Exercise Session 6 : ProB

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

November 29, 2016

1 ProB

1.1 Introduction

The following exercise sessions are an introduction to the B-method, using ProB. We describe the input language of the B-method by presenting some examples in the ProB animator. The input language of the B-method is designed for the formal specification of software systems, using a notation describing abstract machines. These machines are described using a combination constants, variables, invariants and operations or events. Hence these machines are transition systems, where the operations or events represent the actions of the system.

1.2 ProB

ProB[1] is an animator and model checker for the B-method. Correctly specified machines can be simulated and tested by animating the machine.

Essentially it allows to "run" the machine and see the effects of the different events and possible event sequences. Additionally you can ask the animator to search for a sequence of events which satisfy a certain goal. This makes it possible to easily check whether undesired states are reachable by the machine.

1.3 Disambiguation

There are a number of languages derived from the B-method and several tools exist for working with these languages. However the names of these tools and

[1] http://stups.hhu.de/ProB/w/The_ProB_Animator_and_Model_Checker

language are often used to mean the same thing. To avoid confusion below is a list of different languages and tools :

Model checker A model checker is a software tool which allows to check invariants and the correctness of the modelling of a system. This does not coincide with the model checking inference in IDP. For the exercise sessions on ProB, model checking will be used to denote the checking of correctness of a modelling.

B The B language is specification language based on a mathematical notation.

B-method The B-method is a formal software modelling approach, which includes the use of the B language, or one of its derivatives, and the associated tools.

Event-B The Event-B language is an evolution of B, which describes systems using events and incorporates refinement and proof obligations.

ProB Machines created using B or Event-B can be animated and checked using ProB. This is the tool we will be using to perform simulate the Transition Systems we model.

Rodin Rodin is an extension of Eclipse specifically designed to perform modelling using Event-B. You will use this, when doing modelling exercises and for the project.

2 Example

Consider the following simple program in the Event-B language :

```
MACHINE SingleProcess
SETS
  states={busy,ready}
ABSTRACT_VARIABLES
  request,
  state
INVARIANT
     state : states
   & request : BOOL
INITIALISATION
    state := ready;
    request :: {FALSE,TRUE}
```

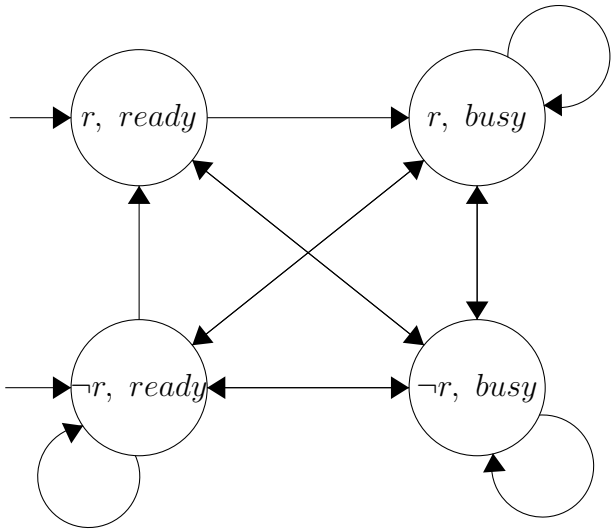
```
OPERATIONS
  accept_request(next_request) =
    PRE
    state = ready
       & request = TRUE
    THEN
          state := busy;
          request := next_request
    END;

  do_not_accept_request(next_request , next_state) =
    PRE
    state = busy or request = FALSE
    THEN
          state := next_state;
          request := next_request
    END
END
```

The possible states of the system are declared as a set, containing the values `busy` or `ready`. The space for the states of the Transition Machine (TS) is determined by the declarations of the variables (`request` and `state`). The variable request is set to be of type BOOL, meaning it can assume the (boolean) values TRUE and FALSE. The variable `state` can take the values `busy` and `ready`. The actions of the TS coincide with the two events `accept_request` and `do_not_accept_request`, the `INITIALISATION` event is used to give the system its initial state and occurs exactly once at the start of an animation.

