Automated Translation from Event-B specifications to Recursive Algorithms

- 21/10/2013 - ver.0.2

Zheng Cheng and Rosemary Monahan

Computer Science Department
National University of Ireland Maynooth
Co. Kildare, Ireland

Abstract

Event − B is a modelling language. It allows the user to develop the software or algorithm in a step-by-step manner (i.e. refining an initial high level specification into a final concrete specification). In previous work, one of the author and her colleague describe a framework that translates the final concrete Event-B specification into its corresponding recursive and iterative implementation. In this document, we provide the technical details of how the translation is performed. Moreover, we interest in the visualization of recursive algorithms to enhance their readability and understandability.

Keywords: Event-B, Recursive Algorithm

1 Introduction

Event-B is a formal modelling language, based on the *refinement calculus*. It uses set theory as a modelling notation, and use refinement to represent software systems at different abstraction levels. The use of mathematical proof will verify the consistency between refinement levels.

Rodin platform is a tool set that helps organize the information for systems written in Event-B. In addition, it provides theorem proving facilities that allow mathematical proofs to be semi-automatically discharged.

An Event-B model consists of *contexts* and *machines*. The contexts give static information about the model, which will be referred in the machines. The machines express dynamic information about the model via *events*. They can modify *state variables* and cause the Event-B model moving into different states. Optionally, a machine can express other properties, such as invariants and safety properties of the model.

Each event can be triggered by conditions (i.e. the *guards*) to take *actions*. When the Event-B model is in a state that satisfies all the guards of an event, such event will take effect by performing defined actions.

The usage of *control variable* is an mechanism to control how the events interact with each other: the user first defines a set of control labels in the context, and declares a state variable (a.k.a the control variable) in the machine to refer these labels. The control variable does not serve any purposes to the state of an Event-B model. It only keeps track of an implicit control flow of the Event-B model. More specifically, if an event's guard refer to the control variable, it implies which events that lead to the current event. If an event's actions refer to the control variable, it suggests which events the current event can move into.

In previous work, Méry and Monahan develop a framework that translate an Event-B specification into its recursive and iterative implementation [1]. In this document, we draw on the usage of control variable to develop a Rodin plugin that implements this translation. This plugin reads in an Event-B model, then produces recursive algorithm and its visualized representation.

Outline The rest of the paper is organised as follows. Section 2 illustrates the technical details of our translation procedure. The overall result is shown in Section 3.

2 The Translation Procedure

To allow our plug-in understand that how to process an Event-B model, the user needs to define a configuration file. This configuration file should specify at least:

- The method signature under consideration.
- The name of the control variable.
- The name of the start label.
- The name of Event-B machine to be processed.

Our plugin reads in the configuration file and starts to process the Event-B model. The target of plugin is the final concrete specification which specified in the configuration file.

To reduce the translation complexity from an Event-B machine to its corresponding recursive algorithm, the plug-in extracts required information out of each event in the original Event-B machine, and stores in a data structure called *bEventObject* (see Fig 1).

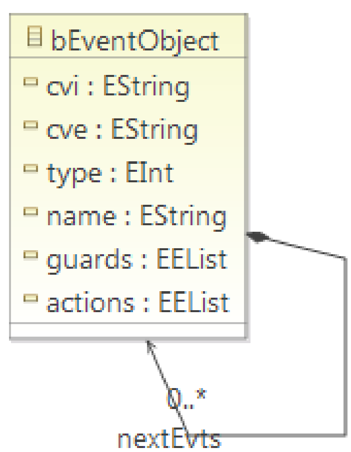

Figure 1: The Data Structure of bEventObject

A bEventObject E_o is a 7-tuple, where $E_o = (cvi, cve, type, name, guards, actions, nextEvts)$. It consists of:

- The control variable initial (cvi).
- The control variable end (cve).
- The type of the event (type).

- The name of the event (name).
- A set of guards of the event (guards).
- A set of actions of the event (actions).
- A set of E_o (nextEvts).

Next, we describe how to extracted these information from the event under consideration.

Each event can reference the control variable in the guards or actions. This control variable controls the order that events take place. The *cvi* and *cve* in the bEventObject (see Fig 1) are short hand for control-variable-initial and control-variable-end. The former one is used to determine which events that lead to the current event. The latter one is used to determine which events the current event can move into. They are derived from the guards and actions of the original event respectively (i.e. extracting the action/guard that references the control variable).

The *type* is determined by the name of an event:

- An event has a **recursive** type if the event's name starts with **REC** (case insensitive).
- An event has a **call** type if the event's name starts with **CALL** (case insensitive).
- An event has a **normal** type if it is none of the above cases.

The *guards* are derived according to the following rules:

- The guards that reference the control variable are not included for any event.
- The guards are not included for the event of recursive or call type.
- In the case of recursive or call type for the event, additional guards might be added from the event name, depending on whether guards appear in the event name (see Section 2.1).

The *actions* are derived according to the following rules:

- The non-deterministic actions are not included for any event, i.e. becomes_such_that assignment and becomes_in_set assignment are eliminated when parsing the event actions.
- The actions that reference the control variable are not included for any event.
- The actions is not included for the event of recursive or call type.
- An additional action is added from the event name for an event of recursive or call type (see Section 2.1).

