Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 259 (2009) 47–65
www.elsevier.com/locate/entcs

Proving Quicksort Correct in Event-B

Stefan Hallerstede[1]

Institut für Informatik
Universität Düsseldorf
Düsseldorf, Germany

Abstract

The Event-B method can be used to model all sorts of discrete event systems, among them sequential programs. We have made the experience that the minimalist nature of Event-B is of advantage when it comes to tool support and to using proof as a means to analyse a model. The downside of the minimalism is that when models get more complex the lack of structure in the models can make them cluttered with auxiliary variables. System decomposition will not solve this problem. This can not be reasonably applied to a sequential program.
In this article we describe our experiences with using Event-B by way of an example. We show how we verified iterative Quicksort in Event-B and intersperse our observations and criticisms. We use them to formulate some suggestions of how we believe Event-B should evolve in future. Some of the minimalism may have to be abandoned in favour of more clarity of the produced formal models.

Keywords: Refinement, Program Verification, Rodin Tool

1 Introduction

Verification of sequential programs can be done directly [7] or by refinement [14]. Both methods yield the same result – a verified algorithm. We believe the refinement-based approach makes it easier to master the complexity of more intricate algorithms. In direct attempts at the verification of sequential programs we have to deal with all details of an implementation at once. Using refinement we can proceed more slowly, introducing the necessary details gradually. Step by step we discharge proof obligations that, taken together, establish correctness of a sequential program with respect to an abstract specifiation.

Event-B [3] can be used in this way to verify (models of) sequential programs [2,3]. Using the Rodin tool [4] proof obligations are generated automatically and updated automatically whenever we change a model. As much as possible proofs associated with a model are maintained. Proof obligations provide feedback on how to improve formal models [11] (of sequential programs). During the design of

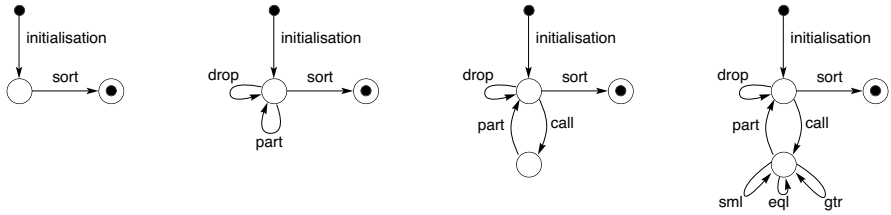
[1] Email: stefan.hallerstede@wanadoo.fr

doi:10.1016/j.entcs.2009.12.017

```
 1 : tp, m, n, p, q := 0, ∅, ∅, 1, N ;
 2 : do tp ≥ 0 →
 3 :    if p < q →
 4 :       z :∈ p .. q ;
 5 :       hz, s, e, g := h(z), p, p, q + 1 ;
 6 :       do e < g →
 7 :          if h(e) < hz → h(s), h(e) := h(e), h(s) ; s, e := s + 1, e + 1
 8 :          [] h(e) = hz → e := e + 1
 9 :          [] h(e) > hz → h(g − 1), h(e) := h(e), h(g − 1) ; g := g − 1
10 :          fi
11 :       od ;
12 :       tp, m(tp + 1), n(tp + 1), p := tp + 1, p, s − 1, g
13 :    [] p ≥ q →
14 :       if tp > 0 → tp, p, q := tp − 1, m(tp), n(tp)
15 :       [] tp ≤ 0 → tp := tp − 1
16 :       fi
17 :    fi
18 : od
```

Fig. 1. Iterative Quicksort

Event-B we have taken great care to justify design choices concerning the modelling method [10]. Event-B and the Rodin tool present a significant progress compared to classical refinement-based verification approaches like the B-Method [1]. Still, some of the choices made come at a price. In this article we want to evaluate some aspects of sequential program development in Event-B. We do not use the term "software development" because we do not consider concepts such as modularity. For our purposes a little more intricate sequential algorithm, such as Quicksort, is sufficient. We prove correctness of an iterative Quicksort algorithm similar to the one presented in [13] using the Rodin tool and Event-B. At first we proved two separate algorithms correct, Partition and Quicksort, subsequently the full Quicksort algorithm containing Partition. For the implementation of Partition itself there is a range of choices, e.g., [3,8,13]. In the final version presented here (see Figure 1) we settled for Kaldewaij's choice [13] of the Dutch National Flag algorithm. (The assignment $h(i), h(j) := h(j), h(i)$ is shorthand for $h := h \lhd \{i \mapsto h(j), j \mapsto$



(a) Model qs0 (b) Model qs1 (c) Model qs3 (d) Model qs4

Fig. 2. Evolution of the algorithm during refinement

$h(i)$}.) Note the non-determinism in line 4. We are interested in an "abstract implementation" showing that an algorithm is correct and finding out why it works.