Events are divided into two parts, indicated by the `PRE` and `THEN` sections. These sections include the guard and actions of the event, respectively. The guard, formula in the `PRE` section, specify *when* an event is enabled, ie. when it can take place. The actions specify what the effect of the event is, ie. the changes in the state of the system.

The event `accept_request` can occur only when the system is ready and a request is available, as specified by the guard. If it occurs the system's state is set to busy. Additionally the system may or may not receive another request, ie. the value of request is set either TRUE or FALSE, non-deterministically.

Similarly the event `do_not_accept_request` is enabled when the system is busy or when there is no request available. As a result of this event the state of the system is set to either busy or ready and with or without a request available.



Remark Note that the state variables of the machine correspond to fluents in LTC. Similarly the operations correspond to actions. Actually a B-machine can be translated to LTC, as will be demonstrated in Section 6. Keep this in mind while going through the exercises, i.e. for each concept in ProB try to figure out what concept in LTC, or IDP, it corresponds to.

Questions

- Consider the state graph of the above TS. It shows all possible states and transitions of the `SingleProcess` machine. The transitions are not annotated, but should be clear when the graph is compared with the B-machine. It is always possible to get in a state where there is no request and the state is `ready`. How would you prove such a statement by looking at the state graph?
  Solution: The state is reachable from each state.

3 Visualization

ProB can visualize the possible states of a machine. Under "Preferences" in the "Graphical Viewer" menu select PostScript. This sets the visualizer so that it prints PostScript files. Now you can select "State Space" from the "Visualize" menu to get a state machine representation of the modeled machine.

Doing this for `SingleProcess.mch` results in a representation of a state machine showing which transitions are possible in the system.

4 Animation

4.1 Single Process

Open the `SingleProcess.mch` file in ProB. The ProB window is divided into several sections.

- The upper half of the ProB window contains the source code of the model.
- The lower half contains the animation parts and consists of three panes:
  - The "State Properties" pane contains the information about the current state of the model. It shows the current assignment of values to the variables.
  - The pane called "Enabled Operations" contains a list of operations that can be applied to the system. Before the system can perform any operations, it has to be initialised. This happens by executing the assigments described in the INITIALISATION section in the source code. Since the initialisation is non-deterministic, there are two options for initialising the system. Thus two options are available : INITIALISATION(FALSE, ready) and INITIALISATION(TRUE, ready).
  - The History pane contains the list of operations which have been executed up to this point.

  Now, double click "INITIALISATION(TRUE, ready)" in the "Enabled Operations" pane. This initialises the system, setting the state to `ready` and request to TRUE. Now "accept_request" becomes available for selection, twice. This indicates the operation has different possible parameter values to choose from. Select either and watch how this influences the state of the system. Go through several selections of operations, look back to the state diagram above and see if you can follow every transition that is shown.

  The animation can be reset by right clicking in the "Enabled operations" pane and selecting reset.

  From the "Animate" list it is possible to select "Random Animation(10)" and "Random Animation...", these execute a selected number of operations at random.

4.2 Traffic Lights

Now open the `RealTrafficLights.mch` example in ProB. Open the "Preferences" and under "Configuration" select "MININT..MAXINT (32-bit)" and reopen the machine by selecting "Reopen RealTrafficLights.mch" from the "File" menu. Whenever you're working with integer values, make sure this range is set, other wise only numbers between -1 and 3 are considered integers. We will use this example to illustrate how the animator can be used to detect undesired qualities of a modelled system. The example models traffic lights found on a intersection. There are a set of lights for the north-south directions and a set for the west-east direction. These lights become green in an alternating fashion (forced by the ns_turn flag). Cars can only enter the intersection in a certain direction if the corresponding light is green and if the intersection isn't full (max 5 cars).

