Under consideration for publication in Formal Aspects of Computing

Retrenchment for Event-B: UseCase-wise Development and Rodin Integration

Richard Banach[1]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract. UseCase-wise Development, an 'Agile Method' which introduces functionality into an application stage by stage, with each stage being carried through (ideally) to 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 an Event-B context. An Event-B formulation of retrenchment aligned to the practicalities of the Rodin toolset is described. The details of refinement/retrenchment interworking needed to handle UseCase-wise development are outlined, and three small case studies are discussed. The details of the integration of the retrenchment proposal into Rodin are outlined.

Keywords: Event-B, UseCase-wise Development, Incremental Development, Refinement, Retrenchment, Tower Pattern, Rodin Toolset.

1. Introduction

One of the notable things about the move from traditional B [Abr96] to the more recent Event-B [Abr,Roda], 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 indeed 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. 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

Correspondence and offprint requests to: Richard Banach, School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, U.K. email: banach@cs.man.ac.uk

would interact. Fortunately, since the original introduction of retrenchment [BP98], we have accumulated a good deal of experience and evidence on which to base the answer (see eg. [BPJS07a]).

In this paper we examine retrenchment in the Event-B context, by looking at a couple of case studies 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 development strategies from the formal perspective as a good thing, since it improves alignment with human intuition. In this instance, it also motivates the formulation of retrenchment for 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 in Section 5 — this case study also provides a good vehicle for comparison with the anticipating event technique of [ACM05]. Section 6 discusses a case study based on trains, and Section 7 considers a small telephony case study. All of these show a good fit between the UseCase-wise approach and the Event-B notion of correctness when retrenchment is available. Section 8 examines the issue of incorporating Rodin tool support for retrenchment and the tower in some detail. Section 9 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]

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.

[1] We coined the term 'UseCase-wise development' in this paper to try to avoid confusion. It is enough to cast a glance at sites such as 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.

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) — and 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 [Abr, Roda] 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 $I(u)$, 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 [Abr, Roda] 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 [Abr, Roda].

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, this reflecting the fact that in the B-Method generally, the view is emphasised that a refinement is seen as (and is therefore syntactically described as), an enhancement of the abstraction towards implementation, rather than as an independent entity. The counterparts of (1) and (2) become:

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

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

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 [Abr, Roda] 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* below, 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
  inv1 : nod ∈ ℙ(NSet)
EVENTS
  INITIALISATION
    WHICH IS ordinary
    BEGIN act1 : 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
  inv1 : nod ∈ ℙ(NSet)
  inv2 : edg ∈ ℙ(NSet × NSet)
EVENTS
  INITIALISATION
    WHICH IS ordinary
    BEGIN act1 : 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 ∈ 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 [Abr96].

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 [BP98] and agrees with the line taken in [BF05, FB07, Fra08].

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 [Rodb]:

[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).

```
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) ]
        [ OUTPUT Predicate O_{Ev_A,Ev_B}(u', v', 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 $W_{Ev_A,Ev_B}(i,j,u,v)$, the OUTPUT relation $O_{Ev_A,Ev_B}(u',v',i,j,u,v)$ 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:

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

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

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, output 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 called the tower compatibility criterion, and which ensures that retrenchment only engages with well defined transitions, and thereby interworks smoothly with refinement. Note however, that the other POs of Event-B refinement, i.e. 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 the requirements arena creates the perception that it is desirable to do so.

Although this is not an appropriate place to examine in depth the arguments regarding why the above design is a good one for retrenchment, we can summarise the main issues as follows. 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 $W_{Ev_A,Ev_B}(i,j,u,v)$ in the hypotheses, and of $O_{Ev_A,Ev_B}(u',v',i,j,u,v)$ and $C_{Ev_A,Ev_B}(u',v',i,j,u,v)$, in the 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 $W_{Ev_A,Ev_B}(i,j,u,v)$, $O_{Ev_A,Ev_B}(u',v',i,j,u,v)$ and $C_{Ev_A,Ev_B}(u',v',i,j,u,v)$ must be specified on a 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 $W_{Ev_A,Ev_B}(i,j,u,v)$, $O_{Ev_A,Ev_B}(u',v',i,j,u,v)$ 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, which ensures that retrenchment can add to rather than detract from what can be done with refinement, is the subject of the next section. See [BPJS07a] 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.

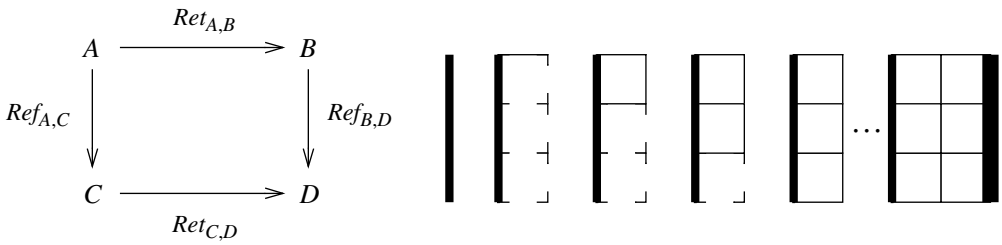

Fig. 2. The basic structure for the Tower Pattern on the left, and on the right, its use in constructing a UCw development whose outcome enjoys the rigour of an Event-B development.

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 that were first studied thoroughly in Jeske's thesis [Jes05], and which were more recently reformulated, revised and generalised in [BJ]. The left hand side of Fig. 2 shows a 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). In [Jes05], the square commutes 'on the nose' in that the two paths round the square from A to D (given by composing the A to B retrenchment with the B to D refinement on the one hand, and on the other, by composing the A to C refinement with the C to D retrenchment) yield the same retrenchment from A to D. In [BJ] this is relaxed a little, and in the general case, the two paths each describe a portion of a larger retrenchment from A to D. Either way, the results of [Jes05, BJ] 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 square that commutes in the appropriate manner — the case studies in Sections 5-7 are concerned with explicit examples of such constructions. Such squares are the fundamental building blocks of the tower, which itself is just an arrangement of such squares into a suitable grid pattern as suits the development at hand.

The tower construction has by now had substantial vindication. In the formal development of the Mondex Electronic Purse [SCW00], 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 [BPJS05, BJPS05, BJPS06].

Although [Jes05] was done in the context of Z refinement to directly serve the needs of the Mondex work, the approach advocated in Section 3, discussed more widely in [BPJS07a], and explored in detail in [Ban], ensures that there is a wide commonality between retrenchment formulations for different variants of refinement. This commonality is also behind the reformulated and generalised approach of [BJ]. 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 [BJP08].

For our purposes, we need a suitable analogue of the Postjoin Theorem from [Jes05, BJ], 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 (for [Jes05]) or compatible (for [BJ]).

Direct application of the Postjoin Theorem from [Jes05] is impeded by two things: (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). The situation as regards the Postjoin Theorem from [BJ] is considerably more pleasant: there, the complexities are confined to details of the characterisation of universality of the main construction, which is itself relatively straightforward. Fortunately, in the context of a simple and natural example, in which the constituents are well behaved to start with, most of the complexity simplifies drastically, and a situation that is 'obviously sensible' emerges. 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 [BPJS07a] 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 ⌐ 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 detail (regarding which, see Section 5.2). In reality, one would only apply the machinery discussed in this paper to significantly more substantial examples.

Resources, modelled as elements of a set, are to be allocated to users. 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 $DSet1, DSet2$ respectively, with $DSet1$ a subset of $ASet1$ and $DSet2$ a subset of $ASet2$, ready for allocation to two individual agents. These static facts are captured in the context Ctx:

CONTEXT Ctx
SETS $GSet$
CONSTANTS $ASet, DSet, ASet1, ASet2, DSet1, DSet2, a$
AXIOMS
$axm1 : ASet \subseteq GSet \wedge ASet1 \subseteq GSet \wedge ASet2 \subseteq GSet$
$axm2 : DSet \subseteq GSet \wedge DSet1 \subseteq GSet \wedge DSet2 \subseteq GSet$
$axm3 : ASet1 \cup ASet2 = ASet$
$axm4 : ASet1 \cap ASet2 = \varnothing$
$axm5 : DSet \subset ASet$
$axm6 : DSet1 = DSet \cap ASet1$
$axm7 : DSet2 = DSet \cap ASet2$
$axm8 : a \in ASet$
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 $ASet$ to the variable x at the global level. Machine A is refined to machine C, in which two agents can allocate from their statically

assigned partitions, with each agent allocating to his own variable $x1$ or $x2$ repectively, and where each agent allocation refines the global allocation event.

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 being 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 $DSet1, DSet2$.