The original Quicksort algorithm is recursive [8] which cannot be modelled in Event-B. Hence, we cannot verify the recursive Quicksort algorithms proved correct in [8] and [7] (where the latter also presents a termination proof). It should be possible to verify exactly the algorithm presented in [13] but this is not our aim. We want to investigate strengths and weaknesses of Event-B in the verification of sequential programs. Event-B is simplified [10] compared to more classical approaches to proof-based program verification by refinement [1,12]. Most of the structuring mechanisms and all control structures have been eliminated to achieve a closer correspondence between proof obligations and formal models and increase the amount of proof obligations that can be discharged automatically [10].

Although this article discusses methodical problems of using Event-B, we think that the discussion and the discussed techniques are of general interest. Similar problems appear in other methods such as VDM [12] or Circus [15] and associated tools. For instance, the technique of decomposing proof obligations by means of witnesses is also well-applicable to Circus[9]. The approach usually followed in Circus is to document corresponding heuristics, whereas in Event-B we try to implement very useful heuristics in the Rodin tool.

Overview.

In Section 2 we introduce briefly the Event-B notation and associated proof obligations. The proof obligations presented are those generated by the Rodin tool except for enabledness proof obligations that are currently not supported by the tool. In the following section we present the development (resp. correctness proof) of Quicksort. In Sections 3 to 7 we introduce the algorithm step by step as shown in Figure 2. The different figures (a) to (d) do not have a formal semantics but are only intended to clarify the structure of the different models. In Section 8 we describe the remaining five refinements that lead to the algorithm shown in Figure 1. While presenting the formal model in Sections 3 to 8, we comment on what we see as strengths and what we see as weaknesses of Event-B. Finally, we collect those remarks in Section 9 and draw some conclusions.

2 The Event-B Modelling Notation

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, where carrier sets are similar to types [6]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section 2.1, and proof obligations in Section 2.2 and Section 2.3.

2.1 Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* the event may contain. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by

any t when	when	begin
$G(t, v)$	$G(v)$	$S(v)$
then	then	end
$S(t, v)$	$S(v)$	
end	end	

The second form is used if event $E(v)$ does not have parameters, and the third form if in addition the guard equals *true*. A dedicated event of the third form is used for *initialisation*.

The action of an event is composed of several *assignments* of the form

$$x := B(t, v) \qquad \text{or} \qquad x :\in B(t, v) \qquad \text{or} \qquad x :| Q(t, v, x')$$

where x are some variables, $B(t, v)$ expressions, and $Q(t, v, x')$ a predicate. The second form assigns x to an element of a set, and the third form assigns to x a value satisfying a predicate. The first two are defined in terms of the third form

$$\begin{aligned} x := B(t, v) &\;\widehat{=}\; x :| x' = B(t, v) \\ x :\in B(t, v) &\;\widehat{=}\; x :| x' \in B(t, v) \end{aligned}$$

The effect of an assignment can also be described by a before-after predicate:

$$\text{before-after-predicate-of } \text{“}x :| Q(t, v, x')\text{”} \;\widehat{=}\; Q(t, v, x')$$

A before-after predicate describes the relationship between the state just before an assignment has occurred (unprimed variable names x) and the state just after the assignment has occurred (primed variable names x'). All assignments of an action $S(t, v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(t, v, x')$. Variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $A(t, v, x')$ with $y' = y$, yielding the predicate:

$$\boldsymbol{S}(t, v, v') \quad \widehat{=} \quad A(t, v, x') \ \wedge \ y' = y \quad .$$

2.2 Machine Consistency

For each event $E(v)$ of a machine, *action feasibility* must be proved,

$$I(v) \ \wedge \ G(t, v) \ \Rightarrow \ (\exists v' \cdot \boldsymbol{S}(t, v, v')) \quad .$$

By proving feasibility, we achieve that $\boldsymbol{S}(t, v, v')$ provides an after state whenever $G(t, v)$ holds. This means that the guard indeed represents the enabling condition of the event.

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called *invariant preservation*:

$$I(v) \ \wedge \ G(t, v) \ \wedge \ \boldsymbol{S}(t, v, v') \ \Rightarrow \ I(v') \quad .$$

2.3 Machine Refinement

A machine N can refine at most one other machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$ associated with the concrete machine N, where v are the variables of the abstract machine and w the variables of the concrete machine.