The B-machine (as found in the `RealTrafficLights.mch` file) is as follows :

In addition to describing how the traffic light system works, this machine also specifies several *desired* invariants. Invariants in ProB are not part of the model. They are properties which the modeled machine are *required* to have. The operations of the B-machine must maintain these invariants.

- Cars on the crossing are either travelling along the north-south direction or the west-east direction. In other words, we don't want any collisions.
- Both sets of lights can't be green at the same time.
- If one set of lights isn't red, the other one must be red.

Play around with the animator to get a feel for the example. Does the system behave as in real life? Does it behave as you would expect?

4.2.1 Finding a path to a state

Rather than going through an animation manually or having it run randomly, it is possible to ask ProB to find a sequence of operations which lead to a state satisfying a given predicate.

Open the "Animate" menu and select "Find State Satisfying Predicate...". We're going to investigate whether traffic lights actually prevent the possibility of collisions on an intersection.

In the window that opens enter : "on_crossing_ns $>$ 0 & on_crossing_we $>$ 0", ie. ask ProB to find a sequence of operations which lead to a collision on the intersection.

The resulting state violates the first invariant specified in the previous subsection.

5 Model Checking

The main purpose of a model checker is to verify that a model satisfies a set of desired properties specified by the user. The ProB model checker can find deadlocks and violations of the invariants.

Additionally temporal specifications can be expressed in two different temporal logics : Computational Tree Logic CTL, and Linear Temporal Logic LTL. The CTL and LTL specifications are checked in order and examples are provided which satisfy or disprove the formulas.

In this section we will describe model checking of specifications in CTL, while in the next section we consider the case of LTL specifications.

5.1 The ProB model checker

In the "Verify" menu select "Model check..." This opens the model checking menu. Make sure "Find Deadlocks" and "Find Invariant Violations" are checked. When you press "Model Check" ProB will look for deadlocks and states violating the invariants. In this case it will find a state where cars are travelling in both directions across the intersection and it will provide the path leading up to that state in the "History" pane.

Remark : To reset the machine after you changed the code, make sure to save and reopen the machine! This option is available in the "File" menu.

Questions

- Can you come up with a simple solution so that collisions are impossible.[2]

  Solution: Add a guard to the ns_green and we_green operations stating the intersection should be empty : "on_crossing_we = 0" for ns_green, and equivalent for ns_green

5.2 Computation Tree Logic

CTL is a *branching-time* logic: its formulas allow for specifying properties that take into account the non-deterministic, branching evolution of a transition system or finite state machine. More precisely, the evolution of a TS

[2]Tip : Could a camera with a view of the intersection help?

from a given state can be described as an infinite tree, where the nodes are the states of the TS and the branching is due to the non-determinism in the transition relation. The paths in the tree that start in a given state are the possible alternative evolutions of the TS from that state. In CTL one can express properties that should hold for *all the paths* that start in a state, as well as for properties that should hold just for *some of the paths*.

Consider for instance CTL formula `AF p`. This formula is true in a state if for all paths (`A`) starting from that state, eventually (`F`) proposition `p` must hold. That is, all the possible evolutions of the system will eventually reach a state satisfying proposition `p`. CTL formula `EF p`, on the other hand, requires than there exists some path (`E`) that eventually in the future satisfies `p`.

Similarly, formula `AG p` is TRUE in a state S if proposition `p` is always, or *globally*, true in all the states reachable from state S. Formula `EG p` is TRUE if there is some path starting from the current state along which proposition `p` is continuously true.

Other CTL operators are:

- `A [p U q]` and `E [p U q]`, requiring proposition `p` to be true *until* a state is reached that satisfies proposition q;
- `AX p` and `EX p`, requiring that proposition p is true in all or in some of the next states reachable from the current state.