```
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 ∈ ASet1 − 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
```

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 the 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
      OUTPUT out1 : 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
      OUTPUT out1 : 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 assumptions pertaining to the two events' parameters are identical. In $Ret_{C,D}$ however, the same situation is less trivial, since machine *C*'s $AddEl1$ *el* is selected from *ASet* while machine *D*'s $AddEl1$ *el* is selected from *DSet*. If we temporarily rename the parameters in these two events by adding subscripts, the real within relation between the $AddEl1$ 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 $AddEl1$ 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. Retrenchments and Anticipating Events

The tiny case study just discussed had one telling feature, namely that due to the simplicity of the case study, the events related by the *A* to *B* retrenchment and the events related by the *C* to *D* retrenchment in each case preserved the retrieve relation. This simplicity was itself a consequence of the desire to keep the example small enough so that all of its ingredients, including four machines and four relationships between machines, could be described in detail within a

[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 [Roda], 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.

reasonable amount of space. When corresponding events of a retrenchment preserve the retrieve relation, an alternative technique is sometimes available for tackling the development step. It may be possible to use the anticipating events of [ACM05].

In this technique, the events that would be related by retrenchment between abstract and concrete models in our approach, are of course related by the refinement subset of that retrenchment. New events, those that modify the abstract variables in a manner that is incompatible with being a refinement of skip, are introduced in the concrete machine as if they *did* refine an implicit abstract event, but this time not a skip event, instead a different, 'keep' event. In utilising a keep event, the top model (machine A in our case study), is modified by the inclusion of an event (called $Keep_A$ say), whose only job is to preserve the invariants. In other words in the keep event, the variables are nondeterministically assigned to any values that make the invariants true. The new events in the concrete machine are now free to modify the variables inherited from the abstract machine in whatever manner they wish, while still conforming to the demands of a refinement, since any behaviour refines maximally nondeterministic assignment.

In the context of our case study, machine A would implictly acquire an event $Keep_A$ which had a trivial guard, and whose action was $x :\in \mathbb{P}(ASet)$. It is clear that once machine A has such an event, then event $SubEl$ in machine B is a refinement of it, and one can even very easily posit a variant (eg. $\mathrm{card}(y)$) that is decreased by it.

While the use of the keep (or anticipating) event can undoubtedly handle some of the more benign situations catered for by retrenchment, there are a number of significant differences.

1. New events introduced via this mechanism still have to decrease a variant, to conform to Event-B's other refinement demands. This may not be appropriate for all modifications to a system that one might contemplate performing, and retrenchment makes no such demand.
2. Likewise, the new events' guards also have to conform to the demands on guards pertaining to Event-B refinement. They thus have to co-operate with the other guards to ensure guard strengthening and relative deadlock freedom. Retrenchment makes no such demands.
3. The introduction of the keep event into the abstract machine alters the problem solved by that machine. The original machine solved a problem that one could phrase as 'reachability under the (original) collection of events'. The same machine modified by the introduction of the keep event, whose action is maximally nondeterministic assignment while maintaining the invariants, solves a problem that one could phrase as 'reachability under the (original) invariants'. Since there is no requirement in Event-B that all states characterised by the invariants have to be reachable via (some sequence of) the events, the former problem defines a stronger reachability problem than the latter. Thus, unlike the implicit introduction of a skip event, which obviously does not alter the reachability problem solved by the machine, the implicit introduction of a keep event *does* alter the reachability problem, and thus constitutes a nontrivial modification of the top level machine.

In contrast to the points just noted, one of the most prominent aims of retrenchment is to *not* require any change in the machine being retrenched (i.e. the machine that plays the role of the abstract model). If one is prepared to change the abstract machine sufficiently, then of course many more transformations become capable of being viewed as refinements than before. Ultimately, the refinement concept itself can become blurred due to the many implicit things that might be being imposed on the abstract model during its use.

5.3. A, B, C, D and the Tower

We return to the discussion of our simple case study. Machines A, B, C, D, (and the various retrenchments and refinements that relate them), form a candidate instance of the Postjoin Theorem. It is time to pick up the discussion left over from Section 4 regarding this. In fact, due to the totality of the retrieve relations (for both the retrenchments and the refinements), the square actually commutes 'on the nose' as far as the various retrenchments and refinements are concerned, so the differences in approach between [Jes05] and [BJ] are not visible in such a simple example. Thus, 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 $AddEl1$ and $AddEl2$ in machine D, by tracing the square via B or C. Since $AddEl1$ and $AddEl2$ 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 output and concedes relations for *AddEl* and *AddEl*1 are determined similarly. Take the output relation. One way round, the output 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 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 output relation of the *C* to *D* retrenchment. Either way round the square yields the same result. The strategy for the concedes relation of the *A* to *B* retrenchment is exactly the same. See [BJP08] 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
      OUTPUT out1 : 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 are all *regular* [Ban95]. Moreover, benign situations also feature compositions of regular relations which themselves turn out to be well behaved. In such cases one can confidently eschew the forbidding complexity of the results in [Jes05], or their much less forbidding analogues in [BJ], and as here in the Event-B context, work by hand.

6. A Small Train Case Study

In this section we examine another case study. The scenario concerns trains. In the old days, the rail system relied for its safety on the vigilance of the train driver who was expected to see, and to respond to, all signals on his route. Inevitably, given the dependence on human vigilance, there were some (rare) tragedies, attributable to the driver's not in fact responding to a signal that he was expected to respond to. Other sources of accidents were attributable to the signal system itself not behaving properly, eg. the signal did not move, or change colour in the way it was supposed to, in response to commands from the signal box.

As a reaction to circumstances like these, Automatic Braking Systems (ABS) were invented. These are electronic systems —built from hardware of the highest dependability— that communicate with onboard equipment on the train in order to override the train's state of motion and bring it to a standstill unconditionally, if the train is not already coming to a stop by conventional means.

ABS systems are expensive and their installation is often resisted by railway operating companies, for as long as it

[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 [Roda] Deliverable D3. See the Rodin User Manual at [Rodb]. When there are no such abstract parameters, the witness relation trivialises.

is politically realistic to do so, until regulatory pressure and/or public opinion forces the issue. We will develop a small model of the installation of ABS onto a conventional system. Since the whole object of the exercise is to change the top level behaviour under certain circumstances, we are faced with a new use-case that conflicts with existing behaviour, and must therefore use retrenchment rather than refinement to accomodate it.[6]

Below on the left is the original, rather primitive, *Train* system. For simplicity, we have folded in the static context information into the body of the MACHINE. There are two variables: the motor *motor*, which is *on* when the train is moving and *off* when it is stationary, and the dead man's handle *dmh*, which must be *on* when the motor is running, as captured in $inv3$. The dead man's handle and motor are independent systems, and the $inv3$ coupling between them provides a higher level of dependability than if the motor alone was present. However, for driver convenience, the two are physically connected in the accelerator handle, which the driver pushes forward to go, holds forward to continue going, and releases to stop. So the two nontrivial events, *Go* and *Stop*, couple the setting and unsetting of the *motor* and *dmh* variables.

