. Open up the tree. Another instance of the unproved ' $(y*2) \bmod 3 = 0$ ' goal is revealed — we seem to be going backwards and creating even more work for ourselves. Click this fresh instance of the goal. The *Selected Hypotheses*, *Goal*, and *Proof Control* windows now display the state of affairs for this goal. Note that the *Selected Hypotheses* are different. Now they contain ' $b=a$ ', which is obviously nonsense. If we have a false hypothesis, then there is nothing more to do, since a proof obligation is only required to hold *provided all its hypotheses are true*. Hover the mouse over the rightmost group of icons in the tool bar of the *Proof Control* window until you find the one whose tooltip says *Search Hypotheses*. Click it. The *Search Hypotheses* window opens. Near the top you see ' $\neg a=b$ '. So Rodin knows that  $a$  and  $b$  are different, yet seems unperturbed that ' $b=a$ ' is a hypothesis of the current PO. This situation illustrates a general feature of automated theorem proving. The two facts, ' $b=a$ ' and ' $\neg a=b$ ' are simply sitting in data structures inside Rodin. Until a time when they are located and processed by a suitable algorithm, Rodin will be none the wiser that a contradiction is lurking inside the current proof structure.

Click the delete cross of the *Search Hypotheses* window to get rid of it. Find the green robot icon in the *Proof Control* window and click it. The lower branch of the proof tree in the *Proof Tree* window now turns green, with ticks, and the ' $b=a$ ' hypothesis disappears. Running the autoprovers is what was needed for Rodin to locate the two contradictory hypotheses and infer the consequences. This branch of the original proof obligation has been successfully proved by Rodin.

We know that Rodin cannot do the remainder of the proof. The only thing we can do, since we obviously believe that ' $(y*2) \bmod 3 = 0$ ' is correct if ' $y \bmod 3 = 0$ ' holds, is to *Review* the goal. *Reviewing* a goal declares it to be true to Rodin without further ado. Go to the *Proof Control* window and click the blue  $\textcircled{R}$  disk icon in the *Proof Control* window's tool bar. The red frown turns into a blue smiley, and both the proof tree and proof obligation (in the *Proof Tree* and *Obligation Explorer* windows respectively) turn blue also. Rodin now accepts that this proof obligation is true by virtue of the fact that the user has said so. Save.

If you examine any of the other unproved obligations, you see similar situations to the ones described. You can examine the origins of the obligations by clicking the  $\textcircled{i}$  icon to open the *Proof Information* window. And you can *Review* them if you wish, though little is accomplished by doing so.

## 10 Adding 'Non-Refining' Events

So far so good. Now consider the following (recalling remarks above). Adding a given element to a set is an idempotent operation (i.e. doing the same thing several times is equivalent to doing it just once; the additional iterations have no effect). However, multiplying a number by say 2 is *not* idempotent; every time you multiply by 2, the number gets bigger. So there is some redundancy in the way that the number  $y$  represents  $x$ . We can keep this under control by introducing events to remove duplicate factors of our chosen primes 2, 3, 5, 7.

Open the *New Event Wizard*. Call the new event 'Prune2'. Give it two guards: ' $y \bmod 2 = 0$ ' (which becomes ' $y \bmod 2 = 0$ '), and ' $(y / 2) \bmod 2 = 0$ ' (which magically transforms to ' $(y \div 2) \bmod 2 = 0$ '). Make the action ' $y := y / 2$ ' which becomes ' $y := y \div 2$ '. Click *OK*.

Go to the *Edit* window and open the **EVENTS** section. For the new *Prune2* event, in the third box (which is a menu selector which defaults to ‘ordinary’) open the menu and select ‘convergent’. Go back to the *Synthesis* window. At the lilac star against *Prune2*, right click to open the menu, and click *Copy*. In a clear area of the *Events* pane right click to open the menu, and click *Paste*. Click the explore icon of the copy to expose its structure. Change its name to *Prune3* and change all references to 2 to 3. In a similar way create *Prune5* with references to 5, and *Prune7* with references to 7. For the copied events *Prune3*, *Prune5*, *Prune7*, you can check that the **CONVERGENT** property has been correctly copied, if you like. Save. The act of saving now takes quite while, and generates various errors which we discuss in a moment.

Note that we did *not* make these new events *Refine* any events in *MyMchA*. The *Prunen* events represent genuinely new functionality in *MyMchC*, relevant to aspects of the state in *MyMchC* which is not pertinent to the state of *MyMchA*. However there *is* a connection with *MyMchA*. It is the policy of Event-B that such new events introduced in a refining machine, must in effect, be refinements of (an unstated) ‘do nothing’ event of *MyMchA*. Moreover, such new events should never disable (the representatives in their system of) the pre-existing more abstract events, by being enabled indefinitely. The most effective way that the latter can be ensured is if there is an expression, say *V*, taking values in the natural numbers (or some other well founded set) such that every time one of the new events is executed, the value of *V* is reduced. (Therefore, since *V* is always a natural number, an infinite sequence of new event executions is impossible, since it would try to drive the value of *V* below zero.) One thing that is *not* prevented by this is an *increase* in the value of *V* when an *old* (i.e. pre-existing abstract) event is executed. So the value of *V* typically increases when an old event is executed, then falls a finite number of times as a finite sequence of new events is executed ... and so on, perhaps indefinitely. An expression *V* that displays these properties with respect to a machine like *MyMchC* is called a *variant* for the new events of *MyMchC*. The action of declaring a new event convergent, is what informs Rodin that it should verify that the execution of the event decreases the variant.