CTL operators can be combined using logic operators (`not,&,|,=>,...`) and they can be nested in an arbitrary way. Typical examples of CTL formulas are `AG not p` ("proposition `p` is absent in all the evolutions"), `AG EF p` ("it is always possible to reach a state where `p` hold"), and `AG (p -> AF q)` ("each occurrence of proposition `p` is followed by an occurrence of proposition `q`").

In ProB a CTL specification can be entered by choosing "Check LTL/CTL Assertions" from the "Verify" menu. Switch to the "Add CTL Formula". Whenever a CTL specification is processed, ProB checks whether the CTL formula is true in all the initial states of the model. If not, then ProB will attempt to generate a counter-example: a (finite or infinite) sequence of operations that exhibits a valid behavior of the model, which does not satisfy the CTL specification. These sequences are very useful for identifying the error in the specification that leads to the wrong behavior. We remark that the generation of a counter-example is not always possible for CTL specifications. Temporal operators corresponding to existential path quantifiers cannot be proved false by a showing of a single execution path. Similarly, sub-formulas preceded by universal path quantifier cannot be proved true by a showing of a single execution path.

In ProB predicates p, pertaining to information of the machine, must be entered in between curly brackets, eg. EF {ns_light = green}, meaning there is a path so that eventually the north-south lights will be green.

Questions

- In the last subsection we fixed the traffic lights. The system is now safe, but may be too restrictive. Check that cars can enter the crossing from both directions.
  Solution: AG (EF {on_crossing_ns > 0} & EF {on_crossing_ns > 0})

5.3 Linear Temporal Logic

ProB also allows for specifications expressed in LTL. Intuitively, while CTL specifications express properties over the computation tree of the TS (branching-time approach), LTL characterizes each linear path induced by the TS (linear-time approach), starting from a chosen state. In other words, while CTL essentially expresses properties of states, LTL generally specifies properties of a path. LTL formulas are true in a state when the LTL formula is true in every path leading from that state.

The two logics have in general different expressive power, but also share a significant intersection that includes most of the common properties used in practice. Some LTL operators are:

- `F p` (read "in the future `p`"), stating that a certain proposition `p` holds in one of the future time instants;
- `G p` (read "globally `p`"), stating that a certain proposition `p` holds in all future time instants;
- `p U q` (read "`p` until `q`"), stating that proposition `p` holds until a state is reached where proposition `q` holds;
- `X p` (read "next `p`"), stating that proposition `p` is true in the next state.

We remark that, differently from CTL, LTL temporal operators do not have path quantifiers. In fact, LTL formulas are evaluated on linear paths, and a formula is considered true in a given state if it is true for all the paths starting in that state.

Equivalently to CTL formulas in ProB, predicates need to be entered in curly brackets.

Questions

- Starting from the initial state, check that the two lights (`ns_light` and `we_light`) are never green at the same time.
  Solution: G $\neg\{nslight = green \& welight = green\}$

- Add a specification to check that there are no collisions, starting from the initial state.
  Solution: G {on_crossing_ns = 0 or on_crossing_we = 0}

- Can you make this system collision free if lights could only be green or red? Find out by changing the events so the lights never turn orange and use LTL, CTL or the animator to answer these questions.
  Solution: You can, in exactly the same way.

---

6 B to FO(·)

It is important to realize that the knowledge which can be represented in B is not of a different kind than that which is representable in FO(·) using LTC. We will show this by translating the `SingleProcess` example to FO(·) using a small set of general translation rules. These rules are divided based on whether they apply to the vocabulary, the theory or the structure.

Vocabulary rules :

- Introduce the type `Time`, constant `Start : Time` and partial function Next(Time) : Time, needed to support LTC theories.

- Create a constructed type for each defined set, constructed from the values in the set.

- Each variable in B corresponds to a fluent in LTC. Hence, create a predicate or function for each such variable, these should have at least a parameter of type Time. Also create a constant `I_VariableName`, which represents the initial value of the Variable, and the causal predicates C_VariableName and C_nVariableName.