```
MACHINE Train
VARIABLES motor, dmh
INVARIANTS inv1 : motor ∈ {on, off}
           inv2 : dmh ∈ {on, off}
           inv3 : motor = on ⇒ dmh = on
EVENTS
  INITIALISATION
    BEGIN act1 : motor := off
          act2 : dmh := off
    END
  Go
    WHEN grd1 : dmh = off
         grd2 : motor = off
    THEN act1 : motor := on
         act2 : dmh := on
    END
  Stop
    WHEN grd1 : dmh = on
         grd2 : motor = on
    THEN act1 : motor := off
         act2 : dmh := off
    END
END
```

```
MACHINE ABSTrain
VARIABLES motor_ABS, dmh_ABS, ABS
INVARIANTS inv1 : motor_ABS ∈ {on, off}
           inv2 : dmh_ABS ∈ {on, off}
           inv3 : motor_ABS = on ⇒ dmh_ABS = on
           inv4 : ABS ∈ {OK, KO}
           inv5 : motor_ABS = on ⇒ ABS = OK
EVENTS
  INITIALISATION
    BEGIN act1 : motor_ABS := off
          act2 : dmh_ABS := off
          act3 : ABS := OK
    END
  Go
    WHEN grd1 : dmh_ABS = off
         grd2 : motor_ABS = off
         grd3 : ABS = OK
    THEN act1 : motor_ABS := on
         act2 : dmh_ABS := on
    END
  Stop
    WHEN grd1 : dmh_ABS = on
         grd2 : motor_ABS = on
         grd3 : ABS = OK
    THEN act1 : motor_ABS := off
         act2 : dmh_ABS := off
    END
  ABSStop
    WHEN grd1 : dmh_ABS = on
         grd2 : motor_ABS = on
         grd3 : ABS = OK
    THEN act1 : motor_ABS := off
         act2 : ABS := KO
    END
  ResetABS
    WHEN grd1 : motor_ABS = off
         grd2 : ABS = KO
    THEN act1 : dmh_ABS := off
         act2 : ABS := OK
    END
END
```

Next to *Train* is the ABS-enhanced version *ABSTrain*, where the variables inherited from *Train* have been given an '$_{ABS}$' subscript for clarity. *ABSTrain* has an extra variable *ABS*, the state of the ABS on-board system, which can be *OK* or *KO*. Normally it is *OK* and this fact becomes an additional guard on the existing *Train* events — thus far we have nothing beyond a superposition refinement [BS96, Kat93]. The novelty comes in the fact that the ABS can take

[6] Of course we could develop the ABS train system *from scratch* using refinement alone, but we could not use the *existing system* as a starting point, thus losing some connection with the system goals if the objective is indeed to modify the existing system.

unilateral action to stop the train. Thus there is a new *ABSStop* event, which stops the train when the state of the dead man's handle is *on* but the on-board ABS equipment goes into the *KO* state on receipt by the on-board ABS equipment of a *STOP* signal from the trackside ABS equipment. Since the motor turns *off* while the dead man's handle remains *on* (the driver might have collapsed and died, falling onto the dead man's handle and jamming it in the *on* position), we have an event that manipulates the *Train* state in a manner incompatible with previous *Train* events. So it can neither be a refinement of any of them, nor can it be equivalent to skip on this state.

A further event *ResetABS* restes the state of the ABS system, provided the motor is not running. Part of its responsibility is to reset the dead man's handle too, if necessary, so it constitutes another case of incompatible manipulation of the top level *Train* state.

Regarding the retrenchment data from *Train* to *ABSTrain*, the most natural retrieve relation will obviously be a pair of equalities $motor = motor_{ABS} \wedge dmd = dmh_{ABS}$. Aside from that, for the common events, *Go* and *Stop*, since their actions are the same (up to variable renaming) in the two models, they re-establish the retrieve relation in the after-states, so that all we need is a non-trivial WITHIN relation, in each case saying $ABS = OK$. This is just a guard strengthening (from the Event-B perspective), but needs to be spelled out in a retrenchment since a general WITHIN relation may contain further constraints if needed.

The simplicity of the retrenchment data attests to the clean separation between the newly introduced behaviour, and the old behaviour, despite the fact that the former manipulates the top level variables in a non-trivial way, and this simplicity of retrenchment data is something we regard as a *good thing*. The reason it arises is because Event-B strongly encourages the encapsulation of (distinct fragments of) distinct behaviour in separate events. This separation into distinct events impacts the name space of the events, and under the usual working assumption, that identically named events are the ones we expect to correspond in the two models, the new behaviour has a fresh name, and therefore lies outside the scope of the retrenchment data. Thus, in the context of the present case study, it is much more natural to regard the *ABSStop* behaviour as constituting a separate event, than to view it as a modified version of the normal *Stop* event. This is strongly supported by the requirements context in this case, since the triggering of the ABS system from trackside, *really is* a separate event in the normal, informal, human, perception of what an event is, and is quite distinct from conventional stopping.

Furthermore, the clean separation can be expected to persist down the refinement hierarchy. So if one develops the *Train* system to a lower level of detail via refinement, the same refinement strategy will be capable of a clean extension to the *ABSTrain* system, for which the tower construction would produce a clean template. We do not elaborate the details further for this example, but refer to the next case study where this issue is pursued more deeply.

7. A Small Telephony Case Study

Telephony is a classic area where there is a large installed base of conventional systems, whose behaviour gets modified by the addition of new equipment featuring a greater range of capabilities. These days, we are no longer surprised by the availability of sophisticated connection services brought about by the digitisation of the telephone network.

The conventional model of telephony is captured in the name Plain Old Telephone System (POTS). In a POTS system, one can dial numbers, and they can ring, be busy, or be unobtainable. That's about it for POTS. We model a small fragment of the modification of POTS by the addition of a call forwarding facility.

Below on the left is an abstraction of a fragment of the POTS model. The only variable is *tone*, which is a function from *NUMBERS* to the tone heard when the handset at the given number is listened to, being one of: *idle*, *ring*ing, *busy*, (for simplicity we do not model the unobtainable tone). We only model *Dial*ling, and the result of a dial is just the tone obtained, which can be either *ring*ing or *busy*. (In particular, we do not model call establishment or disconnection, so that with the tiny models that result, we can illustrate a broader range of manipulations within a small space.)

To the right is an enhancement of this model, the CFPOTS model, permitting call forwarding. This is modelled rather crudely by the addition of a new tone, the *for*warding tone, as an outcome of *Dial*ling. Aside from that, the variables and parameters have been renamed for clarity.

On the understanding that different tones correspond to functionally different outcomes (especially as regards further refinement), the relationship between POTS and CFPOTS cannot be a refinement. However it makes for a sensible retrenchment (which is given in detail below). The retrieve relation for such a retrenchment will simply equate the *tone* and $tone_{CF}$ variables (on the set of values that they have in common). Beyond this, only the *Dial* event requires consideration. It will need a within relation to equate *from* to $from_{CF}$ and *to* to to_{CF}, a trivial output relation, and a concession that says that $tone(from) = busy$ while $tone_{CF}(from_{CF}) = for$.

Pausing a moment, note that we could have attempted to model this situation in a manner analogous to the previous two sections, by splitting off the new functionality into a separate event, *CFDial* say. However, in contrast to the

previous two sections, it is, here, highly unnatural to regard the new behaviour as belonging to a distinct event. When a user picks up a phone to make a call, he perceives himself as performing a single event. It may well be that this single event has a range of outcomes that depends on the details of the system he is using (and whose full capabilities he may, in any case, not be fully aware of), but his perception of performing a single event is not affected by this. Thus, retrenchment gives us a choice of different ways of modelling what are, mathematically, very similar facts. And the argument about which method is superior in any given situation will rest squarely on requirements considerations — retrenchment, as a technique distinct from refinement, would be greatly undermined if it did not permit us to get deeper into requirements issues in this way.

