UseCase-wise Development: Retrenchment for Event-B

Richard Banach

School of Computer Science,
University of Manchester,
Manchester, M13 9PL, UK
banach@cs.man.ac.uk

Abstract. UseCase-wise Development, the introduction of functionality into an application in stages, with each stage being carried through to (ideally) implementation before the next is considered, is examined with a view to its being treated via an Event-B methodology. The need to modify top level behaviour in a non-skip way precludes its naive treatment via Event-B refinement, and paves the way for the use of retrenchment in Event-B. The details of an Event-B formulation of retrenchment, aligned to the practical details of the Rodin toolset, are described. The details of refinement/retrenchment interworking needed to handle UseCase-wise development are outlined, and a simple case study is given.

Key words: Event-B, UseCase-wise Development, Incremental Development, Refinement, Retrenchment, Tower Pattern

1 Introduction

One of the notable things about the move from traditional B [1] to Event-B [2, 3], is the way that the re-engineered refinement theory of Event-B has managed to encompass many 'low hanging fruit' issues, for the handling of which, retrenchment has been advanced in more conventional refinement frameworks in the past. One can mention: the introduction of new events at successive development levels (within certain restrictions); the emphasis on guards (rather than preconditions) and their strengthening during refinement; the migration of information between I/O variables and state variables (since in Event-B there is generally no separate category of I/O variables to worry about); and so on. All of this is beneficial, in bringing such issues under more rigorous control than when using other development techniques (or when using retrenchment).

Nevertheless, because in Event-B (as in every other rigorous refinement framework), the development strategy and the notion of correctness is fixed *ab initio* —and yet the world is richly and subtly structured— it is almost inevitable that sooner or later an application scenario will arise in which the demands of Event-B will prove to be a less than ideal fit for the application in question. It is to help accomodate situations like these that retrenchment was originally conceived, so it is natural to ask what retrenchment amounts to in the Event-B context, and how the notions of Event-B refinement and Event-B retrenchment would interact. Fortunately, since the original introduction of retrenchment [4], we have accumulated a good deal of experience and evidence on which to base the answer (see eg. [5]).

In this paper we examine retrenchment in the Event-B context, by looking at a small case study developed using a UseCase-wise development methodology. UseCase-wise development is our name for a development strategy in which increments of functionality are added in stages, with the introduction of each resulting in a usable application before the next is considered. Such an approach is at odds with the more traditional waterfall model with which typical formal development approaches are frequently aligned. We view the exploration of alternative strategies as a good motivation for studying how retrenchment should be formulated in Event-B, a question which is of independent interest in any case.

The rest of this paper is as follows. In Section 2 we describe UseCase-wise development, contrasting it with conventional Event-B development. Section 3 briefly reviews Event-B and discusses the details of retrenchment for Event-B. In Section 4 we cover retrenchment/refinement interworking and the Tower Pattern. The preceding ingredients are then applied to a small case study, illustrating a good fit between the UseCase-wise approach and Event-B correctness when retrenchment is available. Section 6 concludes.

2 UseCase-wise Development

In Event-B there is a strong emphasis on getting the requirements correct (or as near correct as is achievable) at the outset. One then analyses the requirements and determines the most appropriate order in which to take them into account within a sequence of refinements. The refinements themselves, mix the accretion of requirements issues as identified during requirements analysis, with data refinements, as appropriate. As the models get more detailed, sound decomposition techniques are available to split models into components, allowing further refinements to be done independently. This TopDown (TD) approach, proceeding as it does in an essentially linear manner, shows that the Event-B approach can be viewed as a formal interpretation of a fairly traditional waterfall strategy.

By UseCase-wise (UCw) development, we mean an approach to system development that proceeds by taking one or more of the UseCases identified during requirements analysis, and completes the development of those first, from the abstract models down to implementation, giving a usable system (with limited functionality). Subsequently further UseCases are incorporated, with all the elements of the development getting suitably enhanced, and yielding another working system, this time with greater functionality. The process is repeated until all the UseCases identified during requirements analysis have been developed, yielding a system with all the functionality desired. UCw development can be seen as a member of the ‘Agile Methods’ family of system development techniques.[1]

[1] We coined the term ‘UseCase-wise development’ to avoid confusion. It is enough to glance at http://en.wikipedia.org/wiki/Agile_software_development and the acronym blizzard one finds there, with the same term having different meanings in different settings, to realise what dangers lurk in the casual use of terms invented in this field. What we call UseCase-wise development is also called ‘incremental development’ in other places, but that term is so laden with possibilities for misinterpretation, that we thought it safest to invent a fresh name, inevitably causing yet more terminological proliferation.

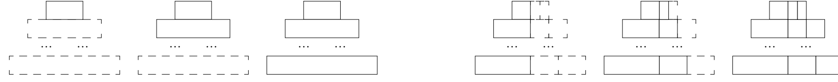

Fig. 1. Illustrating TopDown versus UseCase-wise development strategies.