- Translate each operation/event into an action predicate, taking into account any parameters.

Theory rules :

- Make a definition for each translated variable V:

- Set the initial value for V:
  `V(Start) = I_V`.
- Define the rest of the fluent by using the causal predicates.
- Do not forget to add the inertia rule.

- Make a definition for each pair of causal predicates.
  - For each event that changes the associated variable, add a rule which defines the causal predicate appropriately.
  - Remember to also define the C_nVariableName predicate appropriately.

  This way all the actions of the operations and the variable have been translated.

- All that is left to translate are the guards. For each event E, translated by an action predicate with appropriate parameters, with guard G add the following sentence to the theory :
  $\forall$ `t` $\bar{x}$ [`Time`] : `E(t,` $\bar{x}$`)` $\Rightarrow$ `G.`
  Where all components of G are replaced by their translation. Quantify over variables where needed.

- To make sure the LTC theory results in the same functionality as the B-machine, only one action can happen at any time. Add a constraint which specifies only one action can happen at any time t. One way of doing this is to use a standard concurrency axiom:

$$\forall t : \#\{\bar{x} : Act1(t, \bar{x})\} + ... + \#\{x : Actn(t, \bar{x})\} < 2.$$

Structure rules :

- Add the interpretation of Time, as needed for LTC.
- Add the initialisation of the variables by giving a value to the `I_V` constants.

Performing model expansion on this theory is equivalent to performing a random animation on the B-machine. To specify the animation steps you can constrain the `Occur` function in the structure.

Questions

- Translate the `SingleProcess` machine to FO($\cdot$).
- Verify that model expansion *on this theory* is indeed equivalent to animation in ProB.

7 EXTRA: Jugs

You must obtain exactly 4 liters of water using a 5 liter jug, a 3 liter jug and a water pump. Open the `Jugs.mch` file in ProB.

Questions

- Use the model checker to find a solution. You can do this by enabling the "FIND DEFINED GOAL" option in the "Model Check" window.
- Verify that it is always possible to reach the goal, regardless of previous decisions.
  Solution: AG EF {contents(jug_5) = 4}
- Show that if you have an arbitrarily large jug you do not need the `empty` operation to reach a solution (4 liters in the 5 liter jug or in the arbitrarily large jug).
  Solution: Remove empty operation and add a jug.

---

8 Nim

Nim is a game for two players. In the initial situation, a set of matches are on the table in the form of a set of heaps of increasing length, respectively 1, 3, 5, 7, etc. (see Figure 1). When modeling the game, you may assume that there are only four heaps. In turns, the players take a number of matches away from a heap: at least one match and at most as much as there are in the heap. The player that takes the last match loses the game.

Figure 1: The beginsituation of the game Nim

Starting from the following skeleton, model the Nim game. All essential constants, variables, invariants as well as events are provided. You only have to complete the events.

Skeleton

```
MACHINE Nim
SETS
   winners={player1 ,player2 , no}
CONSTANTS
   players ,
   next_player ,
   heaps
VARIABLES
   turn_counter ,
   current_player ,
   heap_count ,
   winner
PROPERTIES
      next_player = {player1 |-> player2 ,player2 |-> player1}
  & players = winners - {no}
  & heaps = {1,2,3,4}
INVARIANT
      winner : winners
  & turn_counter : NAT
  & current_player : players
  & !(x).(x : heaps => heap_count(x) >= 0)
  & (heap_count(4) > 0 => winner = no)
  & ( winner /= no => heap_count(4) = 0)
INITIALISATION
          heap_count := {(1|->1),(2|->3),(3|->5),(4|->7)};
          winner := no;
          current_player := player1;
      turn_counter := 0
OPERATIONS
   turn(nb, heap) =
     PRE
    nb : {0}
        & heap : {0}

     THEN
    skip


     END;

   game_over =
```

```
    PRE
   TRUE = TRUE

    THEN
   skip


    END
END
```

Tips

- Updating a function in ProB happens through the <+ operator. Eg. Updating the function F = {0|-> 0, 1|-> 0} so it maps 1 to itself is achieved by, F := F <+ {1|-> 1}.