```
MACHINE POTS
VARIABLES tone
INVARIANTS
  inv1 : tone ∈ NUMBERS → {idle, ring, busy}
EVENTS
  INITIALISATION
    BEGIN act1 : tone := NUMBERS × {idle}
    END
  Dial
    ANY from, to
    WHERE grd1 : from ∈ NUMBERS
          grd2 : to ∈ NUMBERS
          grd3 : tone(from) = idle
    THEN
      act1 : tone(from) :∈ {ring, busy}
    END
END
```

```
MACHINE CFPOTS
VARIABLES tone_CF
INVARIANTS
  inv1 : tone_CF ∈ NUMBERS → {idle, ring, busy, for}
EVENTS
  INITIALISATION
    BEGIN act1 : tone_CF := NUMBERS × {idle}
    END
  Dial
    ANY from_CF, to_CF
    WHERE grd1 : from_CF ∈ NUMBERS
          grd2 : to_CF ∈ NUMBERS
          grd3 : tone_CF(from_CF) = idle
    THEN
      act1 : tone_CF(from_CF) :∈ {ring, busy, for}
    END
END
```

```
MACHINE POTSR
REFINES POTS
VARIABLES tone, calls
INVARIANTS
  inv1 : tone ∈ NUMBERS → {idle, ring, busy}
  inv2 : calls ∈ ℙ(NUMBERS × NUMBERS)
EVENTS
  INITIALISATION
    BEGIN act1 : tone := NUMBERS × {idle}
          act2 : calls := ∅
    END
  Dial
    REFINES Dial
    ANY from, to
    WHERE grd1 : from ∈ NUMBERS
          grd2 : to ∈ NUMBERS
          grd3 : tone(from) = idle
    THEN
      act1 : tone(from) :∈ {ring, busy}
      act2 : calls := calls ∪ {(from, to)}
    END
END
```

```
MACHINE CFPOTSR
REFINES CFPOTS
VARIABLES tone_CF, calls_CF
INVARIANTS
  inv1 : tone_CF ∈ NUMBERS → {idle, ring, busy, for}
  inv2 : calls_CF ∈ ℙ(NUMBERS × NUMBERS)
EVENTS
  INITIALISATION
    BEGIN act1 : tone_CF := NUMBERS × {idle}
          act2 : calls_CF := ∅
    END
  DialRB
    REFINES Dial
    ANY from_CF, to_CF
    WHERE grd1 : from_CF ∈ NUMBERS
          grd2 : to_CF ∈ NUMBERS
          grd3 : tone_CF(from_CF) = idle
    THEN
      act1 : tone_CF(from_CF) :∈ {ring, busy}
      act2 : calls_CF := calls_CF ∪ {(from_CF, to_CF)}
    END
  DialF
    REFINES Dial
    ANY from_CF, to_CF
    WHERE grd1 : from_CF ∈ NUMBERS
          grd2 : to_CF ∈ NUMBERS
          grd3 : tone_CF(from_CF) = idle
    THEN
      act1 : tone_CF(from_CF) := for
      act2 : calls_CF :=
             calls_CF ∪ {(from_CF, FORTAB(to_CF))}
    END
END
```

Next, we refine our models. At this next level of detail, we introduce a system internal data structure, the *calls* variable, which is a directed graph over the *NUMBERS* set that captures the active calls. In the POTSR machine, which refines POTS, the *Dial* event simply superposes this functionality onto the assignment of *tone*, adding an edge

from *from* to *to* to the *calls* graph. In the CFPOTSR machine, which refines CFPOTS, the overall behaviour is more interesting. For the *ring* and *busy* outcomes, the refinement of *Dial* to *DialRB* is the same as the refinement of *Dial* in the POTSR machine. However, in the *for* outcome, the system has a forwarding table *FORTAB*, that says where calls should be forwarded to, and in this case, an edge from $from_{CF}$ to $FORTAB(to_{CF})$ is added to the $calls_{CF}$ graph. To avoid technical complications, we assume that any number which is the target of forwarding can never be the source of further forwarding, so that the *FORTAB* graph is a set of disconnected edges.

Regarded as Event-B refinements, both the POTS to POTSR refinement and the CFPOTS to CFPOTSR refinement are basically trivial superposition refinements. And as for POTS and CFPOTS above, the relationship between POTSR and CFPOTSR needs to be a retrenchment. The two retrenchments, *Pots2CFPots* and *PotsR2CFPotsR*, are given in detail as follows.