Fig. 1 illustrates the TD versus UCw distinction. On the left we see a development proceeding TD in layers, while on the right, we see additional slices of functionality being added UCw to an initially developed system. It is important to realise that the TD vs. UCw distinction refers to the *dynamics* of the process by which the system is built. Even though a system may be built using a UCw process, one which is superficially unsympathetic to Event-B perspectives, there is no reason why the end result should not be a collection of models which enjoy the levels of mutual consistency characteristic of Event-B. Thus, even though one might argue that introducing a UCw approach into Event-B would be a retrograde step for Event-B, it is hard to dispute that introducing Event-B's criteria for correctness into the UCw approach would be a positive step for UCw development. This begs the question of how one might incorporate Event-B correctness into the UCw process. This will be dealt with in Section 4.

3 Event-B Machines, Refinement and Retrenchment for Event-B

In this section, we review Event-B machines, refinements, and against this background, we formulate retrenchment in a way that will permit the smoothest possible cooperation between the two techniques.

3.1 Event-B Machines

In a nutshell, an Event-B MACHINE has a *name*, it SEES one or more *static contexts*, and it owns some VARIABLES; these are allowed to be updated via EVENTS, but are required to always satisfy the INVARIANTS. The events can declare their own *parameters* (which are bound variables acting as carriers of input values) — each event has one or more *guards*, and one or more *actions* which are specified via *before-after predicates* (or notations such as assignment for simpler cases). Among the events there is an *INITIALISATION*, whose guard must be true.

The semantics of Event-B machines and of the refinement relationship between machines, is expressed via a number of proof obligations (POs). These must be provable in order for the machine or refinement in question to be well defined. We quote the main ones of interest to us, mentioning the others more briefly. See [2, 3] for full details.

For a machine A to be well defined the *initialisation* and *correctness* POs must hold:

$$Init_A(u') \Rightarrow I(u') \tag{1}$$

$$I(u) \wedge G_{Ev_A}(i,u) \wedge Ev_A(u,i,u') \Rightarrow I(u') \tag{2}$$

In (1), $Init_A$ is the initialisation event and (1) says that the value u' of A's state variable u established by $Init_A$ satisfies A's invariant I. Likewise, (2) says that for an event Ev_A of A, if A's invariant I, and Ev_A's guard $G_{Ev_A}(i,u)$, both hold in the before-state of the event, and Ev_A's before-after relation $Ev_A(u,i,u')$ also holds, then the after-state will satisfy the invariant I once more. In (1) and (2) we have suppressed mention of the details of the static contexts seen by A, but we have singled out Ev_A's input variables i for later convenience. For closer conformance to [2, 3] we have not mentioned any output variables, though it would be trivial to include them in the before-after relation $Ev_A(u,i,u')$ and in (2). Aside from (1) and (2), Event-B machines must satisfy *feasibility* POs for the initialisation and for all events, and also a *deadlock freedom* PO for non-terminating systems; see [2, 3].

3.2 Event-B Refinement

Suppose that as well as machine A, we have another machine C, with state variable w, input variable k, initialisation event $Init_C$, and typical event Ev_C, with guard $G_{Ev_C}(k,w)$ and before-after relation $Ev_C(w,k,w')$. If C is a refinement of A, its invariant $K(u,w)$ will be a relation over both u and w, and the counterparts of (1) and (2) are:

$$Init_C(w') \Rightarrow (\exists\, u' \bullet Init_A(u') \wedge K(u',w')) \tag{3}$$

$$\begin{aligned} & I(u) \wedge K(u,w) \wedge G_{Ev_C}(k,w) \wedge Ev_C(w,k,w') \\ & \quad \Rightarrow (\exists\, i,u' \bullet G_{Ev_A}(i,u) \wedge Ev_A(u,i,u') \wedge K(u',w')) \end{aligned} \tag{4}$$

where Ev_C is an event that is supposed to refine Ev_A and we have amalgamated the *guard strengthening* and *correctness* POs in (4) for later convenience. In (3), each $Init_C(w')$ intialisation must be witnessed by some $Init_A(u')$ intialisation that establishes the joint invariant $J(u',w')$. Likewise, (4) says that when both invariants hold, each $Ev_C(w,k,w')$ event is witnessed by some $Ev_A(u,i,u')$ event that re-establishes the joint invariant. Aside from (3) and (4) there are also *feasibility* POs for the initialisation and for all events, *variant decrease* POs for 'new' C events not declared to be refinements of any event of A, and also an overall *relative deadlock freedom* PO. See [2, 3] for full details.

We give a small example of Event-B refinement. It builds a directed graph from a finite universe of possible nodes contained in a set *NSet* held in a context *NCtx*.

