Refining Nodes and Edges of State Machines

Stefan Hallerstede[1] and Colin Snook[2]

[1] University of Düsseldorf
[2] University of Southampton

Abstract. State machines are hierarchical automata that are widely used to structure complex behavioural specifications. We develop two notions of refinement of state machines, node refinement and edge refinement. We compare the two notions by means of examples and argue that, by adopting simple conventions, they can be combined into one method of refinement. In the combined method, node refinement can be used to develop architectural aspects of a model and edge refinement to develop algorithmic aspects. The two notions of refinement are grounded in previous work. Event-B is used as the foundation for our refinement theory and UML-B state machine refinement influences the style of node refinement. Hence we propose a method with direct proof of state machine refinement avoiding the detour via Event-B that is needed by UML-B.

1 Introduction

Theories and calculi of verification and refinement are established: for instance, Hoare logic [4], refinement calculus [13] and Event-B [2]. Hoare logic is difficult to use on a larger scale. Refinement addresses some shortcomings of Hoare logic allowing properties of less detailed abstractions to be proved before turning to the detailed implementation. However, the refinement calculi are rather restrictive when it comes to system modelling. The refinement method of Event-B relaxes some of the restrictions by abandoning most control structure and using a weaker semantic foundation. In [2] a large number of complex models are presented to demonstrate verification on a larger scale. Still, two problems remain: it can be difficult to build larger models that are inherently structured and to master more complex sequences of refinements. Our main concern in this article is making verification and refinement easier to use. To this end, we are interested in methods and techniques for stating, managing and visualising complex verification and refinement proofs.

UML-B, a UML-based notation defined on top of Event-B, has been developed over the last ten years to support the writing of more complex models with consequent structuring needs, in particular, state machines [17]. UML-B was first invented in [18] as a UML profile with translation to B and has been developed into a diagrammatic front-end to Event-B.

UML-B supports refinement of state machines but is not equipped with its own theory of refinement. It relies on a translation to Event-B using explicit

variables to represent the state machines [14]. Recently we have also evaluated the use of Event-B for the development of sequential programs [8]. The lack of control structures can make modelling of such algorithms difficult. However, the advances made by Event-B with respect to incremental proving [3] are mainly due to the lack of control structures. Avoiding the reintroduction of control structures we use state machine notation to provide the needed features [7]. The refinement method of [14] could be named "node refinement": nodes are replaced by state machines. The choice of [7], "edge refinement" is different: edges are replaced by state machines. In this article we compare the two refinement methods. We are specifically interested in their similarities.[1] For this purpose we have formalised the refinement notion underlying the conventions of UML-B via node refinement. This formalisation of edge and node refinement is independent from Event-B. It is an alternative refinement based on the diagrammatic notations and, unlike UML-B, does not involve translation into the Event-B notation. We suggest a combined method that allows us to switch between the two at any refinement step. In the future, we think they could be merged entirely, so that we would get one refinement method with perspectives of node and edge refinement. However, this is likely to change both refinement methods. We believe it is of interest to present the two methods before unifying them so that it will be easier to judge what is gained and what is lost in the unification.

In our use of state machine diagrams, they serve to describe refinement proofs. The possible execution semantics is secondary. We content ourselves with the potential of an operational interpretation. *Invariant based programming* described in [5] follows a similar approach for the construction of correct programs. It uses refinement in the sense of [19] to construct a correctness proof along with the corresponding correct program. In comparison, our approach is intended to be used for program development but also for systems modelling. Our definitions of refinement obey the "statechart refinement rules for behavioural compatibility" stated in [15]. However, we focus on the development of a proof method whereas [15] uses the rules to formulate an approach for test case generation. The related [12] focuses on common patterns of structural refinement that could be used with state machines. In [11] formal semantics of state machines is discussed and proof rules for superposition refinement are proposed. By contrast, we use the more general Event-B refinement as a foundation of our approach. Comparatively simple structural refinement for state machines based on Event-B has been discussed in [16]. In [6] JSD-like diagrams are used to illustrate concurrent Event-B models and their refinement but the diagrams are not formally linked to Event-B models.

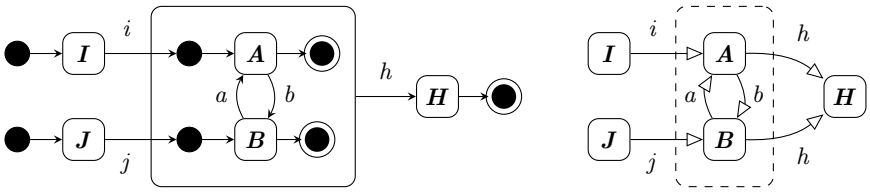
Overview. In Section 2 we briefly introduce the state machine notation that we use and in Section 3 we outline the two refinement methods. In Section 4 we present the construction of an iterative Quicksort algorithm using the two methods side by side. This could give the impression that the two notions are interchangeable. In Section 5 we present a development by node refinement of a simple controller which is not an edge refinement and we suggest a combined

[1] When looking at [14] and [7] the similarities are far from obvious.

refinement method that permits mixing node and edge refinement. Section 6 draws a conclusion and sketches some future work.

2 State machines

State machines are a diagrammatic modelling notation where the condition of a system is represented by states (denoted by the nodes of a graph) and the behaviour of the system is represented by transitions connecting the nodes (denoted by edges of a graph). The UML contains a hierarchical state machine notation which is widely used in industry and that has been adopted by UML-B. Fig. 1(a) shows a typical UML state machine. For our purposes it is easier



(a) UML notation (b) Reduced notation

Fig. 1: State machine notation

to work with a simpler reduced notation without explicit initial "●" and final states "⊙" as shown in Fig. 1(b). This makes it easier to define (refinement) proof obligations.