```
RETRENCHMENT Pots2CFPots
FROM POTS TO CFPOTS
RETRIEVES ret1 : ∀ num •
   num ∈ NUMBERS ∧ tone_CF(num) ≠ for ⇒
      tone(num) = tone_CF(num)
EVENTS
   RAMIFICATIONS Dial
      WITHIN wth1 : from = from_CF
             wth2 : to = to_CF
      OUTPUT out1 : true
      CONCEDES
         con1 : ∀ num •
            num ∈ NUMBERS ∧ tone_CF(num) ≠ for ⇒
               tone(num) = tone_CF(num)
         con2 : tone(from) = busy
         con3 : tone_CF(from_CF) = for
      END
END
```

```
RETRENCHMENT PotsR2CFPotsR
FROM POTSR TO CFPOTSR
RETRIEVES ret1 : ∀ num •
   num ∈ NUMBERS ∧ tone_CF(num) ≠ for ⇒
      tone(num) = tone_CF(num) ∧
      calls(num) = calls_CF(num)
EVENTS
   RAMIFICATIONS Dial TO DialRB
      WITHIN wth1 : from = from_CF
             wth2 : to = to_CF
      END
   RAMIFICATIONS Dial TO DialF
      WITHIN wth1 : from = from_CF
             wth2 : to = to_CF
      OUTPUT out1 : true
      CONCEDES
         con1 : ∀ num •
            num ∈ NUMBERS ∧ tone_CF(num) ≠ for ⇒
               tone(num) = tone_CF(num) ∧
               calls(num) = calls_CF(num)
         con2 : tone(from) = busy
         con3 : tone_CF(from_CF) = for
         con4 : calls' = calls ∪ {(from, to)}
         con5 : calls'_CF = calls_CF ∪
                            {(from_CF, FORTAB(to_CF))}
      END
END
```

First, we make some observations about the *Pots2CFPots* retrenchment. Aside from the simple facts mentioned already above, we note that the retrieve relation is of the form: $\forall\, num \bullet \mathsf{IsGood}(num) \Rightarrow \mathsf{SomeFactsAbout}(num)$. The distinction between 'good' *num* values and 'not-so-good' ones, can conveniently be made on the basis of a property of *num* in the *CFPOTS* system. This is an embodiment of one possible generic *pattern* for the retrieve relation in a retrenchment between two systems which are both (compatibly) structured as functions from an index set (here *NUMBERS*) to the state of an individual subsystem. In Z, such a mechanism is called promotion, and is a frequently used technical device. (See [BPJS] for a broader discussion of the issues surrounding retrenchment in the context of Z promotion.) The lack of promotion here, means that we have the responsibility to speak about the system as a whole when we formulate properties of events —even though we know that only one subsystem is active during a given event— if we wish to forestall the 'saying nothing intends the interpretation of true' consequences of focusing on the subsystem in question alone. This point raises its head in the concession of *Dial*, where we do in fact make a statement about *num*s other than $from_{(CF)}$ and $to_{(CF)}$. In particular, the statement we make about them is a copy of the retrieve relation implication itself, and the fact that we can do so indicates that the retrieve relation and concession are compatible; nothing in the formulation of retrenchment forbids this. In fact, as soon as a number in the *CFPOTS* system opts for the *for* outcome, it ceases to satisfy the hypotheses of the retrieve relation implication, allowing us to include the latter in the concession.

In the context of the last point, and the simplicity of the models we are dealing with, the reader may notice that we could have written a stronger retrieve invariant: $\forall\, num \bullet num \in NUMBERS \wedge [(tone_{CF}(num) \neq for \wedge tone(num) = tone_{CF}(num)) \vee (tone(num) = busy \wedge tone_{CF}(num) = for)]$, and ask why we did not. The result of doing so would, in this very simple case, have allowed us to omit the concession altogether, and we would have ended up with something akin to a superposition refinement, but one that manipulated existing variables (as a result

of introducing a finer case split), rather than one that merely manipulated new variables, as is the convention. While such an approach would have been justified in this simple example, it would not have been very generic. For more complicated examples (even more deeply elaborated versions of this one, were we to introduce a more realist selection of events), the connection between the 'abstract' and 'concrete' models of a retrenchment need not allow such a simple and compact extension of the retrieve relation to the concession cases. In general, one might need something like an enumeration of states reachable in the two incompatible system models when one examines all possible paths, and a suitable classification of them into a relation. In general, if nothing else, this quickly becomes syntactically infeasible as a useful description mechanism for the relationship between the models, for 'path explosion' reasons. For this reason we elected to illustrate the generic treatment of these issues.

All of the above notwithstanding, the *really* interesting question is the extent to which the system *CFPOTSR* and its impinging retrenchment and refinement could have been manufactured automatically by means of a square completion procedure. An automatic square completion procedure is the embodiment of a piece of mathematics, ultimately about a class of transition systems connected via relationships of a particular kind. One thing such an entity cannot do is *design*. For example, in the context of the present case study, there is nothing in the mathematics to preclude the 'pot luck' semantic model for call forwarding, in which the system, on encountering a busy destination, chooses, if it wants to, a random *idle* phone to connect the caller to. The pot luck semantics might be an appropriate design for an 'encounter' service that a phone company may decide to provide for its customers, but it is inappropriate for call forwarding. However, the distinction lies squarely in the requirements arena.

