Event-B Term Project: Robot Factory

Manuel Carro
manuel.carro@upm.es

December 15, 2019

The deadline to turn in the Event B models and reports is

Friday, January 10th 2020, 11:59pm

The project presentations will take place on:

Wednesday, January 15th 2020, from 6pm to 9pm

Every presentation should be 20 minutes max., questions included.

Contents

1 Goal

The objective of this project is to develop an Event-B model of a simplified automated factory where robot arms take items from a conveyor belt and place them in a box. When the box is (nearly) full, it is replaced by an empty one.

2 Requirements

REQ 1 A number of *robot arms* pick items from a *conveyor belt* and place them in a box.

REQ 2 All robot arms pick items from the same point, one arm at a time.

REQ 3 Every robot arm can hold at most one item at once.

REQ 4 The conveyor belt continuously provides items.

I.e., the system to be modeled is supposed to run indefinitely.

REQ 5 There is an unlimited supply of empty boxes.

REQ 6 Every item on the conveyor can have a different weight.

We cannot know the weight of the item in the belt before it appears in the pickup point.

REQ 7 There is a sensor which detects the weight of the item in the conveyor belt. A zero reading means that there is no item.

REQ 8 There is a sensor which detects the weight of the items in the box. A zero reading means that the box is empty.

REQ 9 The weight of any item is always are under a fixed maximum.

REQ 10 Robot arms can pick any item which appears in the conveyor belt, regardless of its weight.

REQ 11 There is a fixed *maximum weight* that the box can hold.

REQ 12 The weight of any item in the conveyor belt is smaller than the maximum weight that the box can hold.

Any item which appears in the conveyor belt can, at least, be dropped in an empty box.

REQ 13 Robot arms ought not drop items in the box if this would make the weight of the items in the box exceed the allowed maximum weight.

REQ 14 When the maximum weight that the box can store is going to be exceeded by an additional drop, the box must be replaced by an empty one.

We need to ensure that we do not store items in the box whose combined weight exceeds the maximum the box can hold.

REQ 15	A box is to be replaced by an empty one only when **no** robot arm can place any item in it.
The box must not be replaced if any arm holds an item whose weight can still be placed in the box. However, planning item dropping to maximize the box load is not necessary: if the box can hold 10Kg. and there are three arms with items weighting 7Kg., 4Kg. and 4Kg., dropping the 7Kg. item instead of the two 4Kg. pieces is admissible, but dropping only the 4Kg. piece and then replacing the box is not.	

REQ 16	The system must not deadlock.

REQ 17	All arms operate independently (e.g., they do not synchronize with each other) in order to get items from the conveyor as fast as possible.

3 Tasks

Your task is to develop an Event-B model to control the arms, belt, and box respecting the requirements presented in Section 2. Use invariants to capture these requirements when possible. You can decide whether to perform model refinement or not. In you choose to do so, there are several strategies you can follow: start with all the events and less requirements, start with all the requirements and not all the operations, a combination of both, etc.

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

If you think the requirements are insufficient (for example, if you believe 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.

4 Teams and Submission

The project is to be done (and turned in) by **teams of three people**. We may accept smaller teams in exceptional cases (please consult with me). We **disrecommend** working on the project alone. 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.

The documents to be submitted are:

1. A Rodin Event-B project including the model(s) for the problem. The proofs mentioned in Section 3 must be discharged. Follow the same instructions as in the Event B homework #2 and the classroom slides to export the project.

2. A written explanation of how the requirements are addressed in the model(s). This can be a set of slides that you can use to make the presentation of the project.

3. If necessary, an additional document showing how the proof obligations that you could not discharge with Rodin are met by your model.

Please refer to *The RODIN Tool* subsection of `http://babel.ls.fi.upm.es/teaching/rsd/` for more information on Rodin.

5 Additional Information

Which Proofs Need to Be / Have Been Discharged? Rodin keeps track of the proofs that need to be completed (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 not yet discharged appear in brown.

