Bank Account

Manuel Carro
manuel.carro@upm.es

December 20, 2015

The objective of this project is to develop an Event-B model of a bank where clients can make several operations on accounts (open accounts, deposit money on an account, withdraw money from an account, and transfer funds between accounts). Some Event-B characteristics which we did not explore in the lectures and can be helpful for this project are presented in Section 5.

1 Requirements

FUN 1	The *bank* deals with *clients* and *accounts*.
The bank keeps the money belonging to its clients and lets them operate through their accounts.	

FUN 2	*Opening* an *account* permanently associates it with a single *client*, the *owner* of the *account*.
Certain operations on the account are restricted to its owner. There are no accounts whose ownership is shared among several clients. There is no limit to the number of accounts that a client can own.	

FUN 3	Every *account* has a *balance*.
The balance of the account represents, at every moment, the consolidated result of all the operations involving that account since it was opened.	

FUN 4	The *balance* of any newly *opened* *account* is zero.

FUN 5	The *balance* of an account cannot be negative at any moment.
The bank does not allow its clients to execute operations for which they do not have enough funds.	

FUN 6	Bank *clients* can *deposit* money in any existing *account*, thereby increasing its *balance*.
Owning an account is not necessary to deposit money on it.	

FUN 7	Money can be *withdrawn* from an existing *account*, and its *balance* is decreased by that amount.

FUN 8	Money can be *transfered* from between two existing *accounts*. The *balance* of the *account* from which money is transferred is *reduced* by the amount being transferred, and the *balance* of the *account* to which money is transferred is *increased* by the amount being transferred.

FUN 9	Only the *owner* of an *account* can *withdraw* or *transfer* money from it.

FUN 10	The decrease of the *balance* of any *account* caused by two consecutive *operations* has to be less than or equal to K (for some constant K).
Continuous transfers / withdrawals above a certain amount are not permitted.	

2 Tasks

Your task is to develop an Event-B model respecting the requirements presented in Section 1. Use invariants as strong as possible (but not stronger than necessary) to capture these requirements. You can choose to perform model refinement. In you choose to do refinement, there are several strategies you can follow: start with all the operations and less requirements, start with all the requirements and not all the operations, a combination of both, etc. You are free to choose any strategy.

All of the proof obligations that Rodin requires you to discharge must be proven. If you are not able to prove some of them using the theorem provers, mark it as reviewed (with the Ⓡ blue button), make an as rigurous as possible hand-made proof and submit it as part of the documents to be turned in.

If the requirements are insufficient (for example, if you think that some conditions are missing) you are free to suggest new requirements as long as they do not contradict other requirements or they do not severely limit the functionality of the system.

3 Teams, Submission, and Deadlines

The tasks described in Section 2 are to be done (and turned in) by **teams of two or (very preferably) three students**. Every team should send me (to `manuel.carro@upm.es`) an email with the team name and the names of the team members as soon as possible. Teams are required to send me some documents by email and to present the project.

Documents to be submitted

1. A Rodin Event-B project including the model(s) for the problem. The proofs mentioned in Section 2 must be discharged.

2. An explanation of how each of the requirements is addressed in the model(s).

3. If necessary, an additional document with the proofs, as convincing as possible, that you could not prove with Rodin.

4. The presentation slides in PDF format.

For point 2 above, either send me a separate document explaining how the model addresses the requirements, or include enough comments in the Rodin model(s) to cover this point. Please refer to *The RODIN Tool* subsection of `http://lml.ls.fi.upm.es/rsd/` for more information on the tool.

Presentation: Every team is required to make a presentation of 20 minutes (maximum). You should present the problem and your strategy to solve it. Every team member must explain part of the project. Your classmates will have the chance to make questions related to the presentation and its contents.

Deadlines:

- The deadline to turn in the reports / Rodin models is **Sunday, January 17th 2016, 11:59pm**. You can turn in the **PDF** for the presentation right before the presentation session.

- The presentations will take place on **Wednesday, January 20th 2016, from 7pm to 9pm**, at the usual classroom. Please come with a laptop. If you do not have any laptop available we can provide one, but we can only guarantee being able to show PDF presentations.