In the earlier A, B, C, D machines example, the objective of the refinement was data refinement. In such a case, provided the retrieve relation enjoys sufficient totality and surjectivity properties, it is not unreasonable to expect that, first mathematics, subsequently its automation, can manage the extension of the data refinement to exceptional cases described using the concession capability of the retrenchment. However, in the present case study, the *calls* data structure is introduced in the *POTSR* machine without any relationship that connects it to the existing *tone* data structure (because it is a superposition). Therefore, the mathematics will stuggle to say anything sensible about what should be done with the *calls* data in the *for* case in the *CFPOTSR* system, since the *for* case falls outside the scope of the *Pots*2*CFPots* retrieve relation. In such a situation, the appropriate mathematical behaviour is to leave the relevant variables unconstrained. This amounts to a maximally unconstrained design (for that part of the system), which can can subsequently be refined by the designer in order to give voice to the design aims of that stage of the development. In our case study, an automatic procedure would have left the system's behaviour in the *for* case of (the automatically produced precursor of) the *CFPOTSR* system unconstrained, leaving it to a human to decide to resolve the maximal nondeterminism in that part of the system by choosing a table lookup strategy (like we had), as opposed to a pot luck strategy, or other alternative. The same thinking guided the partition of the *Dial* event in *CFPOTS* into the two lower level events *DialRB* and *DialF*. An automated system, insensitive the requirements issues, would not have had the wisdom to decide on such a partition (at least not without guidance from sophisticated heuristics — the latter possibility is not excluded). In this paper, it is envisaged that the decision to do the partition was made by a human, while refining the automatically produced precursor of *CFPOTSR*.

8. Rodin Tool Design Issues

In Section 3 we mentioned that our formulation of retrenchment for Event-B was designed to be compatible with the Rodin Toolset (as it is at the time of writing). In this section, we elaborate this observation, and describe some of the details that such an integration of retrenchment into Rodin would consist of. (The actual implementation of such details remains as work for the future.)

The Rodin toolset [Rodb] is built on top of Eclipse [Ecl], as a family of plugins that manage the whole of the Event-B development process. Since the source code of Rodin is in the public domain, the proposed extensions for retrenchment become unproblematic.

The starting point for the incorporation of retrenchment into Rodin would be the introduction of the RETRENCHMENT syntactic construct into Rodin. Since Rodin already has extensive facilities for the incremental processing of syntactic constructs (notably MACHINEs), this requires little more than the adaptation of the existing code. However, since retrenchment is defined to relate top level machines, further facilities need to be provided to mechanically translate a refinement machine into top level form, as observed in Section 3.3. We note though, that this procedure has a generic description, eg. as in Chapter 11 of [Abr96], which makes the mechanisation routine in principle.

Once the RETRENCHMENT syntactic construct exists inside Rodin, the relevant proof obligations can be generated. From Section 3 it is clear that these are of a very similar form to those of refinement, so once more, the adaptation of the existing proof obligation generator will not be excessively challenging.

In Rodin, most routine tasks run unprompted in the background, once their needed inputs become available via a SAVE. The same approach will work for retrenchment. Once a RETRENCHMENT construct is saved, it can be checked for consistency. Once it is seen to be internally consistent (i.e. it conforms to the grammar outlined in Section 3.3), it can be checked for external consistency. This starts by checking that the machines referred to exist and are top level machines. If a needed machine doesn't exist, an error can be flagged. If a needed machine turns out not to be top level, the procedure for generating the equivalent top level machine can be invoked, and the system can police the relationship between the original refinement machine and its top level generated counterpart. (For instance, user-instigated changes to the generated top level machine can be prevented, and changes to the refinement machine can be made to cause regeneration of the top level machine, and a re-doing of the subsequent dependent actions.) Once both needed machines are in a satisfactory state, their internal details can be checked; in particular whether the variables in the RETRENCHMENT construct are indeed variables of the two machines in the required way, and then whether the events related via the retrenchment's RAMIFICATIONS indeed exist as required. Once everything is in order the proof obligation generator can be let loose, leading to the subsequent automatic invocation of the provers which attempt to discharge the generated POs.

The preceding constitutes basic support for retrenchment in Rodin. One can use this as a springboard for more extensive support for the tower. However, whereas most of the activity surrounding the treatment of the basic retrenchment construct can be managed 'behind the scenes' in the good old Rodin way just described, when it comes to the tower, a little more active user control is probably beneficial.

Thus, in theory, Rodin could be programmed to search for 'tower opportunities', typically a $\ulcorner$ shape as in Section 4, or a corresponding $\llcorner$ shape, whenever a new construct was saved.[7] However this might result in the tool discovering tower opportunities that were in conflict with the developer's system architecture aims, especially when there were many machines and relationships already commited to the Rodin system during some development. A better strategy would be to enable the user to select three machines from the current development (that were already arranged in a suitable shape via an existing retrenchment and refinement), and to prompt Rodin to commence tower construction. This, and the ensuing tower-related activities described below, would be best done in a new Rodin 'Tower Perspective'.