In this paper, we represent a node as ▢ and an edge, connecting two nodes, as —▷. Convergent loop edges ↻ may be used to indicate that the loop edge may only be followed finitely often before an "ordinary edge" is followed. The restriction to convergent *loops* is inherited from Event-B where event may be marked 'convergent' or 'anticipated'.[2] Loop edges are often used to prepare for introducing and proving the convergence of a more complex loop involving several edges and nodes. Edges are labelled with *events* that describe the effect of following that edge. An event has the shape any p when g then $x := a$. The *parameters* p are non-deterministically chosen when an event occurs. The *guard* g of an event states the condition, a first-order predicate, under which the event may occur.[3] If its guard is true an event is said to be *enabled*. The *action* of an event is an (simultaneous) update statement of the form $x := a$ where x is a variable (list) of the state machine containing the event and a is an expression (list). Clauses of an event are simply left out when they would have no effect.

[2] We do not distinguish those two concepts but simply allow convergence to be proved at later refinement steps.

[3] Predicates p, q written on consecutive lines are implicitly conjoined.

The parameters may be left out if there are none, a guard if it is *true*, an action if it is $x := x$.

Nodes are labelled by *assertions*. If $\boldsymbol{A}$ is the label of a node, we write "@$\boldsymbol{A}$ p" to say that "$\boldsymbol{A}$ contains p" or, in other words, "p holds at $\boldsymbol{A}$". We also call assertions of nodes with loop edges *invariants*.[4] In formulas we use $\boldsymbol{A}$ to stand for p. State machines are a notation for proofs similarly to proof outlines [4]. An edge labelled e where $e =$ any p when g then $x := a$ connecting a node labelled $\boldsymbol{A}$ to a node labelled $\boldsymbol{B}$ corresponds to a proof obligation: $\boldsymbol{A} \wedge g \Rightarrow \boldsymbol{B}[x := a]$. Formal proof is the central aspect of our notation replacing the operational view of UML-B.

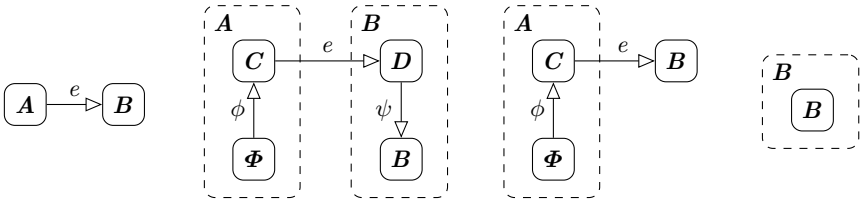
State machine notation supports hierarchical construction where state machines may be nested within a node of the parent state-machine. We refer to the node containing the nested state machine as a super-node and represent it as $\overline{(\_)}$. Super nodes structure assertions: if a super node $\boldsymbol{A}$ contains a node (or super node) $\boldsymbol{B}$ then $\boldsymbol{B}$ contains all assertions that $\boldsymbol{A}$ contains. This is their only function in our approach. We do not attach any operational meaning to super nodes. Super nodes are essential in our definition of node and edge refinement. Super nodes (themselves) are not connected by edges. Sometimes we draw an edge exiting a super-node as an abbreviation for an edge that exits all contained nodes. This is often used in node refinement diagrams. For edge refinement diagrams we need a third kind of edge: anonymous edges $\rightarrow$ that are not labelled. They can be imagined to be labelled with *skip*, the event that is always enabled and does not change the state. In a state machine we identify *initial nodes* to be those nodes that do not have entering edges, and *final nodes* to be those nodes that do not have exiting edges. An anonymous edge entering a super node is to be connected to the initial nodes of the contained state machine; an anonymous edge exiting a super node is to be connected to the final nodes. An anonymous edge connecting $\boldsymbol{A}$ to $\boldsymbol{B}$ corresponds to the proof obligation $\boldsymbol{A} \Rightarrow \boldsymbol{B}$. Anonymous edges are needed in edge refinement diagrams to model conditional statements.

We have adapted the notation to emphasise similarities between the two notions of refinement. In particular, we do not use the notation of [7] for edge refinement and of [14] for node refinement. This makes it easy to see the differences and similarities and suggests how combined use of the two methods is possible. (The striking similarity that results from the common notation strongly suggests combined use or unification.) We believe that it should be possible to unify the two methods completely into a single refinement method, but as a consequence they could both lose their defining characteristics: specialisation on either architectural or algorithmic refinement. The new method will have to recover the two aspects in order to provide strong methodological guidelines for the use of the unified method.

[4] By contrast, an Event-B model has only one "global" invariant. Nodes of our notation would have to be represented in Event-B by abstract program counters.

3 Refinement

We discuss the two refinement notions by means of the refinement diagrams stated in Fig. 2. The concepts are easy to generalise. See, e.g., [11] for node refinement and [7] for edge refinement. Fig. 2(a) shows a state machine that we use as an abstraction (also called *abstract model*) for the refinements shown in Fig. 2(b) to Fig. 2(d) (also called *concrete models*). The proof obligations are adapted from corresponding Event-B proof obligations. We use the same



(a) State machine (b) Refinement of $\boldsymbol{A}$ and $\boldsymbol{B}$ (c) Refinement of $\boldsymbol{A}$ (d) Reuse of $\boldsymbol{B}$

