Exercise Session 8 : Refinement and Proof Obligations in Event-B

ingmar.dasseville@cs.kuleuven.be
laurent.janssens@cs.kuleuven.be

November 24, 2015

1 Event-B

1.1 Introduction

The following two exercise sessions will give a brief introduction into the Event-B language and the use of refinement and proof obligations. Event-B is an evolution of the B-method and uses most of the same concepts and techniques as discussed in the sessions on ProB. The language focuses purely on the modelling of dynamic systems, such as the examples we covered in ProB. Event-B has a slightly different notation compared to B, to more easily allow the modelling of these systems.

The two main differences :

- B-machines are split into two parts : a static part, called a context, and a dynamic part, the machine.
- Operations are called Events in Event-B.

1.2 Rodin

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. It offers a user friendly way of modelling.

1.3 Rodin Handbook and Tutorial

A handbook on Rodin and Event-B is available at `http://handbook.event-b.org/current/pdf/rodin-doc.pdf`. This handbook contains all the infor-

mation you need concerning usage of Rodin, notation, refinement and proof obligations to complete these sessions and the project.

During this exercise session go through the tutorial listed in the book (chapter 2).

- Start at section 2.4 : Follow along with the TrafficLight example and implement it in Rodin.
- Skim through section 2.5 : It details the mathematical notation used in Event-B.
- Read section 2.6 : Make sure you know how to create a context and populate it. There's no need to implement the example.
- Carefully read section 2.7.
- Follow along with the example in 2.8, implement the refinement of the TrafficLight example, as in the handbook.
- Follow along with the example in 2.9 : This'll show you how to use proof obligations to detect flaws in your models.

Completing this tutorial should give you all the information you need for the next exercise session as well as an introduction into the use of refinement for modelling.

If you have time, complete the following introductory example.

1.4 A Beverage Vending Machine

Import the BeverageVendingMachine archive, the same way you would import a .zip in eclipse. Simply create a new empty Event-B project and import all the files from the archive.

The project contains two files: the Beverages context and the VendingMachine0 machine. These are listed below (you can not copy-paste this code into Rodin). Rodin uses a tree-structure to maintain the source code. New elements can be added by right-clicking the desired branch and choosing the type of node to add. Information can be added to these nodes by clicking the desired location.

Context

```
context Beverages

sets drinks

constants cola fanta no_drink

axioms
  @drinks_partition partition(drinks , {cola}, {fanta}, {
     no_drink})
end
```

This context specifies which drinks are available in the vending machine. the set `drinks` consists three constants `cola`, `fanta` and `no_drink`. This is specified in the Axiom `drinks_partition`. The partition command has a strict form, as shown, ie. partition(set-name, element1, element2, ...).

Axioms are used to define the type of the constants as well as provide additional information about the constants and sets.

Remark : Sets can only consist of user-defined constants, for sets of predefined elements you should use constants. For example to specify the set of even numbers between 1 and 10, you create the constant `even` and constrain it via an axiom, as in the following example.

```
context Even_example

constants even

axioms
  @even_def even = {2,4,6,8}
end
```

Machine

```
machine VendingMachine0 sees Beverages

variables drink // The drink which is selected
          paid // Whether you've paid or not


invariants
  @drink_type drink : drinks
  @paid_type paid : BOOL
```

```
  @drinks_are_paid drink /= no_drink => paid = TRUE

events
  event INITIALISATION
    then
      @init_drink drink := no_drink
      @init_paid paid := FALSE
  end

  event insert_coin
    where
      @no_paid paid = FALSE
    then
      @paid paid := TRUE
  end

  event select_cola
    where
      @paid paid = TRUE
      @no_drink_yet drink = no_drink
    then
      @select drink := cola
  end

  event select_fanta
    where
      @paid paid = TRUE
      @no_drink_yet drink = no_drink
    then
      @select drink := fanta
  end

  event get_cola
    where
      @paid paid = TRUE
      @cola_selected drink = cola
    then
      @get_drink drink := no_drink
      @end_payment paid := FALSE
  end

  event get_fanta
    where
```

```
      @paid paid = TRUE
      @fanta_selected drink = fanta
    then
      @get_drink drink := no_drink
      @end_payment paid := FALSE
  end
end
```

This example offers a very simple vending machine, which stocks cola and fanta. After inserting a coin (event insert_coin) it's possible to select a drink, either cola (select_cola) or fanta (select_fanta). After which said drink is provided and the payment completed.

Questions

- Refine the vending machine. Start by right-clicking the VendingMachine0 machine and selecting refine. Notices all events are copied and designated *extended*, this means all information about the event is kept and you only add to it. In other words, there's no data refinement. For this exercise, leave all events on extended.

- Vending machines tend to have a limited stock. Refine the machine so that it has a certain *quantity* of cola and fanta and refine the appropriate events so that these quantities are reduced when you get a certain drink.

- Does the machine you just created constitute a correct refinement of the original vending machine? You can check by right-clicking the machine and selecting "Start Animation / Model Checking". This works just as the ProB Animator. If you ignore the new variables, the machines should have exactly the same behaviour. Do they? Why is that?

- Add a refill event to solve the problem. Make sure this event isn't enabled continuously by adding an appropriate guard. Otherwise it would all the machine to go into a loop and never return to one of the original events.