Machine *Nodes* is concerned with the requirement of assigning nodes to the graph, picking them out of the set *NSet* using the event *AddNode*, starting with the empty set. Machine *Edges* refines Nodes, and addresses the requirement of having edges between some of the graph nodes. In typical Event-B fashion, it simply accumulates the new model elements, leaving the preceding ones unchanged. So *Edges* just contains *Nodes* in its body. The new requirement is handled by adding a new variable *edg* and a new event *AddEdge*. *AddEdge* acts like skip on the existing variable *nod*, as required for such 'new' events. Also since *AddEdge* does not refine any existing event (unlike *AddNode* which refines itself and is thus 'ordinary'), it must be 'convergent', which means that each invocation of *AddEdge* decreases the $\mathbb{N}$-valued VARIANT $\mathrm{card}(NSet \times NSet - edg)$, ensuring relative deadlock freedom. (We suppress the WHICH IS clauses below.)

```
MACHINE Nodes
SEES NCtx
VARIABLES nod
INVARIANTS
  nod ∈ ℙ(NSet)
EVENTS
  INITIALISATION
    WHICH IS ordinary
    BEGIN nod := ∅ END
  AddNode
    WHICH IS ordinary
    ANY n
    WHERE n ∈ NSet − nod
    THEN nod := nod ∪ {n}
    END
END
```

```
MACHINE Edges
REFINES Nodes
SEES NCtx
VARIABLES nod, edg
INVARIANTS
  nod ∈ ℙ(NSet)
  edg ∈ ℙ(NSet × NSet)
EVENTS
  INITIALISATION
    WHICH IS ordinary
    BEGIN nod := ∅ END
  AddNode
    WHICH IS ordinary
    REFINES AddNode
    ANY n
    WHERE n ∈ NSet − nod
    THEN nod := nod ∪ {n}
    END
  AddEdge
    WHICH IS convergent
    ANY n, m
    WHERE {n, m} ⊆ nod
      n ↦ m ∈ NSet × NSet − edg
    THEN edg := edg ∪ {n ↦ m}
    END
VARIANT card(NSet × NSet − edg)
END
```

3.3 Retrenchment for Event-B

We now formulate retrenchment for Event-B against the preceding background. The objective of retrenchment is to offer a flexible relationship between machines or system models that can capture situations in which all the detailed criteria of some species of refinement cannot be met, but where the two models in question are deemed nevertheless (and especially by domain experts rather than refinement specialists) to belong to the same development activity. The focus of retrenchment is on a simulation-like criterion, with the added aim of convenient interworking with refinement. Retrenchment is therefore formulated as a modification of the main POs of the refinement notion, with the incorporation of suitable additional predicates to enhance expressivity.[2]

For the specific context of Event-B, retrenchment is a relationship that is to hold between top level machines. When a retrenchment involving a refinement machine is needed, one must quantify away the dependence on the higher level abstractions to get a self-contained top level machine using the technique described in Chapter 11 of [1].

Unlike refinement in Event-B, in which the refinement data (essentially just the joint invariant and some bookkeeping details, as in our example) are incorporated into the syntax of the refining machine, retrenchment is an independent syntactic construct, as befits the weaker relationship between machines that it expresses, and especially, the desire that none of the details of retrenchment interfere in any way with any refinement that any machine involved in a retrenchment might also be involved in. Notationally this departs from the scheme in [4] and agrees with the line taken in [6–8].

[2] Thus the modification of the relevant refinement POs constitutes the sense in which the *simulation-like criterion* is intended; suitable pairs of transitions in the two models should satisfy an appropriate generalisation of (4).

Suppose we have top level machines A (having the elements mentioned earlier) and B, and B's state and input variables are v, j, the invariant is $J(v)$ and the other pieces can be imagined. Here is a schematic syntax for the retrenchment construct, intended as a good fit for Event-B as currently implemented in the Rodin toolset [9]:

```
RETRENCHMENT Identifier Ret_{A,B}
FROM Identifier A TO Identifier B
[ SEES IdentifierList ]
[ RETRIEVES Predicate R(u, v) ]
[ EVENTS
   [ RAMIFICATIONS Identifier Ev_A [ TO Identifier Ev_B ]
      [ WITHIN Predicate W_{Ev_A,Ev_B}(i, j, u, v) ]
      [ CONCEDES Predicate C_{Ev_A,Ev_B}(u', v', i, j, u, v) ]
      END
   ]+
]
END
```

The construct has a name $Ret_{A,B}$, and is FROM machine A TO machine B. It can SEE static contexts as can a machine or refinement. There is a RETRIEVES relation $R(u, v)$ between the two state spaces, and for each pair of retrenchment-related events in A and B, eg. Ev_A and Ev_B (where one can omit mentioning Ev_B if it has the same name as Ev_A), there are the RAMIFICATIONS, consisting of the WITHIN relation $W_{Ev_A,Ev_B}(i, j, u, v)$ and the CONCEDES relation $C_{Ev_A,Ev_B}(u', v', i, j, u, v)$.