Fig. 2: Refinement diagrams

diagrams to describe both refinement methods. Edges of concrete models may be labelled with events e occurring already in the abstract model: the concrete event $e = \mathsf{any}\ q\ \mathsf{when}\ h\ \mathsf{then}\ y := b$ refines the abstract event $e = \mathsf{any}\ p\ \mathsf{when}\ g\ \mathsf{then}\ x := a$. For instance, the proof obligation for the edge labelled e in Fig. 2(b) is: $\boldsymbol{C} \wedge h \wedge W \Rightarrow g \wedge \boldsymbol{D}[x, y := a, b]$, where W is a predicate, called *witness*, that relates the concrete parameters q to the abstract parameters p. The existence of suitable parameters q must be proved: $\boldsymbol{C} \wedge h \Rightarrow (\exists q \cdot W)$. A refinement may also introduce a new name f for a refined event e by stating the abstract name in brackets behind the new concrete name: $f\ (e)$. Concrete edges otherwise labelled with events that do not occur in the abstract model are said to be *new*. New events, e.g., ϕ in Fig. 2(b), must refine *skip*, the event that is always enabled and does not change the state. For $\phi = \mathsf{any}\ q\ \mathsf{when}\ h\ \mathsf{then}\ y := b$ we have to prove: $\boldsymbol{\Phi} \wedge h \Rightarrow \boldsymbol{C}[y := b]$. For a convergent loop edge $\boldsymbol{A}$ e where $e = \mathsf{any}\ p\ \mathsf{when}\ g\ \mathsf{then}\ x := a$ we have to provide a *variant* u and prove $\boldsymbol{A} \wedge g \Rightarrow u \geq 0$ and $\boldsymbol{A} \wedge g \Rightarrow u[x := a] < u$, or the corresponding proof obligation for a refinement of e.[5] While e has not been proved convergent, we have to show for refinements $\mathsf{any}\ q\ \mathsf{when}\ h\ \mathsf{then}\ y := b$ of e that they do not "disturb" new convergent edges introduced in a refinement of $\boldsymbol{A}$ or e. We have to prove: $\boldsymbol{F} \wedge h \Rightarrow u \geq 0$ and $\boldsymbol{F} \wedge h \Rightarrow u[x := a] \leq u$ where $\boldsymbol{F}$ is a node introduced in a refinement of $\boldsymbol{A}$ or e, and u the variant of some other convergent event.

In Fig. 2(c) only (super) node $\boldsymbol{A}$ looks affected by the refinement. However, in a refinement all nodes are replaced. The outgoing edge labelled e is simply connected to node $\boldsymbol{B}$. The node $\boldsymbol{B}$ shown in the figure is considered a node of the concrete model. We can think of it as a node $\boldsymbol{B}$ inside a super node $\boldsymbol{B}$

[5] We also allow finite set as variants but do not provide proof obligations here. See [2].

(see Fig. 2(d)) that is not shown. Assertions in refinements are always added to concrete nodes. This approach avoids adding assertions accidentally to many nodes when data-refining. The super nodes in refinement diagrams are also used to indicate containment of assertions among concrete nodes. For instance, an assertion added to $\boldsymbol{B}$ in Fig. 2(b) is also added to $\boldsymbol{D}$ as indicated by the super node labelled $\boldsymbol{B}$. Edges in refinement diagrams can only connect concrete nodes. Everything not shown in a refinement diagram stays structurally unchanged.

Node refinement. Node refinement replaces a node with a super-node, hence an assertion with a collection of more precise assertions. The new nodes enable new edges to be added and old edges to be replicated (for instance, elaborating non-deterministic choices present in events, in the diagram). New edges may be added between nodes inside a (refined) super-node and must not exit or enter that super-node. Edges of the abstract state-machine must be preserved: their refinements must connect the corresponding (refined) super-nodes. A loop edge, having the same node for both its source and its target, is refined by a transition between two nodes inside the corresponding refined super-node.

Edge refinement. Edge refinement replaces an edge with a state machine that is to be inserted between the source and the target of the edge. State machines occurring in edge refinements must have at most one initial node where the execution of the modelled algorithm would start. Nodes occurring in state machines introduced by edge refinements may have at most one edge entering from other nodes. But they may have several loops. More complex diagrams can be constructed using super nodes and anonymous edges. The constructed diagrams correspond closely to proof outlines as discussed in [4].

4 Development of a sequential algorithm

In [2] it is shown how Event-B can be used for the development of sequential algorithms. The proof method is well-suited for this purpose, providing strong support for finding invariants and carrying complex termination proofs. Recently, we argued [8] that some structuring facilities would benefit the method in terms of proof methodology and potential scaling. State machines could solve some of the issues involved. Developing a sequential algorithm we present the two approaches to state machine refinement side by side. Node and edge refinement provide two different views on the same development with the same proofs, documenting and explaining different aspects of the involved refinement steps. We do not present the proof obligations and proofs in full. It is rather intricate. Instead, we want to convey that using the two refinement techniques, finding the proof and presenting it are made much easier. The associated proof obligations have been produced by imitating the notation in Event-B. That is, we have used Rodin tool [3] to carry out the proofs but the translation into Event-B has been manual.

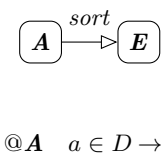
Fig. 3 gives a brief overview of the development. Along the sequence of (refined) models M1 to M7 a number of variables modelling the state of the algorithm are introduced and removed. The table provides, for each model, a short

model	introduced	removed	description	variant
M0	a		specification of sorting	
M1	b, t, m, n		introduction of outer loop and stack	
M2	C		lexicographic convergence of outer loop	C
M3		C	lexicographic convergence of outer loop	t
M4	L, R, π		introduction of inner loop	$R - L$
M5	u, v		implementation of inner loop	$(v - u) + 1$
M6	s, l, o, p, q	t, m, n	new representation of stack	
M7	h		replacement of pivot index by pivot value	

Fig. 3: Overview of the development

description of its purpose and mentions the variant used for termination proofs (if any).

