Hoefler Text

# Automated Translation from Event-B specifications to Recursive Algorithms

**- 21/10/2013 - ver.0.1**

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 software or algorithm in a step-by-step manner, i.e. refining an inital high level specification into a final concrete specification. In previous work, one of the author and her collegue decribes an approach that translating the final concrete specification into recursive and iterative implementation. In this document, we provide technical details of how the translation is performed. Moreover, we interest in the visualization of recursive algorithm for better readability and understandability.

**Keywords:** Event-B, Recursive Algorithm

## 1 Introduction

Event-B is a formal modelling language, based on refinement calculas. 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 consistency between refinement levels.

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

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

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

The control variable is an mechanism to control how the events interact with each other**??**. The user defines a set of control labels in the context, and declare a state variable (namely the control variable) in the machine to refer these labels. The control variable does not serve any purposes to the state of 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 this document, we draw on the control variable mechanism in Event-B model, and present a plugin that read in an Event-B model to produce recursive algorithm and its visualized representation.

## 2 The Translation Procedure

To allow our plug-in understand that how to process an Event-B machine, 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 (a.k.a label).
- The name of the start label.

Our plugin reads in the configuration file and starts to process the Event-B machine. To reduce the translation complexity from a 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 it in a data structure called **bEventObject** (see Fig 1).

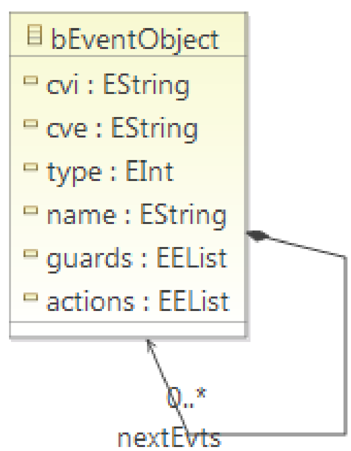


Figure 1: The Data Structure of bEventObject

The bEventObject is a 7-tuple $E_o = (cvi, cve, type, name, guards, actions, nextEvts)$, which consists of:

- The initial control variable (cvi).
- The end control variable (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 *cvi* is used to determine which events that lead to the current event. The *cve* 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** of a bEventObject is determined by the name of event under consideration:

- An event has a **recursive** type if the event name starts with **REC** (case insensitive).
- An event has a **call** type if the event name starts with **CALL** (case insensitive).
- An event has a **normal** type if it is not one of the above cases.

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

- The guards that reference the control variable are not included for the 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 1.1).

The **actions** of the bEventObject 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 1.1).

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

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

Such an event y is then added to the nextEvts list of the target event x.

## 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@grds@self_destructed
```

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

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

This signature is easy to turn into a deterministic action, which can then be added to the action list of the bEventObject.

The guards of recursive events are not included in the guards of the bEventObject. The reason is that we use the **grds** part of event name to show under which the recursive call is allowed to be invoked. In the case that an recursive event has no guards, we specify **NULL** in the grds part of the name.

Notice that it is possible that more than one event's name with the same signature exists, where only the guards of these events differ. They show different outcomes when executing the same recursive 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 a 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 the control flow graph (Section **??**) and the recursive algorithm (Section **??**).

## 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 $CFG = (G, N_{start}, Act, E_{act}, Grd, E_{grd})$, 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.
- $N_{start}$ is the start node, such that $\nexists e \in E : T_g(e) = N_{start}$.
- $Act$ 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.

$G$ defines the structure of the control flow graph. The set $N$ in $G$ is derived from the control variables in each *bEventObject*. An element in $N$ if and only if it is a label The set $E$ contains directed edge of $G$. Using $S_g$ and $T_g$, we can locate the source node and target node of a given edge respectively.

$N_start$ is the entry node of $G$, and none of the other node points to the $N_start$.

## 2.3 Representing in Recursive Algorithm

For example, pretty printing an bEventObject into a textual representation takes 3-steps, i.e. printing the bEventObject's guards, actions and nextEvts in order.

This graph describes the control flow of a recursive algorithm. The construction of such graph is decribed by the following algorithm:

**Algorithm 1** Representing Event-B Machine as Control Flow Graph

```
function toCFG(bEventObject)
    for each event e ∈ nextEvts do
        toCFG(e)
    end for
    return G
end function
```

## 3 Proof Obligations

We think 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 progress)
- The labels in the 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).
- Recursive calls and external function calls are legal (i.e. type checked, signature matched).
- If the *out* label associates with more than one event, all these events should have guard(s) presented. Moreover, these guards should not overlap and should eventually converge.

# 4 Case Study

## 4.1 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 1.1, the event of recursive/call type need to follow a 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 a format that can be recognize by the *Dot* tool of *GraphViz* [1]. In a nutshell, the plug-in draws a circle for each label, and the arrow between two circles indicates that an event occurs. The guards of such an event label the arrow, and the event's actions are indicated the text in the square box.

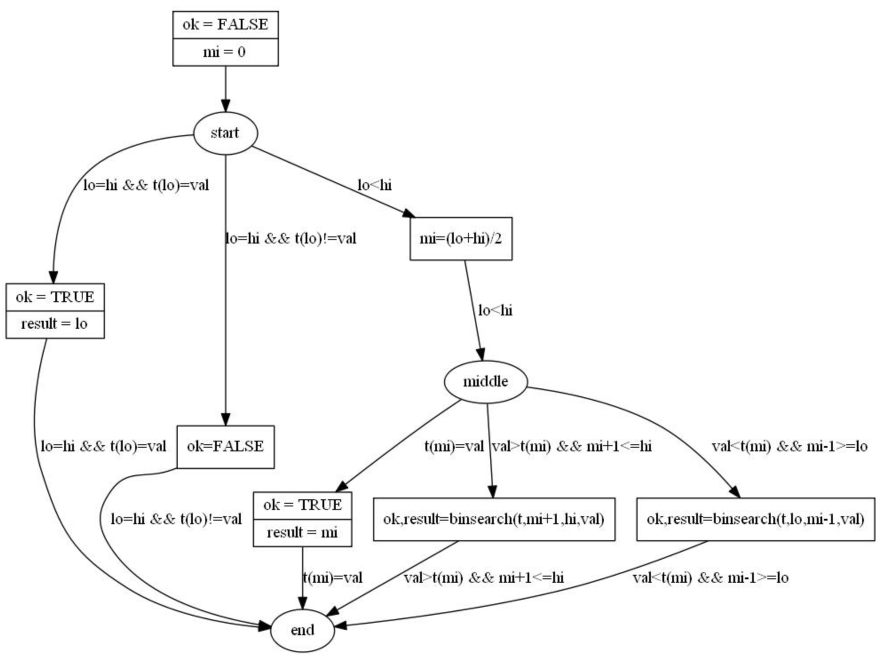


Figure 3: Visualized Representation of the Binary Search Algorithm

Second, in Fig 3, a textual representation of the binary search algorithm is given.

[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