Writing Math Symbols in RODIN There is a guide on writing mathematical symbols in RODIN using the regular keyboard at `http://wiki.event-b.org/index.php/Rodin_Keyboard_User_Guide`. Besides, there is a "Symbols" window/tab in RODIN that can be used to input math symbols, but using the keyboard is way faster.

Automatic Provers You will need to have installed proving tools available for Rodin (e.g. *Atelier B*, which has to be installed separately). It is not difficult to write a model for the proposed problem for which Rodin automatically discharges all the proof obligations (maybe clicking some buttons in the *Proving View*, as mentioned in the the Event B Homework #2). If it does not happen for your case, I recommend to revise the specification, which goals could not be proven, and perhaps add invariants which may be used as intermediate steps for proving more complex invariants. Interactive proving can be used, but (depending on the model) it is not necessary for this project.[1]

Sensors You can assume a simple model for the sensors and actuators. Sensors can be read by consulting the value of a variable. Modelling an action on an external item can be done by changing the value which represents the sensor's reading. The emphasis is on the modeling of the system, which has to be realistic, but not so much that the interaction with the external world dominates the model.

Crafting the Model Your first step should be thinking on the events that describe the system. The simpler, the better: we are modeling how a system works, not directly programming. However, the result is similar to a program, of course, as it has a direct operational semantics as a program.

Then, decide which variables are necessary, including how these external elements (the sensors, in our case) are represented. Then determine which events should be observed in the model and fill in obvious guards and the actions one event at a time.

Now, use these variables to express invariants which capture as many requirements as possible. Some requirements can be dealt with using type invariants (e.g., $a \in \mathbb{N}$). Others need more complex logical formulæ. Some requirements will have to be dealt with in the guards of some events. Since Event B requires invariant preservation, the more requirements expressed as invariants, the stronger the model will be. Also, some invariants may not directly capture requirements, but properties of the model whose preservation we want to ensure. They may also help Rodin to perform the proofs.

Invariant preservation and well definedness proof obligations are proven on a per-event basis, so you can in principle work event by event trying to capture requirements. Note that expressing deadlock freedom uses the guards of all events, so any change in them potentially impacts the expression of deadlock freedom.

Remember that RODIN will retry simple proof strategies every time you save the project.

[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.

Working with the Proofs When a proof cannot be discharged automatically it is useful to know where the proof stopped — i.e., what (sub)goal could not be proved. This can give hints as to what is missing or wrong in the model. A double click on the brown undischarged proof in the Event B explorer window will open a new tab related to the proof. Switch to the "Proving perspective" by selecting Window $\longrightarrow$ Open Perspective $\longrightarrow$ Proving (a shortcut icon is also available). A "Proof Tree" window should appear stating the formula remaining to be proven, which also appears in the "Goal" window. 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). A "Proof Information" tab on the right can show details on the proof being performed, including the event and the environment of the proof.

Invariants Invariants capture requirements by establishing relationships among model variables. 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 are implied by the model itself (and therefore redundant) but which are difficult to establish / deduce automatically. One can look at these invariants as "lemmas" which help provers accomplish their tasks. Adding some of these may be helpful.

Caveats

- After a proof is discharged, *save* it (using `Ctrl-s`) so that it appears green in the Event B explorer window. Note: some early versions of Rodin have an "innocent" bug: one had to quit and restart Rodin to make it show the correct status. This should not be necessary in more modern versions, but you may check if it helps.
- Although the different parts of the guards are in logical conjunction, sometimes changing the order of the guards helps the theorem provers to do their job (this is actually necessary in some cases — see the next bullet).
- The order of formulas is relevant to write "lemmas". Marking a formula 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. The same happens if a formula in the invariant section is marked as a theorem: the theorem provers will try to prove it based on the *previous* invariants.