Each event $E(v)$ of the abstract machine is *refined* by one or more concrete events $F(w)$. Let abstract event $E(v)$ and concrete event $F(w)$ be:

$$\begin{array}{lcl} E(v) & \widehat{=} & \mathsf{any}\ t\ \mathsf{when}\ G(t, v)\ \mathsf{then}\ S(t, v)\ \mathsf{end} \\ F(w) & \widehat{=} & \mathsf{any}\ u\ \mathsf{when}\ H(u, w)\ \mathsf{with}\ W(t, v', u, w, w')\ \mathsf{then}\ T(u, w)\ \mathsf{end} \end{array}$$

Informally, concrete event $F(w)$ refines abstract event $E(v)$ if the guard of $F(w)$ is stronger than the guard of $E(v)$, and the gluing invariant $J(v, w)$ establishes a simulation of the action of $F(w)$ by the action of $E(v)$. The predicate $W(t, v', u, w, w')$ denotes *witnesses*. They link abstract parameters t and variables v' to concrete parameters u and variables w'. Witnesses describe for each event separately more specific how the refinement is achieved. Let $K(v, w) \ \widehat{=} \ I(v) \ \wedge \ J(v, w)$. The corresponding proof obligations for refinement are called *guard strengthening*,

$$K(v, w) \ \wedge \ H(u, w) \ \wedge \ W(t, v', u, w, w') \ \wedge \ \boldsymbol{T}(u, w, w') \ \Rightarrow \ G(t, v) \quad ,$$

action simulation,

$$K(v,w) \wedge H(u,w) \wedge W(t,v',u,w,w') \wedge \boldsymbol{T}(u,w,w') \Rightarrow \boldsymbol{S}(t,v,v') \quad ,$$

and (again) *invariant preservation*,

$$K(v,w) \wedge H(u,w) \wedge W(t,v',u,w,w') \wedge \boldsymbol{T}(u,w,w') \Rightarrow J(v',w') \quad .$$

We have to prove *witness feasibility* in order to be able to add $W(t,v',u,w,w')$ to the premises in the proof obligations above

$$K(v,w) \wedge H(u,w) \Rightarrow (\exists t,v' \cdot W(t,v',u,w,w')) \quad .$$

In the course of refinement, often *new events* $F(w)$ are introduced into a model. New events must be proved to refine the implicit abstract event *skip* that does nothing. Moreover, it may be proved that new events do not collectively diverge by proving that a *variant* $V(w)$ is bounded below,

$$K(v,w) \wedge H(u,w) \Rightarrow V(w) \in \mathbb{N} \quad ,$$

and is decreased by each new event,

$$K(v,w) \wedge H(u,w) \wedge \boldsymbol{T}(u,w,w') \Rightarrow V(w') < V(w) \quad ,$$

where we assume that the variant is an integer expression.[2] We call events that satisfy these two proof obligations *convergent*. *Anticipated* events can be used to prove convergence on a lexicographic order [5] or just to delay convergence proofs. Anticipated events can be refined by anticipated or convergent events, but must ultimately be refined by a convergent event. For an anticipated event the second proof obligation is replaced by

$$K(v,w) \wedge H(u,w) \wedge \boldsymbol{T}(u,w,w') \Rightarrow V(w') \leq V(w) \quad .$$

We may prove that whenever the abstract machine may continue by means of event $E(v)$ with guard $G(t,v)$ then the concrete machine may continue by means of concrete event $F(w)$ with guard $H(w)$ or some new events $F_1(w), \ldots, F_k(w)$ with guards $H_1(u_1,w), \ldots, H_k(u_k,w)$,[3]

$$\begin{array}{l} K(v,w) \wedge G(t,v) \Rightarrow \\ \quad (\exists u \cdot H(u,w)) \vee (\exists u_1 \cdot H_1(u_1,w)) \vee \ldots \vee (\exists u_k \cdot H_k(u_k,w)) \quad . \end{array}$$

[2] Instead of an integer expression also a finite set expression can be used.

[3] The Rodin tool does not support enabledness proof obligations at the moment. So we have "generated" the corresponding proof obligations manually.

3 Sorting

We present sequential program development in Event-B by way of the sorting algorithm Quicksort. Let N be the number of elements in the array, D the domain of the array,

$$N > 0 \land D = 1 \mathinner{..} N \quad ,$$

and P the permutations on D, that is, the set of bijections on D,

$$P = D \rightarrowtail\!\!\!\!\!\twoheadrightarrow D \quad .$$

With $b \in D \to \mathbb{Z}$ the array to be sorted we would like to specify

sort_spec

begin

$act1 \;:\; b :\mid \exists p \cdot p \in P \land \forall x, y \cdot$

$$x \in D \land y \in D \land x \leq y \Rightarrow (b' \circ p)(x) \leq (b' \circ p)(y)$$

end

Event-B requires us to prove feasibility of the non-deterministic assignment to b. This seems inappropriate because we are going to implement a program that yields such a b' that is sorted. This proves feasibility of the assignment if all assignments occurring in the implementation are feasible. We should only prove feasibility for non-deterministic assignments that remain in the implementation.