The semantics of retrenchment is given by its POs. These are:

$$Init_B(v') \Rightarrow (\exists\, u' \bullet Init_A(u') \wedge R(u', v')) \tag{5}$$

$$I(u) \wedge R(u, v) \wedge J(v) \wedge W_{Ev_A,Ev_B}(i, j, u, v) \wedge Ev_B(v, j, v') \\ \Rightarrow (\exists\, i, u' \bullet Ev_A(u, i, u') \wedge (R(u', v') \vee C_{Ev_A,Ev_B}(u', v', i, j, u, v))) \tag{6}$$

where there is an instance of (6) for each ramifications-related pair Ev_A and Ev_B. We see that the intialisation PO is standard, while the correctness PO permits considerable deviation from refinement-like behaviour by virtue of the presence of the within and concedes relations. In addition to the above, we demand for each Ev_A/Ev_B pair that:

$$W_{Ev_A,Ev_B}(i, j, u, v) \Rightarrow G_{Ev_A}(i, u) \wedge G_{Ev_B}(j, v) \tag{7}$$

which is the criterion that ensures smooth retrenchment/refinement interworking. Note however, that the other POs of Event-B refinement, variant decrease and relative deadlock freedom, do not have counterparts in retrenchment; we want to be able to relate machines with significantly different behaviour as regards these aspects, if desired.

Although we do not have space here to fully examine the arguments why the above design is a good one for retrenchment, we can make the following remarks. Firstly, the aim of a notion that *departs from* refinement or *desires to accommodate inability to satisfy* refinement, must amount to a weakening of refinement — there is clearly no point in doing the opposite. The proposal we have given above does this, since the occurrences of $W_{Ev_A,Ev_B}(i, j, u, v)$ and $C_{Ev_A,Ev_B}(u', v', i, j, u, v)$, in the hypotheses and conclusions respectively, of (6), clearly weaken (4). Secondly, we want this weakening to be as general as possible so as not to have to invent a different notion of non-refinement for every conceivable departure from refinement that might arise. Again, (6) achieves this since $W_{Ev_A,Ev_B}(i, j, u, v)$ and $C_{Ev_A,Ev_B}(u', v', i, j, u, v)$ must be specified on a

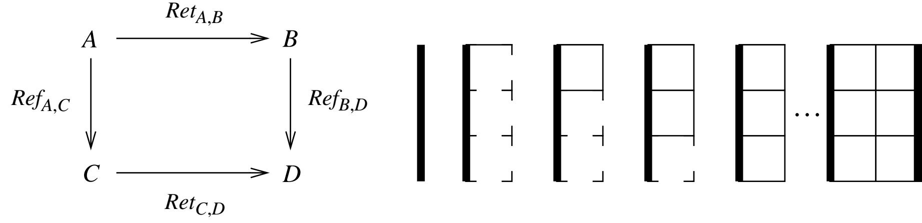

Fig. 2. The basic structure for the Tower Pattern on the left, and on the right, its use in constructing a UCw development whose outcome enjoys the rigour of an Event-B development.

per-event-pair basis. Thirdly, we would want the departure from refinement to be quantified in some way. Again, (6) achieves this, at least indirectly, since $W_{Ev_A,Ev_B}(i,j,u,v)$ and $C_{Ev_A,Ev_B}(u',v',i,j,u,v)$ must actually be specified by the user in each particular case of retrenchment — in doing this the precise details of how refinement fails to hold is made clear by the user in the details of these (otherwise unconstrained) relations. Lastly, we would want good interworking with refinement. This important topic is the subject of the next section. See [5] for more extensive discussion of generalities such as these concerning retrenchment.

4 Retrenchment and Event-B Interworking: the Tower Pattern

A definite *sine qua non* of retrenchment is that the use of retrenchment should not spoil the rigour achievable via refinement. The best results are obtained when the two notions work closely together, with retrenchment being used to connect together otherwise incompatible refinement strands. The more tightly such different refinement strands are coupled via retrenchment, the more restraint is exercised over retrenchment's otherwise extreme permissiveness.

The paradigmatic arrangement of retrenchments and refinements, that achieves both the tight coupling that restricts retrenchment and the non-interference with the rigour of refinement, is the *Tower Pattern*, an epithet that summarises a host of square completion and factorisation results in Jeske's thesis [10]. The left hand side of Fig. 2 shows a commuting square of retrenchments and refinements among four system models A, B, C, D, with the retrenchments horizontal and the refinements vertical (and the data that characterises these retrenchments and refinements implicit). The results of [10] show that whenever you start with two adjacent sides of such a square, the square can be completed by building the missing system model and its impinging retrenchment and refinement out of the existing elements in a canonical way, and that the result is indeed a commuting square. (Section 5 is concerned with an explicit example of this construction.) Such commuting squares are the fundamental building blocks of the tower, which itself is just an arrangement of such squares into a suitable grid pattern appropriate to the development at hand.

The tower construction has by now had substantial vindication. In the formal development of the Mondex Electronic Purse [11], there were a number of requirements issues that were handled less than ideally in the formal modelling. These have all been handled convincingly via retrenchment, mostly using the tower [12–14].