- If you want early feedback on the slides / project approach, please send them to me by **Thursday, January 14th, 2016, 9pm**. I cannot promise to be able to deliver feedback on time if I receive a request later than that moment.

I encourage you to have a look at some advices on how to prepare and make presentations. The link

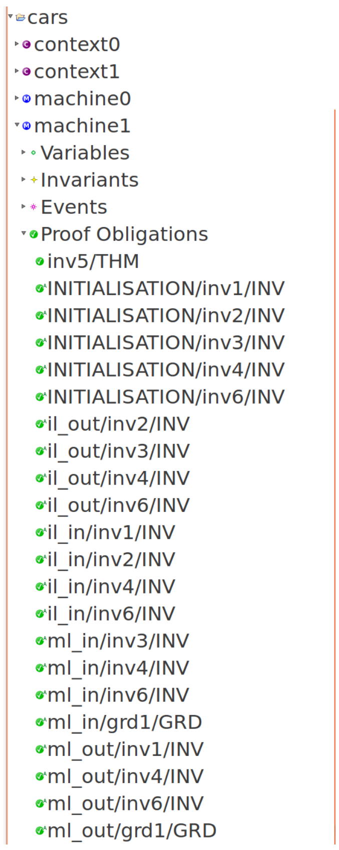
`http://www.cs.cmu.edu/~mihaib/presentation-rules.html`

contains some good recommendations, although I do **not** agree with the “no LaTeX” one: what you use to prepare presentations is quite independent of the presentation contents.

Feedback: Every student will be asked to rank all the presentations. We will use it to give an overall ranking using the Kemeny-Young votation method. *Note:* this is **not** to determine your grade, but rather to give you an idea of how much the audience liked your presentation. That may have little to do with how good is the contents or the resulting model.

4 Tips and Additional Information

Which Proofs Need to Be / Have Been Discharged? Rodin keeps track of the proofs that need to be done (both the standard proof obligations and those derived from user-stated `theorems`) and tries to prove them every time the project is saved. Their state can be seen by opening the *Event B Explorer* view (if not already open) and expanding the model, then the machine, then the *Proof Obligations* section. Discharged POs appear in green, POs still not discharged appear in brown (Figure 1).



```
cars
 context0
 context1
 machine0
 machine1
  Variables
  Invariants
  Events
  Proof Obligations
   inv5/THM
   INITIALISATION/inv1/INV
   INITIALISATION/inv2/INV
   INITIALISATION/inv3/INV
   INITIALISATION/inv4/INV
   INITIALISATION/inv6/INV
   il_out/inv2/INV
   il_out/inv3/INV
   il_out/inv4/INV
   il_out/inv6/INV
   il_in/inv1/INV
   il_in/inv2/INV
   il_in/inv4/INV
   il_in/inv6/INV
   ml_in/inv3/INV
   ml_in/inv4/INV
   ml_in/inv6/INV
   ml_in/grd1/GRD
   ml_out/inv1/INV
   ml_out/inv4/INV
   ml_out/inv6/INV
   ml_out/grd1/GRD
```

Figure 1: Discharged proofs.

Automatic Provers It is not difficult to write a model for the proposed problem for which Rodin automatically discharges all the proof obligations. If it does not happen for this case, manual help will be needed. You will need to have proving tools available for Rodin installed (e.g. *Atelier B*, which has to be installed separately).[1]

Crafting the Model The starting point is figuring out which variables are necessary and which invariants can be used to capture the requirements. Some requirements can be dealt with with invariants which are akin to type declarations (e.g., $a \in \mathbb{N}$). Others need logical formulas. Some requirements will however have to be dealt with in the guards of some events. Since Event B requires invariant preservation, the more requirements are expressed as invariants, the stronger the model will be. Last, some invariants may not capture directly requirements, but properties of the model whose preservation we want to ensure. They may also help Rodin to perform the proofs.