The specification of sort_spec does not follow the pattern suggested in [3] where input and output are modelled by different variables of a machine. The guard of an event is used to specify the precondition. This will show more clearly when we introduce the partitioning algorithm in the course of refinement. If we tried to refine the specification above, we would need to introduce another variable a to store the before-state of b. Otherwise, we cannot state the invariant corresponding to the sorting of the array. Let $sorted(b)$ is some predicate specifying that b is sorted. The formula $sorted(b) \land \exists p \cdot b = (b \circ p)$ is only true if b is already sorted. Using an extra variable a for the initial state we specify sorting in the intended way

$$sorted(b) \land \exists p \cdot b = (a \circ p) \quad .$$

Introducing such a variable a in a refinement we would have to introduce an auxiliary variable akin to a program counter. We want to avoid this if possible. (Program counters do not make a model more readable because they complicate the invariant.) Instead, we introduce variable a already in the abstract model. Eventually, we can refine both variables by only one variable h representing both a and b [3]. The price for this decision is that the specification is more loose and the proper algorithm will only appear right at the end of the development.

Finally, we are ready to state the specification of a sorting algorithm in Event-B. It consists of an initialisation:

qs0_initialisation

begin

$act1 \;:\; a, b :| \; a' = b' \wedge b' \in D \to \mathbb{Z}$

end

and an event *sort* that specifies sorting of array b:

qs0_sort

any p when

$chc1 \;:\; p \in P$

$chc2 \;:\; \forall x, y \cdot x \in D \wedge y \in D \wedge x \leq y \Rightarrow (a \circ p)(x) \leq (a \circ p)(y)$

then

$act1 \;:\; b := (a \circ p)$

end

(The invariants are $a \in D \to \mathbb{Z}$ and $b \in D \to \mathbb{Z}$.) We prefix the names of the events with the machine qs0, qs1, etc., to which they belong, in order to avoid confusion about the different refinement levels. In qs0_sort the existential quantifier has disappeared.

Reasoning with non-deterministic assignments in Event-B is not as well supported as reasoning about non-deterministic choices made by means of guards. This lead us to modelling non-deterministic choices that are to be refined by means of guards and those that are not to be refined by non-deterministic assignments. As a consequence, we have to remember which guards are genuine and which model non-determistic choices but this did not pose a problem in this development. But we think Event-B should support this systematically.

The Rodin tool generates proof obligations referring to $chc1$ and $chc2$ separately and provides support for instantiating parameter p in refinements. The action $act1$ of event sort_spec above would be treated atomically by the Rodin tool leading to more complicated proof obligations.

4 Partitioning

Quicksort consecutively partitions the array b until partitions of size 0 or 1 are obtained which do not need to be sorted. The boundaries of the partitions are kept on a stack and the algorithm terminates when the stack is empty.

Lower bounds of partitions are stored on a stack mS and upper bounds on a stack nS. The top of a stack is stored in a variable *top*. Finding the invariant of this

model would have been very difficult without the tool. It required many attempts and fast feedback was essential. The invariant consists of three sections, one section describing the stack data type:

$$inv1 \;:\; top \in \mathbb{N}$$
$$inv2 \;:\; mS \in 1 \mathinner{..} top \to \mathbb{Z}$$
$$inv3 \;:\; mS[\{1\}] \subseteq \{1\}$$
$$inv4 \;:\; nS \in 0 \mathinner{..} top \to \mathbb{Z}$$
$$inv5 \;:\; nS(0) = 0$$

The second section describes the partitions stored on the stack:

$$inv6 \;:\; \forall i, j \cdot i \in 1 \mathinner{..} top \wedge j \in 1 \mathinner{..} top \wedge i < j \Rightarrow mS(i) \leq mS(j)$$
$$inv7 \;:\; \forall i, j \cdot i \in 1 \mathinner{..} top \wedge j \in 1 \mathinner{..} top \wedge i < j \Rightarrow nS(i) \leq nS(j)$$
$$inv8 \;:\; \forall i, j \cdot i \in 1 \mathinner{..} top \wedge j \in 1 \mathinner{..} top \wedge i < j \Rightarrow nS(i) < mS(j)$$
$$inv9 \;:\; \forall i \cdot i \in 1 \mathinner{..} top \Rightarrow mS(i) - 1 \leq nS(i)$$