M0. Fig. 4 shows the specification of the sorting algorithm consisting of a state machine, an assertion $a \in D \to \mathbb{Z}$ specified to hold at $\boldsymbol{A}$, and an event *sort* that specifies sorting of array a using a permutation p. Initially, we assert that a is

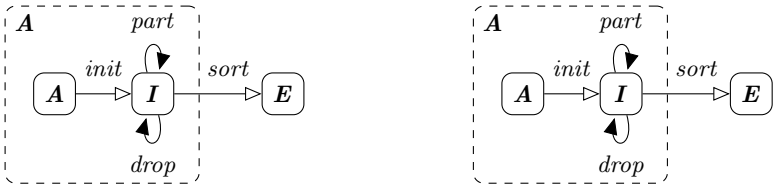


$sort =$ any p when
$p \in P$
$\forall x, y \cdot x \in D \wedge y \in D \wedge x \leq y \Rightarrow (a \circ p)(x) \leq (a \circ p)(y)$
then
$a := a \circ p$

Fig. 4: Specification of a sorting algorithm

an array with domain D and range $\mathbb{Z}$. There is nothing to prove because no assertion has been specified at $\boldsymbol{E}$. Our aim is to construct a state machine that implements iterative Quicksort based on [4] and [10].

M1. Fig. 5 shows the first node and edge refinement steps. Although the two diagrams look identical they describe different viewpoints of the same proof. Diagram 5(a) describes how the abstract node $\boldsymbol{A}$ can be replaced by a super node, indicating the internal structure of the super node and how the concrete edge *sort* is to be connected to neighbours of the super node. Diagram 5(b) describes how the abstract edge can be replaced by the four edges *init*, *part*, *drop* and *sort*. The super-state node in this diagram only indicates that at $\boldsymbol{I}$ all assertions of $\boldsymbol{A}$ hold. Event *init* sets up the variables for the loop. Event *part* specifies partitioning of the section $m(t) \,..\, n(t)$ of the array b containing at least two elements described by the top of the stack. The sub-sections $m(t) \,..\, r$ and



(a) Node refinement of $\boldsymbol{A}$ (b) Edge refinement of *sort*

Fig. 5: First refinement

$l \mathinner{..} n(t)$ are stored on the stack and the corresponding partitioning is stored in b,

$$
\begin{array}{l}
\mathit{part} = \mathsf{any}\ p\ l\ r\ f\ \mathsf{when} \\
\qquad t > 0 \land m(t) < n(t) \land f \in m(t) \mathinner{..} n(t) \land p \in P \land l > r \land \ldots \\
\qquad \forall x \cdot x \in (b \circ p)[m(t) \mathinner{..} l{-}1] \Rightarrow x \leq b(f) \\
\qquad \forall x \cdot x \in (b \circ p)[r{+}1 \mathinner{..} n(t)] \Rightarrow b(f) \leq x \\
\quad \mathsf{then} \\
\qquad b, m, n, t := b \circ p, m \lhd\!\!\!\cdot \{t{+}1 \mapsto l\}, n \lhd\!\!\!\cdot \{t \mapsto r, t{+}1 \mapsto n(t)\}, t{+}1\ .
\end{array}
$$

Event *drop* removes intervals from the stack that contain at most one element. The abstract event *sort* of Fig. 4 is refined by the concrete event *sort* of Fig. 5 (as indicated by the reuse of the name), $\mathit{sort} = \mathsf{when}\ t \leq 0\ \mathsf{then}\ a := b$. We have to prove this: using $p \in P \land b = a \circ p$ as a witness for the abstract parameter p —its existence is guaranteed by $\boldsymbol{I}$ below—, the invariant and concrete guard $\boldsymbol{I} \land t \leq 0$ imply the guard of the corresponding abstract event *sort* and the equality $a = b$ which establishes the simulation by the abstract event's action $a := a \circ p$. Among other assertions $\boldsymbol{I}$ contains the following:

$$
\begin{array}{ll}
@\boldsymbol{I} & t \geq 0 \land n(0) = 0 \land (\exists q \cdot q \in P \land b = a \circ q) \land \ldots \\
& \forall x, y \cdot x \in D \land y \in n(t){+}1 \mathinner{..} N \land x \leq y \Rightarrow b(x) \leq b(y)\ .
\end{array}
$$

We omit the proofs that the new events *init*, *part* and *drop* refine *skip*. During those proofs more assertions would be added to node $\boldsymbol{I}$ incrementally [3].

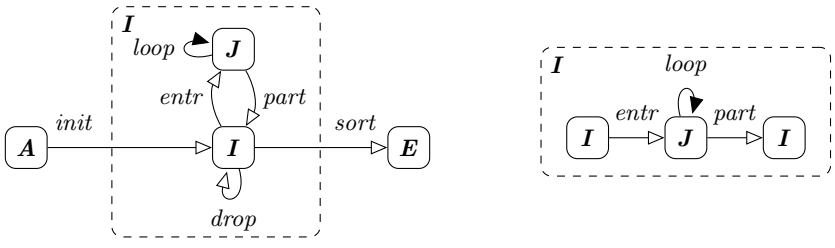
M2 and M3. In refinement step M2 convergence of event *part* is proved and convergence of event *drop* in refinement step M3, establishing a lexicographic variant (see [2]). We introduce a variable C to express the variant, adding $C := 0 \mathinner{..} N{+}1 \times 0 \mathinner{..} N{+}1$ to the action of event *init* and $C := C \setminus ((0 \mathinner{..} m(t) \times r{+}1 \mathinner{..} N{+}1) \cup (0 \mathinner{..} l{-}1 \times n(t) \mathinner{..} N{+}1))$ to the action of event *part*. We add some assertions to the node $\boldsymbol{I}$:

$$
\begin{array}{ll}
@\boldsymbol{I} & C \in 0 \mathinner{..} N{+}1 \leftrightarrow 0 \mathinner{..} N{+}1 \\
& \forall i \cdot i \in 1 \mathinner{..} t \Rightarrow m(i) \mapsto n(i) \in C \\
& \forall x, y \cdot x \mapsto y \in C \land y \leq N \Rightarrow (\forall v \cdot v \in x{+}1 \mathinner{..} y{+}1 \Rightarrow v \mapsto y \in C) \\
& \forall x, y \cdot x \mapsto y \in C \land x \geq 1 \Rightarrow (\forall w \cdot w \in x{-}1 \mathinner{..} y{-}1 \Rightarrow x \mapsto w \in C)\ .
\end{array}
$$

Using C as a variant we can prove that *part* is convergent. Event *drop* obviously does not change C. Compared to direct verification (e.g. [4]) Event-B refinement offers the advantage of introducing and removing auxiliary variables whenever it appears convenient. Compared to program refinement [13] it offers more flexibility with complex refinement steps. Convergence of *drop* can be verified with the variant t, the height of the stack. The first component of the lexicographical variant is a set, the second a number. The chosen proof method frees us from having to construct the lexicographical variant explicitly; or rather, the construction is automated.

M4. We introduce a nested loop to compute the partitioning. In this refinement step the outer loop is introduced, the inner loops in the next step. As this refinement concerns inner nodes, the node refinement diagram becomes more complicated than the edge refinement diagram. The reason for this is that node refinement diagrams can potentially express more complex refinements. An edge refinement replaces always one edge. Node refinements can replace several edges in one go. However, the node refinement diagram contains all elements that are involved in the proof. In this sense the edge refinement diagram is less complete. We have to show that *init* establishes the concrete invariant ***I***. The edge *sort* in Fig. 6(a) is redundant: neither event *sort* nor node ***E*** are changed, and ***I*** may only be stronger than its abstract counterpart. Still, both diagrams represent the same proof.

Note the difference of how loops are refined in node and edge refinement diagrams. Nodes are uniquely identifiable in node refinement diagrams whereas in edge refinement diagrams only edges need to be uniquely identifiable. An edge refinement has start and final nodes that are connected to the start and final node of the refined edge. If a loop is edge-refined, the concerned node is replicated in the refined diagram. E.g., Fig. 6(b) has two copies of node ***I***. The two copies do not denote the same node. If a loop is node-refined, the concerned node is not replicated. Instead, the loop remains in the diagram either as a loop or as a cycle involving several nodes.



(a) Node refinement of ***I*** (b) Edge refinement of *part*

Fig. 6: Fourth refinement

The node $\boldsymbol{J}$ specifies the loop invariant. It is established initially by event *entr*, where $entr = \mathsf{when}\ t > 0\ \mathsf{then}\ c, L, R, \pi := b, m(t), n(t), (m(t)+n(t)) \div 2$. At $\boldsymbol{J}$ all assertions of $\boldsymbol{I}$ hold plus the following:

$$@\boldsymbol{J} \quad \begin{array}{l} t > 0 \land m(t) < n(t) \land \pi \in m(t) \mathinner{..} n(t) \land \ldots \\ L > R{+}1 \Rightarrow m(t) < L \land R < n(t) \end{array}$$

Similarly to the first refinement step these assertions are mostly determined by the shape of the guard and action of the abstract event *part* of M1. This is driven by the proof obligations for the refinement of *part*, where $part = \mathsf{when}\ L > R\ \mathsf{then}\ b, m, n, t := c, m \lhd\!\!\!+ \{t{+}1 \mapsto L\}, n \lhd\!\!\!+ \{t \mapsto R, t{+}1 \mapsto n(t)\}, t{+}1$.[6] However, during the development, assertions were also propagated bottom up. In refinement M5 the assertions that already hold at $\boldsymbol{J}$ in m4 are essential for refinement proofs of the loop body. Note that the last three assertions at $\boldsymbol{J}$ would be difficult to guess in a top down manner. They were propagated upwards from the refinement proofs of events *swap* and *done* of M5. The guard of event *loop* has subsequently been chosen such that it preserves these assertions:

$$\begin{array}{l} loop = \mathsf{any}\ p\ l\ r\ \mathsf{when} \\ \qquad L \leq R \land p \in P \land \ldots \\ \qquad l > r{+}1 \Rightarrow (m(t) < l \land r < n(t)) \\ \quad \mathsf{then} \\ \qquad c, L, R := c \circ p, l, r \end{array}$$

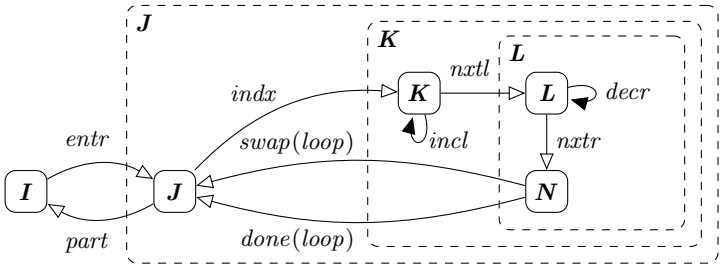
The redundancy between *loop* and $\boldsymbol{J}$ is intentional; the assertions that hold at $\boldsymbol{J}$ are established dynamically by choosing appropriate parameters p, l and r nondeterministically. Often the construction is guided by invariant preservation proofs. The same principle is already present in the B-Method [1]: it emphasises assertions and requires statement of suitable events respecting the assertions.