Then determine which events should be observed in the model and fill in obvious guards and the actions one event at a time. Invariant preservation and well definedness proof obligations are proven on a per-event basis, so you can in principle work event by

[1]See `http://handbook.event-b.org/current/html/tut_proving.html` for a tutorial and examples and `http://handbook.event-b.org/current/html/proving_perspective.html` for a description of the perspective. Some tips can be found at `http://handbook.event-b.org/current/html/use_provers_effectively.html`. Note that names of menu items may differ w.r.t. the version you have installed.

event trying to capture requirements. Rodin will retry simple proof strategies every time you save the project.

Seeing the Proofs When a proof cannot be discharged automatically it is anyway useful to see in which node the proof stopped — i.e., what had to be proven but could not be proved. As we saw several time during the lectures, this can give hints as to what is missing in the model. In order to do this, double click on the brown undischarged proof in the Event B explorer window. That will open new tab related to the proof. Then switch to the "Proving perspective" by selecting Window ⟶ Open Perspective ⟶ Proving. A "Proof Tree" window should appear stating the formula remaining to be proven; a "Proof Control" window should appear as well, where you can add intermediate goals to be proven, hypotheses in the search tree, and select additional strategies from the installed theorem provers (e.g. pp, p1, ml).

Invariants Invariants primarily capture requirements by means of establishing relationships among the variables which are used to model the system under study. However, and since models often contain variables which cater for fine-grained details not obvious at the level of the conceptual model, invariants can become complex and hard to prove automatically. In this case, automatic theorem provers can be helped by adding invariants that while implied by the model itself (and therefore redundant), are difficult to establish / deduce automatically. One can look at these invariants as "lemmas" which help provers accomplish their tasks.

Caveats

- Some versions of Rodin have an "innocent" bug: even if a proof is discharged, it is still shown brown in the Event B explorer, so one would think it has not been proved. Quit and restart Rodin to make it show the correct status.
- Although the different parts of the guards are in logical conjunction, sometimes changing the order of the guards help the theorem provers to do their job.
- This is also relevant to write "lemmas". Marking an axiom in a context as a "theorem" makes Rodin to try to prove it from the *previous* axioms. If it is proven, it becomes available to further proofs. It would therefore work as a lemma which helps prove additional properties.

5 Additional Event-B Characteristics

Some Event-B characteristics which were not necessary in the previous class examples / homework example can be of use for this term project (and also in general).

5.1 Event Parameters

Events may need to have private "parameters" whose value can be different for every event firing, and which are not shared with other events. For example, in the *Cars* example one may want to associate a plate to each car to assign spaces or to regulate parking time. These are simulated with event parameters. Parameters are made explicit in a clause named "**any**", as in the following event template:

```
Event  evt ≙
    any
        $a$
        $b$
    where
        grd1 : $a \in X$
        grd2 : $b \in Y$
        grd3 : $a \neq f(b)$
    then
        action : ...
    end
```

Parameters are used in the actions, but their values can not be updated — they are only "input", not "output". They commonly appear in guards, because their types or the properties / constraints they have to meet can only be stated there: since they are not part of the model state, they cannot appear in invariants. Section 6 presents a model where parameters are used.

5.2 Finiteness of Carrier Sets

Some proofs on sets are easier if they are performed on finite sets. Carrier sets can be stated to be finite by adding the finite(CS) axiom, where CS is a carrier set.

5.3 Functions

Functions have been informally introduced and used in the classroom examples and previous homework. A more rigorous description of what functions are and an introduction of and notation for function updates follow.

A **function** is a special class of binary relation.

A binary relation R on two sets S and T, declared $R \in S \leftrightarrow T$, is a set of *ordered pairs* $R = \{x \mapsto y \mid x \in S \land y \in T\}$. $S \leftrightarrow T$ denotes the set of all possible binary relations on S and T. $x \mapsto y$ denotes one ordered pair in a given relation. Several operations on binary relations (and therefore on functions) are available (see `http://lml.ls.fi.upm.es/rsd/Reference/EventB-Summary.pdf`). If R is a binary relation:

- $\mathrm{dom}(R)$, the domain of R, is the set of the first component of all the pairs in R: $\mathrm{dom}(R) = \{x \mid x \mapsto y \in R\}$.
- $\mathrm{ran}(R)$, the range of R, is the set of the second component of all the pairs in R: $\mathrm{ran}(R) = \{y \mid x \mapsto y \in R\}$.
- R^{-1}, the inverse of R, is defined as $R^{-1} = \{y \mapsto x \mid x \mapsto y \in R\}$.