Seeing the Whole Model A large model can sometimes be difficult to work with — the display may be cluttered with large formulas. Rodin can export models to LaTeX which can later be displayed / printed. That is performed by a plugin — see `http://wiki.event-b.org/index.php/B2Latex` for an explanation of how to install the plugin and use the generated LaTeX file.

Deadlock Freedom To prove deadlock freedom, the disjunction of the guards has to be always true — i.e., there is always some event that can be executed. This disjunction can be written as an invariant or, alternatively, as a `theorem` that can be proven based on the previous invariants. Both are logically equivalent. The latter is preferred because its proof does not depend on changes in the actions / guards of the events, but it may need additional work. Try both.

To use the `theorem` approach:

- In the **INVARIANT** section, add a formula with the disjunction of the guards. Make sure it is the **last** in the list of the invariants.

- Mark it as a theorem: after the formula there should be a label reading `not theorem`. Click on it and it will change to mark it as a `theorem`.

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.

Parameters are made explicit in a clause named "**any**", as in the following event template:

```
MACHINE Example Any
EVENTS
Event sample ⟨ordinary⟩ ≙
    any
        a
    where
        grd1: a ∈ X
        grd3: a > f(a)
    then
        act1: ...
    end
END
```

Parameters can not be updated in the actions — they are only "input". Their scope is restricted to the event and they are not part of the model. Therefore, their type and properties cannot be stated in the invariants and have to appear in the guards. Section 7 presents a model where parameters are used.

6 Additional Event-B Characteristics

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

6.1 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.

6.2 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 to function-related notation 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://babel.ls.fi.upm.es/teaching/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: $\mathrm{ran}(R) = \{y \mid x \mapsto y \in R\}$.

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 element in the range 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 $f \in S \nrightarrow T$ and it may be undefined in some elements of S.

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., $f \in S \nrightarrow T \Rightarrow \mathrm{dom}(f) = S$.

A function application written $f(x)$, 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*.

Updating a function A function can be updated by adding new tuples to its definition:

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

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

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

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 $f(x) := y$. More than one point can be updated at the same time:

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

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

Initializing a function Sometimes it is necessary to set a total function to some (default) value in all the points of its domain. If $f \in S \rightarrow T$, then the expression

$$f := S \times \{a\}$$

where $a \in T$ makes $f(x) = a$ for all $x \in S$. The expression $A \times B$ is the cartesian product of all the elements in A and all the elements in B. Therefore, $S \times \{a\}$ is the set of pairs $\{s_0 \mapsto a, s_1 \mapsto a, \ldots\}$ for all the elements $s_i \in S$.

7 An Example of Functions: 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 The set of all persons. We do not know what the members or properties of this sets are.
    DATE The set of all dates. We know nothing about dates. For example, we do not have a notion of a
        date being before or after some other date.
END
```

```
MACHINE BirthdayBook
SEES BirthdayBook
VARIABLES
    birthday
INVARIANTS
    invBDay: $birthday \in PERSON \pfun DATE$ Partial function: we do not store the birthday of every
        person
EVENTS
Initialisation
    begin
        initBDay: $birthday := \varnothing$ We start with an empty agenda
    end
Event addBDay ⟨ordinary⟩ $\widehat{=}$
    any
        p
        d
    where
        inPerson: $p \in PERSON$
        inDate: $d \in DATE$
        notRepeated: $p \notin \mathrm{dom}(birthday)$
    then
        newBDay: $birthday := birthday \cup \{p \mapsto d\}$ Or, simply, $birthday(p) := d$
    end
Event modifyEntry ⟨ordinary⟩ $\widehat{=}$
    any
        p
        d
    where
        existingPerson: $p \in \mathrm{dom}(birthday)$ Therefore $p$ is a PERSON whose birthday we have already
            registered
        newDate: $d \in DATE$
    then
        chgBday: $birthday := birthday \lhd\!\!- \{p \mapsto d\}$ Or $birthday(p) := d$
    end
END
```