M5. In this refinement the body of the inner loop is implemented. It demonstrates how nested assertions are used in more complex steps of a refinement proof. In refinements M6 and M7 we will show two more refinements of the model that has now become quite complex. The degree of difficulty does not increase as the model grows in complexity. This was the main motivation that started this work on top of Event-B. We preserve the strengths of Event-B: the emphasis on reasoning, formal proof, and incremental modelling [8]. The key to incremental modelling in Event-B is the generation of fine grained proof obligations exploiting proof-oriented facts specified in formal models. Fig. 7(a) shows the node refinement where $\boldsymbol{J}$ is refined and two nested super nodes $\boldsymbol{K}$,

$$@\boldsymbol{K} \quad L \leq R \land u \leq v{+}1 \land \ldots$$

and $\boldsymbol{L}$, with $@\boldsymbol{L}\ c(u) \geq b(\pi)$, are introduced. So node $\boldsymbol{N}$, with $@\boldsymbol{N}\ b(\pi) \geq c(v)$, contains all assertions of $\boldsymbol{I}$, $\boldsymbol{J}$, $\boldsymbol{K}$ and $\boldsymbol{L}$. Following the nesting the structure could be introduced step-wise but we find that the larger step that we chose is

[6] With appropriate witnesses for the abstract parameters: $f = \pi$ and so on.

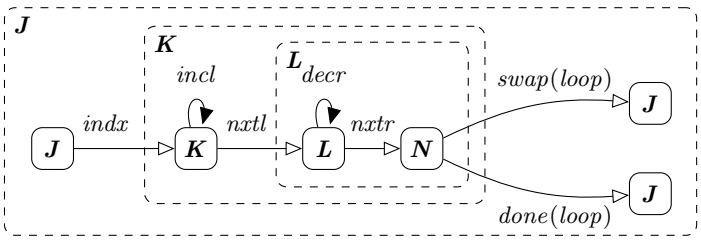


(a) Node refinement of ***J***

Fig. 7: Fifth refinement

not difficult to prove. Nothing would be gained by using additional refinement steps. In our experience the liberty in choosing the granularity of refinement steps makes it easier to produce the proof for a whole development. The mixture of program verification and step-wise refinement techniques supports the user in choosing appropriate abstractions. Supporting this mixture is not common in verification or refinement methods. The events *done*, with $done = \mathsf{when}\ u > v\ \mathsf{then}\ L, R := u, v$, and *swap*, with $swap = \mathsf{when}\ u \leq v\ \mathsf{then}\ c, L, R := c \Leftarrow \{u \mapsto c(v), v \mapsto c(u)\}, u{+}1, v{-}1$, refine the abstract event *loop* as indicated by writing the name of the abstract event name in brackets behind the concrete event names.

Fig. 7(b) shows the refinement as an edge refinement. It emphasises more how we would read the body of a loop as a sequence of commands. The structure of the inside of the loop is more obvious than in node refinement. In the corresponding node refinement one has to look more closely to identify the relevant part. The

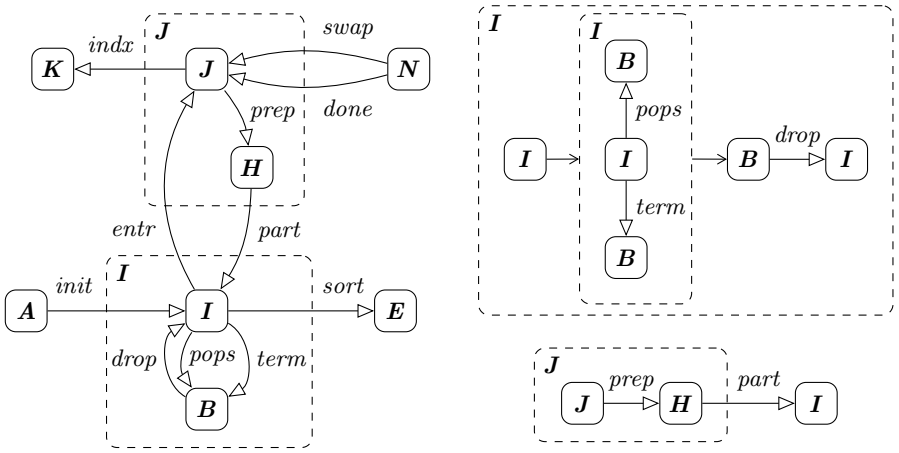


(b) Edge refinement of *loop*

Fig. 7: Fifth refinement

two events *done* and *swap* specify different values for the witnesses of the abstract parameters (of event *loop*). For example, *done* specifies $p = D \lhd id$ and *swap* specifies $p = (D \lhd id) \Leftarrow \{u \mapsto v, v \mapsto u\}$ giving a clue about how *loop* is implemented. Witnesses are a versatile feature of Event-B being applicable to verification techniques besides proof [9].