There are also operations to perform restrictions and substractions on domains and ranges of binary relations in order to define new relations based on existing relations.

A partial function is a binary relation where each element in the domain has at most one range element associated with it: if $R \in S \leftrightarrow T$, and for any $x \mapsto y \in R$ and $x \mapsto z \in R$, it is always the case that $y = z$ (*uniqueness constraint*), then R is a function. Functions are so useful that there is a special notation for them. A partial function f between two sets S and T is denoted $f \in S \nrightarrow T$.

A function application, written $f(x)$ for $x \in \mathrm{dom}(f)$, denotes the (unique) element in $\mathrm{ran}(f)$ associated with x, i.e., $x \mapsto f(x) \in f$. If $x \notin \mathrm{dom}(f)$, then $f(x)$ is *undefined*.

The inverse f^{-1} of a function f is not always a function.

A function can be constructed by adding new ordered tuples to its definition:

$$f := f \cup \{x \mapsto y\} \text{ if } x, w \notin \mathrm{dom}(f)$$

This can be abbreviated as $f(x) := y$ when only one point in the domain (x

As a generalization, given two functions f and g such that $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \varnothing$, h defined as $h := f \cup g$ is also a function. For example,

$$f := f \cup \{x \mapsto y, w \mapsto z\} \text{ if } x, w \notin \mathrm{dom}(f)$$

Functions can be updated by replacing some ordered pairs with new ones:

$$f := f \mathbin{\lhd\!\!-} \{x \mapsto y\}$$

makes $f(x) = y$ be true regardless what $f(x)$ was before the update. The definition of $\lhd\!\!-$ for the case above is as follows:

$$f \mathbin{\lhd\!\!-} \{x \mapsto y\} \equiv \begin{cases} (f \setminus \{x \mapsto f(x)\}) \cup \{x \mapsto y\} & \text{if} x \in dom(f) \\ f \ \cup \{x \mapsto y\} & \text{if} x \notin dom(f) \end{cases}$$

This can be abbreviated as $f(x) := y$. Overriding can also be generalized to overriding one function with the contents of another function: $f \mathbin{\lhd\!\!-} g$ returns a function which contains all the tuples in g plus the tuples in f whose first component was not in $\mathrm{dom}(g)$. Then, for example,

$$f := f \mathbin{\lhd\!\!-} \{x \mapsto y, w \mapsto z\}$$

makes $f(x) = y$, $f(w) = z$ regardless of the previous values (if any) of $f(x)$ and $f(y)$.

A total function is a special kind of partial function. It is declared as $f \in S \rightarrow T$ and is defined for all the elements in S, i.e., $f \in S \nrightarrow T \wedge \mathrm{dom}(f) = S$.

Specific notations exist for functions which are injective, surjective, or bijective.

6 Example: Birthday Book

What follows is a simple but illustrative example of a birthday agenda which uses both event parameters and function updates. The slides at `http://deploy-eprints.ecs.soton.ac.uk/264/8/4_Functions.pdf` show how to construct this example stepwise and illustrate concepts related to functions.

```
CONTEXT  BirthdayBook
SETS                PERSON, DATE

MACHINE  BirthdayBook
SEES  BirthdayBook
VARIABLES
        birthday
INVARIANTS
        invBDay: birthday ∈ PERSON ⇸ DATE
EVENTS
Initialisation
      begin
            initBDay : birthday := ∅
      end
Event  addBDay ≙
      any
            p
            d
      where
            inPerson : p ∈ PERSON
            notRepeated : p ∉ dom(birthday)
            inDate : d ∈ DATE
      then
            newBDay : birthday := birthday ∪ {p ↦ d}
      end
Event  modifyEntry ≙
      any
            p
            d
      where
            existingPerson : p ∈ dom(birthday)
        then    newDate : $d \in DATE$
                chgBday : $birthday := birthday \lhd\!\!\!- \{p \mapsto d\}$
        end
END
```