Finally the third section describes the progress of the sorting of the array (with $top = 0$ yielding a sorted array):

$$inv10 \;:\; \exists p \cdot p \in P \wedge b = a \circ p$$
$$inv11 \;:\; \forall x, y \cdot x \in D \wedge y \in nS(top) + 1 \mathinner{..} N \wedge x \leq y \Rightarrow b(x) \leq b(y)$$
$$\begin{aligned} inv12 \;:\; \forall i \cdot i \in 1 \mathinner{..} top - 1 \Rightarrow \\ (\forall x, y \cdot x \in mS(1) \mathinner{..} mS(i+1) - 1 \wedge \\ y \in nS(i) + 1 \mathinner{..} nS(i+1) \Rightarrow \\ b(x) \leq b(y)) \end{aligned}$$

All variables are initialised so that they satisfy the invariant.

qs1_initialisation

begin

$$act1 \;:\; a, b :| \; a' = b' \wedge b' \in D \to \mathbb{Z}$$
$$act2 \;:\; top, mS, nS := 1, \{1 \mapsto 1\}, \{0 \mapsto 0, 1 \mapsto N\}$$

end

The refined event qs1_sort only checks the termination condition $top \leq 0$. Note the use of the witness $p \in P \wedge b = a \circ p$ for the abstract parameter p. Its feasibility is

trivially given by invariant $inv10$. The use of witnesses adds clarity to a model and is often straightforward.

qs1_sort

when

$grd1 \ : \ top \leq 0$

with

$p \ : \ p \in P \ \wedge \ b = a \circ p$

then skip end

A new event qs1_part models partitioning of the array whose bounds are on the top of the stack. We distinguish guards and non-deterministic choices by a naming convention. Guards are named grd and choices chc. In the case of event qs1_part it becomes more apparent what we gain by modelling choices the way we do; turning $chc1 \wedge \ldots \wedge chc8$ into a non-deterministic assignment would yield a rather voluminous action simulation proof obligation in the models qs3 and qs4 where partitioning is implemented. The event should be a specification of partitioning. However, reuse of a partitioning algorithm developed beforehand was difficult. The specification method using two variables that we use for sorting is difficult to employ for reuse. Even with the introduction of an additional variable c in qs3 it is not clear how it could be done. Some procedure-like concept would be useful where one event resembling qs1_part would be the specification of the procedure. But in Event-B this is not possible for now. In fact we would have been content with a method to prove partitioning separately.

qs1_part (anticipated)
any p, l, r, f when
$grd1 \ : \ top > 0$
$grd2 \ : \ mS(top) < nS(top)$
$chc1 \ : \ f \in mS(top) \mathinner{..} nS(top)$
$chc2 \ : \ p \in P$
$chc3 \ : \ mS(top) \mathinner{..} nS(top) \lhd p \subseteq D \lhd id$
$chc4 \ : \ r + 1 < l$
$chc5 \ : \ r \in mS(top) - 1 \mathinner{..} nS(top)$
$chc6 \ : \ l \in mS(top) \mathinner{..} nS(top) + 1$
$chc7 \ : \ \forall x \cdot x \in (b \circ p)[mS(top) \mathinner{..} l - 1] \Rightarrow x \leq b(f)$
$chc8 \ : \ \forall x \cdot x \in (b \circ p)[r + 1 \mathinner{..} nS(top)] \Rightarrow x \geq b(f)$
then
$act1 \ : \ top := top + 1$
$act2 \ : \ mS := mS \lhd\!\!+ \{top \mapsto mS(top), top + 1 \mapsto l\}$
$act3 \ : \ nS := nS \lhd\!\!+ \{top \mapsto r, top + 1 \mapsto nS(top)\}$
$act4 \ : \ b := (b \circ p)$
end

We have made event qs1_part and event qs1_drop below anticipated to delay the

proof of termination. Delaying termination proofs has turned out to be very useful. We used machine qs1 to prove invariant preservation and enabledness, and machine qs2 to prove termination. This made our model less sensitive to modifications; proofs in model qs2 often remained valid after changes in model qs1.