Check the following properties: Take into account there is a problem in ProB where a CTL specification starting with AF leads to a false negative, please use LTL in these cases.

Questions

- Either player has the possibility to win the game
  Solution: Both players can possibly win: EF {winner = player1}) & (EF {winner = player2}

- Eventually, one of the players must win
  Solution: One of the two players has to win: F {winner = player1 or winner = player2}

- Every game comes to an end
  Solution: Every game ends eventually: F {winner $\neq$ no}

- When the game has come to an end, it remains over forever
  Solution: When the game ends, it keeps being over: AG {winner $\neq$ no} $\Rightarrow$ AG {winner $\neq$ no}

- The game is not over until all the heaps are empty
  Solution: The game is not over until the heaps are all empty: AG {winner = no} $\vee$ {heap_count(4) = 0}

- There is a game that lasts for exactly 16 moves(turns)
  Solution: There is a game that takes 16 moves before somebody wins: EF {winner $\neg$ no & turn_counter = 16}

- There is no game that lasts for at least 17 moves
  Solution: There is no game that takes 17 moves before somebody wins: not EF {winner ¬ no & turn_counter = 17}

- Suppose the first player wants the game to end as quickly as possible. How many moves does he need? (hint: use two specifications — one to show that the first player can always make sure that the game ends after n moves, and one to show that this is not the case for $n-1$)
  Solution: Player one cannot force the game to end after 6 moves: not EX AX EX AX EX AX AX {winner ≠ no}
  player one can force the game to end after 7 moves: EX AX EX AX EX AX EX AX {winner ≠ no}

- Suppose the first player wants the game to go on for as long as possible. For how many moves can he avoid the game end? Keep in mind player 2 will not cooperate.
  Solution: Player one can force the game to last for at least 6 turns: EX AX EX AX EX AX {winner = no}
  player one cannot force the game to last for at least 7 turns: not EX AX EX AX EX AX EX {winner = no}

- It can be shown that games like this are either *first-player-wins* or *second-player-wins*, that is, one of the players has a winning strategy: no matter what the other player does, he can always win the game. Which player has a winning strategy in this game? Stated differently, state the proposition that the first player has a winning strategy and verify. Then state the property that the second player has a winning strategy and verify. Again, keep in mind the other player will not cooperate.
  Solution: Player two has a winning strategy: AX EX AX EX AX EX AX EX AX EX AX EX AX EX AX EX AX EX {winner = player2}
  player one has no winning strategy: not EX AX EX AX EX AX EX AX EX AX EX AX EX AX EX AX EX AX EX {winner = player1}

---

9 Farmer puzzle

Open the file `Farmer.mch`. This file contains the skeleton for a machine, modelling the following puzzle:

A farmer wants to cross a river and take with him a fox, a chicken, and grain. There is a boat that can fit himself plus either the fox, the chicken, or the grain. If the fox and the chicken are alone on one shore, the fox will eat the chicken. If the chicken and the grain are alone on the shore, the chicken will eat the grain.

How can the farmer bring the fox, the chicken, and the grain across the river?

Questions

- Complete the machine, so that it properly models the puzzle. You only need to fill in the operations `Move_far` and `Move_near`.

- Show that there is a solution.
  Solution: EF {card(far) = 4}

- Show that as long as you haven't lost, you can still reach a solution. Use `e(YouLose)` to show the `YouLose` operation is enabled, you don't have to put this value in curly brackets.
  Solution: AG (not e(YouLose) => EF {card(far) = 4})

- Show that you are obligated to cross the first time with the chicken.
  Solution: not EX ({far = {chicken}} & EF {card(far) = 4})