Although [10] was done in the context of Z refinement to directly serve the needs of the Mondex work, the approach advocated in Section 3 and discussed more widely in [5], ensures that there is a wide commonality between retrenchment formulations for different variants of refinement. The insistence that retrenchment is confined to the initialisation and correctness POs helps here, since most notions of refinement have initialisation and correctness POs that are either identical to, or extremely close to, (5) and (6). In particular, this applies to Event-B and Z refinements, so, since the composition of refinements and retrenchments is defined via their initialisation and correctness POs, these compositions (which yield retrenchments), will be identically defined for both Z and Event-B. See [15].

For our purposes, we need a suitable analogue of the Postjoin Theorem from [10], which states that if we have three systems A, B, C, as in Fig. 2, the square can be completed in a canonical way with a system D, and a connecting retrenchment $Ret_{C,D}$ and refinement $Ref_{B,D}$, so that the two retrenchment+refinement compositions round it are equal. What impedes the immediate application of the theorem from [10] is: (a) its ferocious technical complexity; (b) its detailed Z dependence as regards 'non-correctness' POs (i.e. the POs that in Z replace guard strengthening and relative deadlock freedom). Fortunately the same remedy overcomes both problems. The ferocity of the postjoin construction comes from wanting to deal with the most general situation possible, which means allowing the relations that comprise the given retrenchment and refinement to be as unrestricted as possible. The postjoin construction then has to extract those parts of these constituent entities that compose smoothly, and it attempts to do so in the most general manner achievable — this generates formidable complexity. However, if we are dealing with a situation in which the constituents are well behaved to start with, most of the complexity simplifies drastically, and a situation that is 'obviously sensible' emerges. The same applies to the non-correctness POs; in a well behaved context, these do not cause problems either. Further discussion of these points is best given in the context of an example, so we pick up this thread again near the end of Section 5.

What does any of this have to do with reconciling the TD and UCw strategies? Well, a single step of the UCw strategy takes the pre-existing development, and incorporates a new UseCase of functionality. We can imagine that the pre-existing development has been captured within a sequence of Event-B refinements, starting with the most abstract formulation of the pre-existing functionality, and descending into more concrete levels of description, perhaps aggregating additional events into the description as we go in the usual Event-B way. We can represent this pre-existing development by the thick vertical line in the middle of Fig. 2.

The incorporation of the new functionality may well require the introduction of new events at the top level, a reworking of the top level invariant, reworked top level guards, and so on. As such it will not generally fit into the preceding refinement sequence, not least because the new top level functionality will usually not manipulate the top level state in a skip-like fashion. (These of course are the crucial reasons why one cannot, in

general, capture such increments of functionality using Event-B refinements.) However, the new functionality can be related to the existing development via a retrenchment.

(We can say the latter with confidence since we show in [5] that *any* two system models can be related via a retrenchment, the potential vacuousness of such a statement being alleviated by the observation that the various relations that comprise a retrenchment help to *quantify* the difference between the models, as noted above. In a well-controlled situation, such as the introduction of new functionality, the difference between the two models will not be capriciously arbitrary (despite not necessarily conforming to Event-B refinement desiderata), and so the retrenchment between them will, in fact, be able to say quite a lot.)

Depicting the retrenchment from previous top level model to new top level model horizontally, we arrive at the Γ shape given by the solid part of the next piece of Fig. 2.

Now the tower construction can take over, and complete a sequence of refinements from the new top level via the requisite sequence of postjoin square completions, working downwards, as illustrated in the next part of Fig. 2. The new bottom level will be at the right level of abstraction to correspond with the pre-existing bottom level model. Thus one bout of UseCase introduction has been achieved via the tower. Successive bouts follow the same route. In each case we draw up the retrenchment that takes us to the new top level model, and allow the tower to do the rest. Finally, the right hand column of the last bout yields a pristine Event-B development of the full functionality, shown as an even thicker line on the right of Fig. 2.

5 A Simple Case Study

We tackle a toy distributed allocation problem. It is carried out in the way done here only for purposes of illustrating our techniques. In reality, one would only apply the machinery discussed in this paper to significantly more substantial examples.

Elements are to be allocated. At the most abstract level, there is a large (potentially infinite) set, *ASet*, whose elements are to be allocated, and at a low level this is replaced by a much smaller finite subset *DSet*. Also, at low enough levels of abstraction, *ASet* and *DSet* are statically partitioned into $ASet1, ASet2$ and $DSet1, DSet2$ (the latter being subsets of the former) for allocation to two individual agents. These static facts are captured in the context *Ctx*:

CONTEXT *Ctx*
SETS $ASet, DSet, ASet1, ASet2, DSet1, DSet2$
AXIOMS
  $axm1 : ASet1 \cup ASet2 = ASet$
  $axm2 : ASet1 \cap ASet2 = \varnothing$
  $axm3 : DSet \subset ASet$
  $axm4 : DSet1 = DSet \cap ASet1$
  $axm5 : DSet2 = DSet \cap ASet2$
END