Event qs1_part modifies abstract variable b. So we have to add an anticipated event to qs0 that modifies b. The need to do this originates in the rule that new events must refine skip — which provides non-interference in concurrent models. It seems odd to have to do this when modelling sequential programs. Maybe a more liberal refinement relation could be used.

qs0_part (anticipated)

begin

$act1 \;:\; b :\in D \to \mathbb{Z}$

end

If there is at most one element in the partition on top of the stack that partition can be dropped. It can only be ordered.

qs1_drop (anticipated)

any p, l, r, f when

$grd1 \;:\; top > 0$

$grd2 \;:\; mS(top) \geq nS(top)$

then

$act1 \;:\; mS, nS, top := \{top\} \lhd mS, \{top\} \lhd nS, top - 1$

end

5 Termination

Currently, it is not possible to define mathematical functions in Event-B, such as the inductive function t below. Function t denotes the expression

$$\textstyle\sum_{i \in 1..top} (nS(i) + 1 - mS(i) + 1)^2$$

which we use in the variant expression of the termination proof.

$inv13 \;:\; t \in 0 \,..\, top \to \mathbb{N}$

$inv14 \;:\; t(0) = 0$

$inv15 \;:\; \forall i \cdot i \in 1 \,..\, top \Rightarrow t(i) = (nS(i) + 1 - mS(i) + 1)^2 + t(i-1)$

We have not found a simpler variant than $t(top)$

variant $t(top)$

We do not present the whole model qs2 but only the assignments that have to be added to the different events. In particular, the second assignment looks complicated. But it was really suggested by the proof obligation of invariant preservation for $inv15$.

$$t := \{0 \mapsto 0, 1 \mapsto (N+1)^2\} \qquad \text{qs2\_initialisation}$$

$$\begin{aligned} t := t \Leftarrow \{top \mapsto (r+1-mS(top)+1)^2 + \\ t(top-1), \\ top+1 \mapsto (nS(top)+1-l+1)^2 + \\ (r+1-mS(top)+1)^2 + \\ t(top-1)\} \end{aligned} \qquad \text{qs2\_part (convergent)}$$

$$t := top \lhd t \qquad \text{qs2\_drop (convergent)}$$

In the recursive version of [7] that follows the same structure as the algorithm presented here the variant simply is the size of the partitions passed in the recursive calls. Because the size can be 0 this does not work here. Dropping the top element from the stack would not decrease the variant. This is an example where the recursive version of an algorithm seems easier to verify than the iterative version; there would be no need to model the stack and the variant would be simpler.

6 Structure

In the third refinement we drop variable t that we used to prove termination and prepare to implement partitioning. In order to identify the section of the machine that computes the partitions we have to introduce an auxiliary variable $part$ of boolean type.

qs3_initialisation

begin

$act1 \;:\; a, b :| \; a' = b' \wedge b' \in D \to \mathbb{Z}$

$act2 \;:\; top, mS, nS := 1, \{1 \mapsto 1\}, \{0 \mapsto 0, 1 \mapsto N\}$

$act3 \;:\; part := FALSE$

$act4 \;:\; z :\in \mathbb{Z}$

end

Variable *part* is used to attach certain invariants to those computations expressing sequential control flow between the events. It models the abstract program counter, a concept that we tried to avoid in the beginning.

$$inv16 \;:\; part = TRUE \Rightarrow top > 0$$
$$inv17 \;:\; part = TRUE \Rightarrow mS(top) < nS(top)$$
$$inv18 \;:\; part = TRUE \Rightarrow z \in mS(top) \mathinner{..} nS(top)$$

A new event qs3_call initialises the computation of a partition. At first it only contains the choice of the pivot element location z. We have to prove termination of this new event. (A suitable variant is $\{part\} \cap \{FALSE\}$.) Given that we really model a sequential composition, this seems unnecessary. We would like to keep termination proofs to loops and the like.

qs3_call (convergent)

when

$grd0 \;:\; part = FALSE$

$grd1 \;:\; top > 0$

$grd2 \;:\; mS(top) < nS(top)$

then

$act0 \;:\; part := TRUE$

$act1 \;:\; z :\in mS(top) \mathinner{..} nS(top)$

end

qs3_part (convergent)

any p, l, r when

$grd0 \;:\; part = TRUE$

$chc2 \ldots chc8$

with

$f \;:\; f = z$

then

$act0 \;:\; part := FALSE$

$act1 \ldots act4$

end

Event qs3_part contains only the non-deterministic choice of partitioning to be implemented. The pivot position is chosen in event qs3_call so we only need to specify a witness for f stating that z contains that choice. Although nothing essential changes in event $qs3\_drop$ we have to refine it. We have to specify that it only can occur when $part = FALSE$. If we had some structuring mechanism we would not

have to refine $qs3\_drop$ at this stage.

qs3_drop (convergent)
  any p, l, r, f when
    $grd0 \;:\; part = FALSE$
    $grd1 \;:\; top > 0$
    $grd2 \;:\; mS(top) \geq nS(top)$
  then
    $act1 \;:\; mS, nS, top := \{top\} \lhd mS, \{top\} \lhd nS, top - 1$
  end

Finally we remark that we do not have local variables. Variable z is global and must be initialised. Variable $part$ is also global and is visible in all proof obligations. With some structuring this would not be necessary. We would also have less proof obligations because, for instance, event $qs3\_drop$ would not have to be refined.