The *nextEvts* association in a bEventObject helps the plug-in to understand how the transition system progress (i.e. where an event should moves to the next). An event **x** counts as the next event of a target event **y** if it has the following property:

```
x.cvi = y.cve
```

2.1 Processing Event with Recursive/Call Type

The recursive (or call) type events must follow the following naming convention, so that the plug-in knows how to process it:

```
rec@call_signature@Guards@self_destructed
```

The **rec** indicates this event is of recursive type. The **call_signature** part indicates the name of the function call to be invoked. It takes the format:

```
call_name(in_parameters; out_parameters)
```

This function signature is easy to turn into an Event's action, and added to the action list of a bEventObject.

As described in Section 2, the guards for events of recursive/call type are not included in the guards of the bEventObject. The reason is that the guards for an event of recursive/call type is explicitly given in the event's name. In the case that an event of recursive/call type has no guards, we specify **NULL**.

It is possible that more than one event's name contain the same function signature, where only the guards of these events differ. They show different outcomes when executing the same recursive function call. Thus, they should be combined. The plug-in uses **self_destructed** part in the event name to control which event to display (i.e. among all related events for the same recursive call, only one of them is displayed).

Eventually, each event is able to be translated into an bEventObject and be related through the *nextEvts* association. Next, we illustrate the algorithms that translate bEventObjects into a control flow graph (Section 2.2) and the recursive algorithm (Section 2.3).

2.2 Representing in Control Flow Graph

An intuitive diagram allows easier understanding of the algorithm, and is a prerequisite for modularizing complex algorithms. Therefore, we construct a control flow graph for each Event-B model by using bEventObjects.

The control flow graph is defined as $CFG = (G, Act, E_{act}, Grd, E_{grd})$, where:

- G is a directed graph $G = (N, E, S_g, T_g)$, where N is a set of nodes; E is a set of directed edges; $S_g : E \to N$ is the source function for edges; and $T_g : E \to N$ is the target function for edges.
- Act is a set of actions.
- $E_{act} : E \to Act$ is an action function.
- Grd is a set of guards.
- $E_{grd} : E \to Grd$ is an guard function.

Our plugin iterates over all the *bEventObjects* to construct such a CFG. On traversing each *bEventObject*, our plugin:

1. Instantiates two nodes, N_{cvi} and N_{cve} , corresponding to the *cvi* and *cve* in the *bEventObject*, and adds them to the set N.
2. Adds the *actions* of this *bEventObject* into the set Act.
3. Adds the *gurads* of this *bEventObject* into the set Grd.

4. Creates a fresh edge e and add into E, where $S_g(e) = N_{cvi}$ and $T_g(e) = N_{cve}$; Then, adding $\{e \mapsto actions\}$ to E_{act} (i.e. attachs Event-B actions to the edge e); Finally, adding $\{e \mapsto guards\}$ to E_{grd}.

2.3 Representing in Recursive Algorithm

To help generate a recursive implementation from the Event-B specification, we design the pretty print procedure *Print* as indicated in Alg 1.

Algorithm 1 Representing Event-B Machine as Control Flow Graph

function PRINT(o: *bEventObject*)
    PRINTGUARDS(o.guards)
    PRINTACTIONS(o.actions)
    for each *bEventObject* $e \in o.nextEvts$ **do**
        PRINT(e)
    end for
end function

2.4 Proof Obligations

There are a set of proof obligations that could be generated to ensure that an Event-B machine can be translated into a recursive algorithm:

- The control variable in the actions and guards of each event are different. (i.e. the event always progresses)
- The labels in an Event-B machine forms an acyclic graph.
- Only one event does not have control variable in its guards (i.e. the start event).
- Only one event does not have control variable in its actions (i.e. the end event).
- The control variable in the events' actions is deterministic (i.e. An Event always know which label it should move into).
- If an event has more than one outgoing edges, then all these edges should be labelled with guards, and eventually converged. Moreover, these guards should be independent to each other.

3 Case Study

3.1 Binary Search Algorithm

The first case study targets the **binary search** algorithm developed in the Event-B machine (a part of the machine is displayed in Fig 2).

EVENT m1 REFINES find
WHEN
$grd1 : l = start$
$grd2 : lo = hi$
$grd3 : t(lo) = val$
WITNESSES
$j : j = lo$
THEN
$act1 : l := end$
$act2 : ok := TRUE$
$act3 : i := lo$
END

EVENT m3 REFINES find
WHEN
$grd1 : l = middle$
$grd3 : t(mi) = val$
WITNESSES
$j : j = mi$
THEN
$act1 : l := end$
$act2 : ok := TRUE$
$act3 : i := mi$
END

EVENT m2 REFINES fail
WHEN
$grd1 : l = start$
$grd2 : lo = hi$
$grd3 : t(lo) \neq val$
THEN
$act1 : l := end$
$act2 : ok := FALSE$
END

EVENT split
WHEN
$grd1 : l = start$
$grd2 : lo < hi$
THEN
$act1 : l := middle$
$act2 : mi := (lo + hi)/2$
END