In the *Edit* window of *MyMchC*, the **EVENTS** section is highlighted in red/yellow, and if it is open, the ‘convergent’ attribute for the new *Prunen* events is also highlighted in yellow. If you are in the Event-B perspective you will have seen some problems in the *Problems* window, complaining that no variant has been specified for *MyMchC*. If you are in the *Prover* perspective you will see a lot of proof obligations with a yellow exclamation mark against them. In both perspectives the complaint is that there are new events, which are declared **CONVERGENT**, but that a variant cannot be found.

A variant for *MyMchC* is easy to invent. Consider as a candidate for *V* the expression ‘*y*’. It is obviously natural number valued. Also every time one of *Prune2*, *Prune3*, *Prune5*, *Prune7* is executed, its value decreases. So it will do as a variant. (Note that executing an *old* event, i.e. one of *AddEla*, *AddElb*, *AddElc*, *AddEld*, will multiply *y* by something, and so the value of *V* will increase, as predicted.)

Open the *New Variant Wizard*. Input ‘*y*’. Click *OK*. Save. More proof obligations appear, but the yellow exclamation marks against the previous ones have gone. You can have more fun playing at trying to prove as much or as little of as many or as few of the unproved proof obligations as you wish, in the manner described above.

## 11 Creating an Archive of the *SetProject* Project


It is often necessary to move projects around, send them to others etc. The best way to do this is by creating an archive. Create an archive of the *SetProject* development as follows. First ensure everything is saved. Then go to the top level *File* menu and select *Export*. A dialog box appears, containing a highlighted text line, and an explorer area with two entries: *General*, *Team*. Click the  $\oplus$  explore icon beside *General*, and click on *Archive File*. Observe how the *Next* button is now lit up. Click the *Next* button. In the new pane that appears click the project you wish to archive, and click the *Select All* button (to select all of the components of the project). Use the *Browse* file browser to input a name for the archive that is about to be created, noting that the *Finish* button lights up when you do so. Click *Finish*. Your archive is created.

## 12 Loading External Material into a Project

There are various ways of getting stuff created elsewhere into a project. Basically it is all done via the *Import* dialog selectable from the top level *File* menu. To import a project into your workspace, open the *Import* dialog from the top level *File* menu. There is a highlighted text line, and an explorer area with three entries: *General*, *Run/Debug*, *Team*. Click the  $\oplus$  explore icon beside *General*, and click on *Existing projects into Workspace*. Observe how the *Next* button is now lit up. Click the *Next* button. In the new pane that appears, choose the *Select archive file* option, and use the *Browse* button to locate the archive you wish to access, noting that the *Finish*

button lights up when you do so. Click *Finish*. The contents of the selected archive are imported into the workspace. Other possibilities are also offered by the *Import* dialog.

### 13 Printing the Results

At the moment, everything is held internally within the Rodin workspace, and is saved as XML text when you exit Rodin, not the most readable of formats. To get a printable version of a component do the following. First open the component in the main window and ensure that it is selected. Then click the  wizard at the right of the wizard icon list above the main window. Nothing much visible happens, but behind the scenes, the click causes the internal form of the component to be converted to L<sup>A</sup>T<sub>E</sub>X source, which can be found in the file *componentname\_<date>.tex* in the `latex` directory of the project you are working on. Observe that the `latex` directory also contains the style file `b2latex.sty`.

To process the file further, it is best to use `linux`. Make sure you are in the right directory; make sure (to be on the safe side) that there is a copy of the `b2latex.sty` file in the same directory; and then (assuming you've removed the date information from the filename), at the `linux` command line type:

```
pdflatex componentname.tex
```

This produces the file *componentname.pdf* (as well as some other intermediate files). The pdf can now be opened and/or printed using the usual tools.

Alternatively, the source file *componentname.tex* may be opened in a text editor, and its innards may be plundered to create, or to add to, other more elaborate L<sup>A</sup>T<sub>E</sub>X documents.

### 14 Capturing Rodin Screenshots

There is nothing much Rodin-specific about capturing screenshots of course. The following is included for completeness.

First get your Rodin session as you'd like to capture it. Ensure the Rodin window is selected. Capture the selected window to the Clipboard by pressing *Alt-PrintScreen* on the keyboard. Open the Windows Paint program under *Programs > Accessories > Paint*. Type *Ctrl-V*. Choose *Save As ...* and enter a suitable filename **ensuring the type is .PNG**. The `.PNG` graphics type is the the only graphics file type that will work with the supplied `Template.tex` document discussed elsewhere in CS60110. Furthermore, the `.PNG` **suffix must be converted** to `.png` before it will be recognised by L<sup>A</sup>T<sub>E</sub>X.