7 Trisection

In the fourth refinement we implement array partitioning by the Dutch National Flag algorithm. Event qs4_call is used for its initialisation. All variables are global. We have to assign values to them in the initialisation event ($act4$). We could write anything that type-checks. A concept of local variables would provide more clarity.

qs4_initialisation
  begin
    $act0 \ldots act3$
    $act4 \;:\; c, s, e, g :\in \mathbb{Z} \to \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}$
  end

All invariants must be prefixed with "$part = TRUE \Rightarrow$". We have avoided to introduce program counters from the beginning. That is why we started with two variables a and b for the array. And this is the reason. As soon as we have control flow dependencies between events the model gets complicated. A few such auxiliary variables can make an invariant quite difficult to read. The control flow of the events

is also not easy to infer from the corresponding guards of the events.

$$
\begin{aligned}
inv19 &: part = TRUE \Rightarrow (\exists p \cdot p \in P \land c = b \circ p \land \\
&\qquad mS(top) \mathinner{..} nS(top) \lhd p \subseteq D \lhd id) \\
inv20 &: part = TRUE \Rightarrow s \in mS(top) \mathinner{..} e \\
inv21 &: part = TRUE \Rightarrow e \in mS(top) \mathinner{..} g \\
inv22 &: part = TRUE \Rightarrow g \in mS(top) \mathinner{..} nS(top) + 1 \\
inv23 &: part = TRUE \Rightarrow (\forall x \cdot x \in c[mS(top) \mathinner{..} s - 1] \Rightarrow x < b(z)) \\
inv24 &: part = TRUE \Rightarrow (\forall x \cdot x \in c[s \mathinner{..} e - 1] \Rightarrow x = b(z)) \\
inv25 &: part = TRUE \Rightarrow (\forall x \cdot x \in c[g \mathinner{..} nS(top)] \Rightarrow x > b(z)) \\
inv26 &: part = TRUE \Rightarrow b(z) \in c[s \mathinner{..} g - 1] \qquad \text{(theorem)}
\end{aligned}
$$

Event-B has explicit support to prove some invariants as theorems, that is, it is proved that it is implied by the other invariants. This is an important an useful feature. Proving $inv26$ as an invariant is complicated. We would have to show that the events qs4_sml and qs4_gtr that modify c, s, and g, maintain the pivot element $b(z)$ in the middle part $s \mathinner{..} g{-}1$ of the array. By contrast, the theorem is a simple consequence of $inv19$ to $inv25$.

qs4_call (convergent)

when

$grd0 \ldots grd2$

then

$act0 \ldots act1$

$act2 : c := b$

$act3 : s := mS(top)$

$act4 : e := mS(top)$

$act5 : g := nS(top) + 1$

end

qs4_sml (convergent)

any p, l, r, f when

$grd0 : part = TRUE$

$grd1 : e < g$

$grd2 : c(e) < b(z)$

then

$act1 : c := c \lhd\!\!+ \{s \mapsto c(e), e \mapsto c(s)\}$

$act2 : s, e := s + 1, e + 1$

end

qs4_eql (convergent)

any p, l, r, f when

$grd0 \ : \ part = TRUE$

$grd1 \ : \ e < g$

$grd2 \ : \ c(e) = b(z)$

then

$act2 \ : \ e := e + 1$

end

qs4_gtr (convergent)

any p, l, r, f when

$grd0 \ : \ part = TRUE$

$grd1 \ : \ e < g$

$grd2 \ : \ c(e) > b(z)$

then

$act1 \ : \ c := c \Leftarrow \{g - 1 \mapsto c(e), e \mapsto c(g - 1)\}$

$act2 \ : \ g := g - 1$

end

As before the more complicated witness is just a copy of a part of an invariant ($inv19$). The witnesses for l and r explain how the result of the Dutch National Flag algorithm is used to achieve array partitioning.

qs4_part (convergent)

any p, l, r when

$grd0 \ : \ part = TRUE$

$grd1 \ : \ e \geq g$

with

$p \ : \ p \in P \wedge c = b \circ p \wedge mS(top) \mathinner{..} nS(top) \lhd p \subseteq D \lhd id$