EVENT REC@rightsearchOK REFINES find
ANY j
WHERE
$grd1 : l = middle$
$grd2 : val > t(mi)$
$grd3 : j \in mi + 1 \,..\, hi$
$grd4 : t(j) = val$
$grd5 : mi + 1 \leq hi$
THEN
$act1 : i := j$
$act2 : ok := TRUE$
END

EVENT REC@rightsearchKO REFINES fail
WHEN
$grd1 : l = middle$
$grd2 : val > t(mi)$
$grd4 : \forall j \cdot j \in mi + 1 \,..\, hi \Rightarrow t(j) \neq val$
$grd5 : mi + 1 \leq hi$
THEN
$act2 : ok := FALSE$
END

Figure 2: Event-B machine developed for the Binary Search Algorithm

As described in Section 2.1, the event of recursive/call type need to follow the naming convention so that the plug-in knows how to process it. In this example, the **REC@rightsearchOK** and **REC@rightsearchKO** are event of recursive type. Their event name has been shortened in the above machine. The REC@rightsearchOK is a shorthand for:

```
rec@binsearch(t,mi+1,hi,val;ok,result)@NULL@SELF_DESTRUCTED
```

and the REC@rightsearchKO is shorthand for:

```
rec@binsearch(t,mi+1,hi,val;ok,result)@val>t(mi) && mi+1<=hi
```

The result of our translation is two-fold. First, to help people comprehend the algorithm, the plug-in reads in the Event-B machine and visualize it as in Fig 3. This is done by translating an bEventObject into the *CFG* as described in Section 2.2. We use the *Dot* tool of *GraphViz* to assist this translation [1]. In a nutshell, we draws a circle for each node, and the directed edge between two nodes indicates that an event occurs. The guards of such an event label the arrow, and the event's actions are indicated the text of the square box.

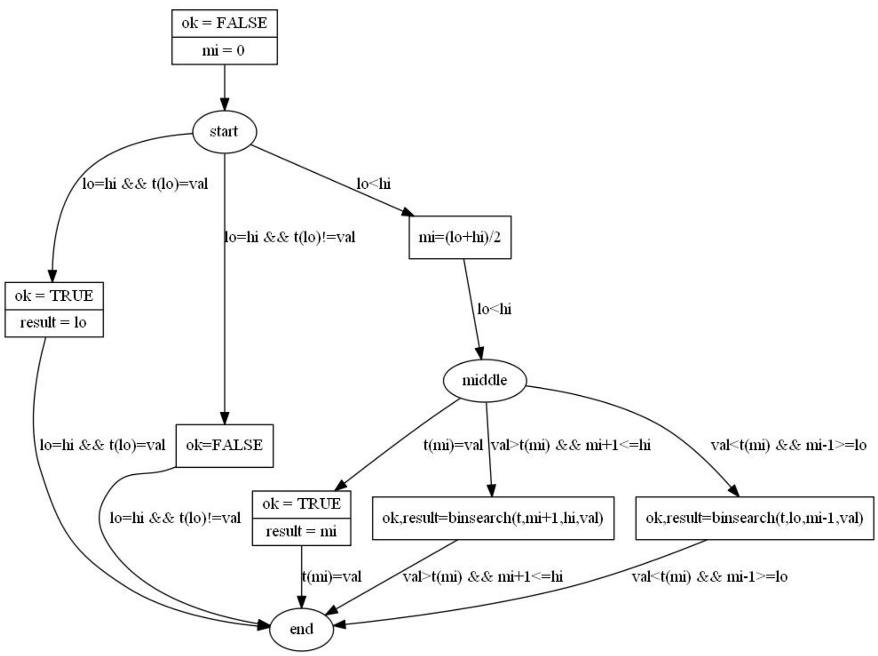

Figure 3: Visualized Representation of the Binary Search Algorithm

Second, in Fig 3, a textual representation of the binary search algorithm is given. It is constructed from the pretty print procedure *Print* as described in Alg 1.

[1] http://www.graphviz.org/

```
binsearch(t,lo,hi,val){
    ok = FALSE
    mi = 0
    if(lo=hi && t(lo)=val){
        ok = TRUE
        result = lo
    }else if(lo=hi && t(lo)≠val){
        ok=FALSE
    }else if(lo<hi){
        mi=(lo+hi)÷2
        if(t(mi)=val){
            ok = TRUE
            result = mi
        }else if(val>t(mi)∧mi+1≤hi){
            ok,result=binsearch(t,mi+1,hi,val)
        }else if(val<t(mi)∧mi-1≥lo){
            ok,result=binsearch(t,lo,mi-1,val)
        }
    }
}
```

Figure 4: Textual Representation of the Binary Search Algorithm

References

[1] Méry, D., Rosemary, M.: Transforming EVENT B Models into Verified C# Implementations. In: Lisitsa, A., Nemytykh, A. (eds.) VPT 2013 - First International Workshop on Verification and Program Transformation. EPIC, vol. 16, pp. 57–73. Alexei Lisitsa and Andrei Nemytykh, Saint Petersburg, Russia (Jul 2013)