5.1 Four Machines

Below are four machines, *A*, *B*, *C*, *D*, deliberately arranged as in Fig. 2. The left hand column treats only one UseCase, that of allocation. Machine *A*, the most abstract one,

simply models the allocation of an element from *ASet* at the global level. Machine *A* is refined to machine *C*, in which two agents can allocate from their statically assigned partitions, each agent allocation refining the global allocation event.

```
MACHINE A
SEES Ctx
VARIABLES x
INVARIANTS inv1 : x ∈ ℙ(ASet)
EVENTS
  INITIALISATION
    BEGIN act1 : x := ∅ END
  AddEl
    ANY el
    WHERE grd1 : el ∈ ASet − x
    THEN act1 : x := x ∪ {el}
    END
END
```

```
MACHINE B
SEES Ctx
VARIABLES y
INVARIANTS inv1 : y ∈ ℙ(ASet)
EVENTS
  INITIALISATION
    BEGIN act1 : y := ∅ END
  AddEl
    ANY el
    WHERE grd1 : el ∈ ASet − y
    THEN act1 : y := y ∪ {el}
    END
  SubEl
    ANY el
    WHERE grd1 : y ≠ ∅
          grd2 : el ∈ y
    THEN act1 : y := y − {el}
    END
END
```

```
MACHINE C
REFINES A
SEES Ctx
VARIABLES x1, x2
INVARIANTS inv1 : x1 ∈ ℙ(ASet1)
           inv2 : x2 ∈ ℙ(ASet2)
           inv3 : x = x1 ∪ x2
EVENTS
  INITIALISATION
    BEGIN act1 : x1 := ∅
          act2 : x2 := ∅
    END
  AddEl1
    REFINES AddEl
    ANY el
    WHERE grd1 : el ∈ ASet1 − x1
    THEN act1 : x1 := x1 ∪ {el}
    END
  AddEl2
    REFINES AddEl
    ANY el
    WHERE grd1 : el ∈ ASet2 − x2
    THEN act1 : x2 := x2 ∪ {el}
    END
END
```

```
MACHINE D
REFINES B
SEES Ctx
VARIABLES y1, y2
INVARIANTS inv1 : y1 ∈ ℙ(DSet1)
           inv2 : y2 ∈ ℙ(DSet2)
           inv3 : y = y1 ∪ y2
EVENTS
  INITIALISATION
    BEGIN act1 : y1 := ∅
          act2 : y2 := ∅
    END
  AddEl1
    REFINES AddEl
    ANY el
    WHERE grd1 : el ∈ DSet1 − y1
    THEN act1 : y1 := y1 ∪ {el}
    END
  AddEl2
    ... ...
  SubEl1
    REFINES SubEl
    ANY el
    WHERE grd1 : y1 ≠ ∅
          grd2 : el ∈ y1
    THEN act1 : y1 := y1 − {el}
    END
  SubEl2
    ... ...
END
```

The right hand column introduces the deallocation UseCase. Machine *B* is like machine *A*, except that (aside from variable renaming for clarity) it has a *SubEl* event as well as an *AddEl* one. Machine *B* is refined to machine *D*. In machine *D*, the allocation and deallocation events are refined into their agent-wise counterparts (the ones for agent 2 are just like the ones for agent 1, and so are suppressed to save space). Also machine *D* introduces the use of *DSet* and its partition into $DSet1, DSet2$.

Let us consider the relationships between these various machines. The *A* to *C* refinement is a normal Event-B refinement, as is the *B* to *D* refinement. However there is a

difference between the two. In the A to C refinement, the static set $ASet$ stays the same, whereas in the B to D refinement, we are able to replace $ASet$ by $DSet$. The reason we are able to do this in the case of the B to D refinement but not the A to C refinement is connected with the details of Event-B refinement POs. One of these, the relative deadlock freedom PO, demands that the disjunction of the guards of all the abstract events implies the disjunction of the guards of all the concrete ones. Consider then the state in which all $DSet$ elements have been allocated. If we used $DSet$ instead of $ASet$ in machine C, then, whereas the machine A $AddEl$'s guard would be true (since there are plenty of elements left in $ASet - DSet$) the disjunction of the machine C $AddEl1$ and $AddEl2$ guards would be false (since by definition, $(DSet1-x1)\cup(DSet2-x2)$ is empty in this state). So the disjunction of the abstract guards would not imply the disjunction of the concrete ones, and the refinement would fail. The same is not true of the B to D refinement. There, when all the $DSet$ elements have been allocated, the disjunction of the abstract guards is true as before, but now, at the concrete level, even though $AddEl1$ and $AddEl2$ are disabled as in machine C, we have the $SubEl1$ and $SubEl2$ events enabled, so the disjunction of the concrete guards is true as well, and the refinement succeeds.[3]