$l \ : \ l = g$

$r \ : \ r = s - 1$

then

$act0 \ldots act3$

$act4 \ : \ b := c$

end

8 Implementation

We have made five more refinements to arrive at the algorithm of Figure 1. In qs5 variable nS is replaced nT where the index 0 is eliminated from the stack. In qs6 the top of the stack is stored in variables p and q and the stack is adapted accordingly. In qs7 the pivot element is stored in a variable hz. In qs8 the variables a, b, and c are replaced by a single variable h. Finally, in qs9 the restrictions on the stack are relaxed so that the top elements do not need to be deleted when tp is decreased.

qs5	$inv27$	:	$nT = 1 \mathinner{..} top \lhd nS$
qs6	$inv28$	:	$tp = top - 1$
	$inv29$	:	$ms = 1 \mathinner{..} tp \lhd mS$
	$inv30$	:	$ns = 1 \mathinner{..} tp \lhd nT$
	$inv31$	:	$tp \geq 0 \Rightarrow p = mS(top) \land q = nT(top)$
qs7	$inv32$	:	$part = TRUE \Rightarrow hz = b(z)$
qs8	$inv33$	:	$part = TRUE \Rightarrow h = c \land h = b$
qs9	$inv34$	:	$m \in \mathbb{N}_1 \pfun \mathbb{Z}$
	$inv35$	:	$n \in \mathbb{N}_1 \pfun \mathbb{Z}$
	$inv36$	:	$1 \mathinner{..} tp \lhd ms = m$
	$inv37$	:	$1 \mathinner{..} tp \lhd ns = n$

The sequential compositions in lines 4-5, 7, and 8 in Figure 1 would each need an auxiliary variable and we would have to introduce new program counters. In addition we would have to introduce new variables for intermediate values of modified variables or introduce anticipated events events earlier in the model. Both options appear quite heavyweight. Event-B does not cope well with sequential components of a model. We have simply rewritten our final model to suit Figure 1 because it was obvious how to do it. This is not the place where we would have expected tedious refinements.

9 Conclusion

We have seen that in the verification of iterative Quicksort in Event-B we have profited from the incremental development method that is well supported by the Rodin tool. In particular, finding invariants is much easier due to quick feedback and close correspondence between models and proof obligations.

Two of the techniques used in Event-B we found to be of particular value

(i) *Anticipated events* help to structure complex termination proofs. (It does not matter whether one is interested in the associated lexicographic orders or not.)

(ii) *Witnesses* clarify refinement relationships and are, in general, easy to use. The examples of witnesses in this article are representative of our experiences.

We have made some observations that could lead to easy improvements and others that would require more profound changes of the method. In particular, we would like to keep models of more complicated problems as simple as possible.

(i) Non-deterministic assignments should be represented in the same way as guards in order to get better decomposed proof obligations. (As a side effect also primed variable names could be removed entirely from formulas in Event-B machines.)

(ii) In particular, when introducing straightforward sequential compositions the restriction that new events must refine skip seems not appropriate. It is worth investigating more refinement types.

(iii) We found that keeping ourselves track of what is proved (by making some annotations here and there in a machine) is a burden. It requires careful review of a model to be sure that we have proved all relevant facts.

(iv) Structuring notation could be used to avoid modelling program counters and the like. The advantage of the simplicity of the Event-B notation is offset against the complications arising as a consequence in the models. Note the notation we used in Figure 2 to document our model.

(v) A way to specify local variables would keep the number of global variables manageable.

(vi) A procedure-like concept would help to decompose proofs.

The practical experiences we make with Event-B serve us to study possible improvements to Event-B, such as a way to specify structure. We have to measure it against the observations reported. Our pragmatic requirement is to make modelling and proving in Event-B easier.

Acknowledgement

This research was partly supported by the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems). Leo Freitas helped to clarify some points in the article, in particular, concerning Circus.

References

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.

[2] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 51–74. Springer, 2003.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To appear.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.

[5] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and Reachability in EventB. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005*, volume 3455 of *LNCS*, pages 222–241, 2005.

[6] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1-2), 2007.

[7] K. R. Apt, , F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 2009. To appear.

[8] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *The Computer Journal*, 14(4):391–395, 1971.

[9] Leo Freitas. Private communication.

[10] Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 49–63. Springer, 2007.

[11] Stefan Hallerstede. On the purpose of event-B proof obligations. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, pages 125–138. Springer, 2008.

[12] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[13] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. International Series in Computer Science. Prentice-Hall, 1990.

[14] Carroll C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall, 1994.

[15] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in Circus. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *LNCS*, pages 451–470. Springer-Verlag, 2002.