M6. In the sixth refinement the two nodes ***I*** and ***J*** are refined simultaneously demonstrating how sub-nodes of the refined nodes are to be connected. This is a data-refinement replacing the pointer to the top of the stack t by a new pointer s such that $t = s{+}1$, storing the top of the stack $m(t)$ and $n(t)$ in dedicated variables p and q, and finally, replacing the stack m and n by the "smaller" stack l and o. In the edge refinement diagram (Fig. 8(b)) we have collected



(a) Node refinements of ***I*** and ***J*** (b) Edge refinements of *drop* and *part*

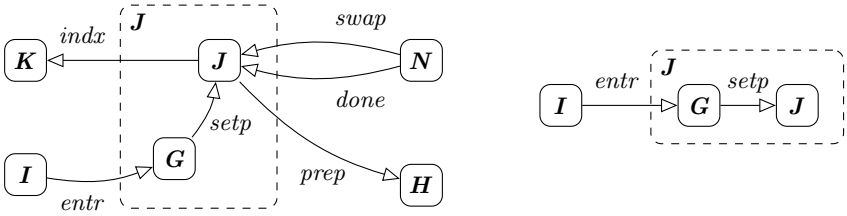
Fig. 8: Sixth refinement

two simultaneous edge refinements. The corresponding two simultaneous node refinements are shown in Fig. 8(a). In the edge refinement diagram we have to draw an additional super node —the inner super node ***I***— and connect it using anonymous edges. This is necessary because of restrictions on the shape of edge refinement diagrams that are imposed in order to be able to map such diagrams to customary control structures.

M7. The last refinement step introduces a new variable h to replace $b(\pi)$ in all event guards. In other words we add $h = b(\pi)$ to the nodes ***J***, ***K***, ***L*** and ***N***. The new event *setp* contains the assignment $h := b(\pi)$. Note how new events in the refinement diagrams are indicated by the nesting of the (super) nodes; compare Fig. 8 and Fig. 9 in this respect.

Closing remarks. We can carry out a series of data refinements to remove "synonyms" of variables. For instance, a, b and c by a variable h. This does not affect the structure of the diagrams. No further diagrams need to be drawn for these refinements.

We think the diagrams are easy to understand and manipulate. With their help, complex refinement steps using the Event-B refinement method are possible that would not be feasible in Event-B itself. Using multiple refinement steps



(a) Node refinement of ***J*** (b) Edge refinement of *entr*

Fig. 9: Seventh refinement

in Event-B does not always solve the problem. This is particularly important because in Event-B the ordering of the refinement steps influences the shape of the developed program. Using state machines its shape is specified and refinement is only concerned with structuring a complex correctness proof.

5 Design of a controller

The edge refinement diagrams in Section 4 are simpler than the corresponding node diagrams. The developed algorithmic structure is more discernible. Edge refinement was developed for this purpose and is therefore more specialised towards algorithm development than node refinement. This specialisation is achieved by imposing greater restrictions on the refinements that can be made. Lacking these restrictions, node refinement allows more flexibility in refinements. Node refinement is suited for the modelling and refinement of systems level models. It was developed for this purpose. In this section, we demonstrate the greater generality of node refinement by means of a model of a simple controller system which has mechanisms for responding and recovering from faults. The controller could not be developed using edge refinement. Although this example is simple and somewhat manufactured, it is intuitive and sufficient to illustrate the greater generality of node refinement. One can easily imagine that the model can be expanded in later refinements with similar patterns that would be impossible with edge refinement. Usually, there is a collection of informal requirements describing possible behaviours on which formal system modelling is based. Feedback from the formal model can then be used to improve the requirements: pointing to specification gaps and contradictions. However, for the present purpose we are not concerned with discussing requirements and do not refer to them explicitly. We also do not go into detail concerning the assertions and events that occur in the model.

The controller model. The initial abstract model of the controller (see Fig. 10) has three states: the power is off "***U***"; the power is on "***P***"; the power is on but the controlled is in a fault state "***F***". An edge labelled *pwr* models the power

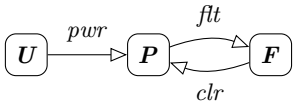


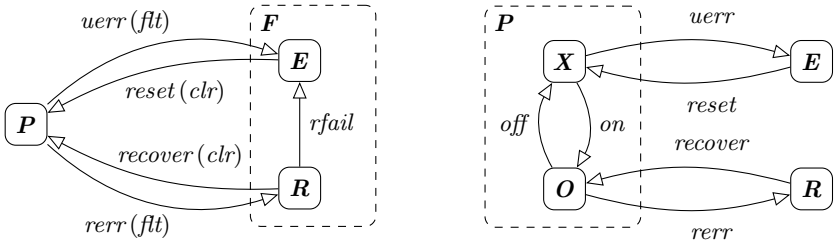
Fig. 10: Abstract controller model

being switched on and while the power is switched on, faults may occur *flt* and are subsequently cleared *clr*.

Firstly, the fault state "***F***" is refined to distinguish two sub-categories of fault (see Fig. 11(a)), ones that can be recovered from "***R***", and ones that require a reset "***E***". This enables the edge *flt* to be refined by spitting it into two edges *uerr* and *rerr* representing the two categories of fault. Similarly, *clr* is refined into *reset* and *recover* originating from their respective fault categories. Recovery may be unsuccessful resulting in a recoverable fault becoming transmuted into a resettable one by edge *rfail*.

The powered state "***P***" is then refined to distinguish two sub-modes of operation (see Fig. 11(b)). The control is switched off "***X***", and the control is switched on "***O***". Edges *on* and *off* form a loop allowing power to be cycled. This enables us to refine edges *uerr*, *reset*, *rerr* and *recover* so that recoverable errors originate and recover to the powered sub-state, "***O***", while unrecoverable ones originate and reset to the unpowered sub-state, "***X***".

The behaviour of the controller while being in one of the states ***P*** or ***F*** is more general than the patterns arrived at by edge refinement. If we were to implement a control program, we would introduce a dedicated variable to model the current operational state of the controller. This would obfuscate the model hiding the control structure in the program text. If we do not insist on program structure, state machines can concisely and clearly capture the behaviour.