The relationship from machine A to machine B cannot be an Event-B refinement since machine B's $SubEl$ event manipulates the machine A state in a non-skip manner (and furthermore, the relationship cannot be a converse Event-B refinement since then machine B's $SubEl$ event would not be refined by anything). To capture this relationship we need retrenchment, and the trivial retrenchment $Ret_{A,B}$ that follows will do:[4]

```
RETRENCHMENT Ret_{A,B}
FROM A TO B
SEES Ctx
RETRIEVES ret1 : x = y
EVENTS
  RAMIFICATIONS AddEl
    WITHIN wth1 : true
    CONCEDES con1 : false
    END
END
```

```
RETRENCHMENT Ret_{C,D}
FROM C TO D
SEES Ctx
RETRIEVES ret1 : x1 = y1
          ret2 : x2 = y2
EVENTS
  RAMIFICATIONS AddEl1
    WITHIN wth1 : true
    CONCEDES con1 : false
    END
  RAMIFICATIONS AddEl2
    ... ...
END
```

Alongside $Ret_{A,B}$, we have $Ret_{C,D}$, the retrenchment required to relate machine C to machine D. Note that neither retrenchment needs to say anything about the initialisation events, since they are required to work just as in refinement. $Ret_{C,D}$ looks just as trivial as $Ret_{A,B}$ but in fact it is less so. In the Rodin toolset, there is a convention that when one event refines another, any parameters that are identically named in the two events are in fact equal, and the relevant equalities are automatically factored in to the automated reasoning. We have availed ourselves of a similar convention for retrenchments, and it applies in both $Ret_{A,B}$ and $Ret_{C,D}$. In $Ret_{A,B}$ this has little impact, since the only place where it applies (the parameters of the machine A and machine B $AddEl$ events), the as-

[3] The success can be attributed to the fact that we are using the *weak* relative deadlock freedom PO rather than the strong one (see [3], Deliverable D3). The strong version demands that for *each* abstract event, its guard implies the disjunction of the corresponding concrete guard with all the 'new event' guards. Such a PO would fail here, a circumstance that could be overcome with a more extensive use of retrenchment.

[4] One could introduce syntax to deal with such trivial event retrenchments more succinctly.

sumptions pertaining to the two events' parameters are identical. In $Ret_{C,D}$ however, the same situation is less trivial, since machine C's *AddEl*1 *el* is selected from *ASet* while machine D's *AddEl*1 *el* is selected from *DSet*. If we temporarily rename the parameters in these two events by adding subscripts, the real within relation between the *AddEl*1 events in $Ret_{C,D}$ becomes:

$$el_C = el_D \wedge el_C \in ASet \wedge el_D \in DSet \tag{8}$$

which enforces an additional constraint on el_C. So, despite appearances, the within relation of *AddEl*1 has some real work to do. (Note that a similar thing is silently accomplished in the course of the B to D refinement. And if we had taken name identity even further, and avoided renaming the A/C variables $x, x1, x2$, to the B/D variables $y, y1, y2$, we could have simplified $Ret_{A,B}$ and $Ret_{C,D}$ even more by trivialising the retrieve relations.)

5.2 *A*, *B*, *C*, *D* and the Tower

Machines A, B, C, D, (and the various retrenchments and refinements that relate them), form a commuting square and an instance of the Postjoin Theorem. It is time to pick up the discussion left over from Section 4 regarding this. If we follow a state element from A through the A to B retrieve relation and then through the B to D joint invariant, we arrive at the same set of possibilities as if we had first gone through the A to C joint invariant and then the C to D retrieve relation, i.e. the relevant relational compositions are equal (a claim easy enough to check by hand in this simple example), and they constitute the retrieve relation for the composed retrenchment. The rest depends on the events. Of these, the initialisations behave straightforwardly of course; assuming the truth of the component initialisation POs enables the truth of the composed initialisation PO to be proved, given the composed retrieve relation.

For the other events, we note that machine A's *AddEl* event is going to be retrenched to both *AddEl*1 and *AddEl*2 in machine D, by tracing the square via B or C. Since *AddEl*1 and *AddEl*2 are so similar, it will be sufficient for us to discuss *AddEl*1 and to leave *AddEl*2 to the reader. To discuss *AddEl*1, we first need the within relation for *AddEl* and *AddEl*1. This can be obtained in one of two ways. One can compose the within relation of the A to B retrenchment with the conjunction of the joint invariant and WITNESS relations[5] of the B to D refinement, or one can compose the joint invariant and WITNESS relations of the A to C refinement with the within relation of the C to D retrenchment. Since the square commutes, these two calculations agree, as they must, and as the reader can check.

The concedes relation for *AddEl* and *AddEl*1 is determined similarly. One way round, the concedes relation of the A to B retrenchment is composed with the joint invariant and witness relations of the B to D refinement for the before-state and input

[5] In Rodin, when an event and its refinement have different parameters, the refined event has a WITNESS clause to say how any abstract parameters not occurring in the refinement are to be related to the refined ones. This is like the within relation of a retrenchment and goes beyond what is documented in [3] Deliverable D3. See the Rodin User Manual at [9]. When there are no such abstract parameters, the witness relation trivialises.

parameters, and another copy of the B to D refinement joint invariant is used for the after-state. The other way round, the witness relation and two copies of the joint invariant of the A to C refinement are composed with the concedes relation of the C to D retrenchment. Again, either way round the square yields the same result. See [15] for more detailed calculations and proofs regarding the general case.

Altogether, we get the composed retrenchment $Ret_{A,D}$, in which the familiar facts hold for the common *el* parameter of *AddEl* and *AddEl*1:

```
RETRENCHMENT Ret_{A,D}
FROM A TO D
SEES Ctx
RETRIEVES ret1 : x = y1 ∪ y2
EVENTS
  RAMIFICATIONS AddEl TO AddEl1
    WITHIN wth1 : true
    CONCEDES con1 : false
    END
  RAMIFICATIONS AddEl TO AddEl2
    ... ...
END
```

The above sketches a confirmation that machine D (which we pulled out of a hat) has the right characteristics to be the desired square completion. In general, when machines are constructed to solve plausible problems, their interrelationships are benign, and it is normally transparent what the square completion should look like, without resorting to the general theory. Benign situations are characterised by the fact that the state (and other) spaces partition into equivalence classes, which the various relations in play treat in an 'all or nothing' manner. In other words, the relations involved and their needed compositions are all *regular* [16]. In such cases one can confidently eschew the forbidding complexity of the results in [10], or as here, their Event-B analogues, and work by hand.

6 Conclusions

Assuming that bringing the correctness achievable using techniques like Event-B to today's 'Agile Methods' would be a good thing, we argued that, in general, Event-B's insistence that levels of abstraction be complete at the point of introduction blocks its ready deployment in such methodologies. Thus, in the specific UCw approach, one might want to introduce new top level events that manipulate the state in non-skip ways, and perhaps to make even more drastic modifications. We then showed that retrenchment, which we reformulated in a manner suitable for Event-B, and for Rodin, could address such situations via the *Tower Pattern*, which we illustrated 'by hand'.

The same technical constructions also enlarge the scope for Event-B to tackle a wider variety of 'real-world' applications. For example, Event-B's insistence that all data types are discrete (at least) inhibits its application in real-world scenarios in which the intrinsic variable types are continuous. Of course in all such cases, the continuous variables must eventually be reduced to discrete ones in order to implement digital controllers, but carrying out the argument to justify this replacement within a retrenchment context allows it to make real contact with the formal development, whereas otherwise, it would have to be expelled completely from the formal considerations. Other ways in

which retrenchment might capture the 'grey areas' surrounding a formal development using Event-B could be easily imagined.

It would of course be desirable to mechanise the technology introduced here. For this, as well as the obvious tool development, it would be necessary to formulate an Event-B version of the theorems of [10]. This would need to focus on the useful cases of the tower in a manner that allowed for ready mechanisation of the whole square completion process. These aspects remain as work for the future.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Event-B. to be published.
3. Rodin. European Project Rodin (Rigorous Open Development for Complex Systems) IST-511599 http://rodin.cs.ncl.ac.uk/.
4. Banach, R., Poppleton, M.: Retrenchment: An Engineering Variation on Refinement. In Bert, D., ed.: 2nd International B Conference. Volume 1393 of LNCS., Montpellier, France, Springer (April 1998) 129–147
5. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. **67** (2007) 301–329
6. Banach, R., Fraser, S.: Retrenchment and the BToolkit. In: ZB 2005: Formal Specification and Development in Z and B. Volume 3455 of LNCS., Springer (2005) 203–221
7. Fraser, S., Banach, R.: Configurable Proof Obligations in the Frog Toolkit. In: Proc. Fifth IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society Press, IEEE (2007) 361–370
8. Fraser, S.: Mechanized Support for Retrenchment. PhD thesis, School of Computer Science, University of Manchester (2008)
9. The Rodin Platform. http://sourceforge.net/projects/rodin-b-sharp/.
10. Jeske, C.: Algebraic Integration of Retrenchment and Refinement. PhD thesis, University of Manchester (2005)
11. Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory (2000)
12. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Retrenching the Purse: Finite Sequence Numbers, and the Tower Pattern. In: Proc. FM-06, LNCS. (2005) 382–398
13. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: Finite Exception Logs, and Validating the Small. In: Proc. IEEE/NASA SEW30-06. (2005) 234–245
14. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties. In: Proc. IEEE ISOLA-06. (2006) to appear.
15. Banach, R., Jeske, C., Poppleton, M.: Composition Mechanisms for Retrenchment. J. Log. Alg. Prog. **75** (2008) 209–229
16. Banach, R.: On Regularity in Software Design. Sci. Comp. Prog. **24** (1995) 221–248