(a) Node refinement of ***F*** (b) Node refinement of ***P***

Fig. 11: Refinements of the controller model

The refinement of node ***F*** splits the incoming and outgoing edges into cases that are revealed by the node refinement. This would not be possible using edge refinement.

The refinement of node ***P*** introduces a cycle between the states ***X*** and ***O***. If this was introduced via edge refinement it would require a loop at ***P*** in the abstraction. This would require prediction of later refinements in the abstract model which would be detrimental to its objective. The aim of abstract system modelling is to simplify the model in order to concentrate on important properties. Abstract models could become unnecessarily complex if stricter rules were imposed.

Allowing more general diagrams to be constructed supports forms of reasoning that would be difficult to achieve using the simpler algorithmic diagrams enforced by edge refinement. For instance, we may want to argue whether edge *rfail* is reasonable: is it reasonable for a supposedly recoverable error to result in a reset of the controller. The explicit modelling of the control states makes it possible to discuss such questions. This would not be possible if the control state was encoded by a program variable.

A combined refinement method. Using node refinement we can deal with more general architectural requirements. Edge refinement on the other hand provides only algorithmic structures that can be safely mapped on to (sequential) programs. A combined method would have the strengths of both. One could, for instance, develop the architecture of the controller using node refinement and implement the code at the edges using edge refinement. We have seen in Section 3 that the proof obligations of the methods could be easily mixed. We could simply consider every edge refinement to be a stylised node refinement allowing them to be mixed freely. Edge refinement can also be used to prove properties of deadlock-freedom [7]. Node refinement does not support this. The main difficulties are to achieve a clear refinement method and to avoid large complex proof obligations. Our next aim is to investigate deadlock-freedom properties of node refinement.

6 Conclusion

We have demonstrated the use of state machines for the formalisation of complex models based on Event-B. We have discussed two approaches to refinement that suggest themselves when modelling with state machines: node refinement and edge refinement. We have defined the two notions of refinement (based on Event-B refinement). Node and edge refinement have similar proof obligations. We have argued that, for the development of programs, they can be seen as providing two views of the same proof of correctness and refinement. However, node refinement is more general. It has been conceived for system-level modelling and it is not so obvious how to develop programs by this means alone. Edge refinement on the other hand has been conceived for program development, but is too restrictive to be used for system modelling. Combined use of both can address a large class of systems using node refinement for architectural modelling aspects and edge refinement for algorithmic aspects. We believe the two notions of refinement could be unified. However, care has to be taken to preserve the strong support of the two modelling aspects: architecture and algorithms. In this article we have

not discussed deadlock-freedom. For edge refinement it is obvious how properties of deadlock-freedom can be proved. For node refinement it is less clear how this can be done. We will still be looking for a method that is easy to apply. A unified method could transfer the concept of deadlock-freedom as dealt with by edge refinement to node refinement.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. CUP, 2010.
3. J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
4. K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2009.
5. R.-J. Back. Invariant based programming: basic approach and teaching experiences. *Formal Asp. Comput*, 21(3):227–244, 2009.
6. A. S. Fathabadi and M. Butler. Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *FMCO*, volume 6286 of *LNCS*, pages 89–104. Springer, 2009.
7. S. Hallerstede. Structured Event-B Models and Proofs. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *ABZ*, volume 5977 of *LNCS*, pages 273–286. Springer, 2010.
8. S. Hallerstede and M. Leuschel. Experiments in Program Verification using Event-B. *Formal Asp. Comput*, 2011. to appear.
9. S. Hallerstede, M. Leuschel, and D. Plagge. Refinement-Animation for Event-B – Towards a Method of Validation. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *ABZ 2010*, volume 5977 of *LNCS*, pages 287–301. Springer, 2010.
10. A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.
11. A. Knapp, S. Merz, and M. Wirsing. Refining Mobile UML State Machines. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *LNCS*, pages 274–288. Springer, 2004.
12. K. Lano and D. Clark. Semantics and Refinement of Behavior State Machines. In J. Cordeiro and J. Filipe, editors, *ICEIS 2008*, pages 42–49, 2008.
13. C. C. Morgan. *Programming from Specifications: 2nd Edition*. Prentice Hall, 1994.
14. M. Y. Said, M. J. Butler, and C. F. Snook. Language and tool support for class and state machine refinement in UML-B. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 579–595. Springer, 2009.
15. A. J. H. Simons. A theory of regression testing for behaviourally compatible object types. *Softw. Test, Verif. Reliab*, 16(3):133–156, 2006.
16. C. Snook and M. Waldén. Refinement of statemachines using event B semantics. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 171–185. Springer, 2007.
17. C. F. Snook and M. J. Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol*, 15(1):92–122, 2006.
18. Colin Snook. *Exploring the Barriers to Formal Specification*. PhD thesis, Electronics and Computer Science, University of Southampton, 2002.
19. N. Wirth. Program development by stepwise refinement. *CACM*, 14(4):221–227, April 1971.

Some Event-B specific symbols

$a \circ p$ denotes composition of a and b: $x \mapsto y \in a \circ p \Leftrightarrow (\exists z \cdot x \mapsto z \in p \land z \mapsto y \in a)$.
$t \lhd r$ denotes domain restriction of r by t: $x \mapsto y \in t \lhd r \Leftrightarrow x \in t \land x \mapsto y \in r$.
$t \lhd\!\!\!\!- r$ denotes domain subtraction of r by t: $x \mapsto y \in t \lhd\!\!\!\!- r \Leftrightarrow x \notin t \land x \mapsto y \in r$.
$s \lhd\!\!\!\!\!- r$ denotes relational override of s by r: $s \lhd\!\!\!\!\!- r \Leftrightarrow (\mathrm{dom}(r) \lhd s) \cup r$.