Assuming that we were dealing with top level machines throughout (and if not, the situation would be dealt with as described above), Rodin would then have two tasks. Firstly, the checking of any hypotheses needed for the square completion process to work; this amounts to simply a standard bout of proof obligation generation and subsequent discharge. Secondly, the square completion process itself; here Rodin would assemble the new system following the prescription contained in the requisite theorem. Optionally, the proof obligations for the refinement and retrenchment that establish the two new edges of the completed square could be generated. Since these confirm facts proved generically in the requisite theorem, they would always be provable in principle (regardless of whether any specific prover was up to the task in any particular instance or not).

An important feature of the tower theorems, is their encapsulation of the constructed system within a universality class of systems, these being interconnected by relationships that invariably include inter-refinability. This enables the replacement of the generically constructed system by one that is more obviously aligned with the application requirements. A Rodin user would thus create a new refinement machine (intended for the purpose), and signal its intended role as replacement square completion to Rodin. Rodin could then not only create the standard proof obligations intrinsic to the refinement, but also the converse ones, needed for establishing the refinement in the other direction. (Note that this may require a mild extension of the Event-B/Rodin convention that an event may be refined by more than one refinement event but not vice versa; since if an event Ev_A is indeed refined by more than one refinement event, Ev_{C1} and Ev_{C2} say, then in the converse refinement, the original event Ev_A will have to refine the disjunction of Ev_{C1} and Ev_{C2}.) An alternative to two-way inter-refinability (the true significance of which —when considering the system requirements— depends heavily on the nature of the retrieve relation used) is conventional one-way refinement. If the inverse refinement is not provable, it means that some additional genuine *design towards implementation* has been incorporated into the refinement step. There is no reason to try to prevent this within the tool.

9. Conclusions

Event-B, like all refinement based methodologies, proceeds top-down. This means that levels of abstraction must be complete (in terms of what use will ever be made of the variables that belong to that level) at the point that they are

[7] The other two possible 'tower opportunities', corresponding to shapes $\urcorner$ and $\lrcorner$ are of largely theoretical rather than practical interest —since they refer to properties of *converse retrenchments*, corresponding to 'undeveloping' an application, which one seldom does in practice— thus they probably do not merit the investment of effort required to incorporate them into the tool.

introduced.[8] This insistence on the order in which things are introduced in refinement based methodologies prevents their integration with today's 'Agile Methods' and other system construction practices, which are typically much more flexible about the order in which different pieces of the system are brought into the evolving design. Assuming that bringing the correctness achievable using techniques like Event-B to such agile (and similar) methods would be a good thing, we argued that retrenchment, with its toleration of manipulating the top level state in non-skip ways, and of even more drastic modifications forbidden by Event-B, provided a means by which we could bridge this gap.

For the sake of being specific, we focused on the UCw approach, which tends to develop the final system via vertical columns of functionality rather than horizontal layers of abstraction. In order to achieve this we reformulated retrenchment in a form suitable for Event-B, and for Rodin. Retrenchment bridges the gap via the *Tower Pattern*, a commuting arrangement of retrenchments and refinements which can be constructed in a number of ways in order to suit the desired system construction strategy. We illustrated the use of the 'top down' orientation of the tower as a means of integrating Event-B correctness with the UCw approach, by means of a number of small case studies: simple set manipulation, automatic train braking, and call forwarding in telephony. Some of these examples nicely illustrated the distinction between what could be expected to be achieved by an automatic construction, and what would have to be added, in terms of design intent, by hand, afterwards.

The same technical considerations that we have been discussing also enlarge the scope for Event-B to tackle a wider variety of 'real-world' applications. For example, the fact of Event-B's insistence that all data types are discrete, inhibits (at minimum) 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.

On a technical level, the notion of retrenchment we introduced in Section 3.3 was the natural adaptation of the generic formulation of retrenchment in [BPJS07a, BJP08] to the Event-B situation. This incorporated the usual three relations of a generic retrenchment into Event-B. In particular, it included the output relation in the PO conclusions, even though this never got used nontrivially in any of our case studies. It would be tempting (in the context of Event-B) to omit the output relation entirely, since Event-B insists on treating system outputs using normal state variables, thus enabling all output considerations to be relegated to the retrieve relation.[9] This certainly suffices for most applications, including all of ours. Just occasionally though, it is desirable in a retrenchment to highlight specific before-after properties of state variables, or to relate these to variables that are performing output roles, in situations where the retrieve relation is maintained —i.e. to *strengthen* the retrieve relation— as for example happens in the Mondex case study in [BPJS07b].[10] In Event-B (and in general) it would be easy enough to include all such relevant facts in the concession, but while the ramifications of the resulting operations would encapsulate the relevant facts in a provable way, the strengthening of the retrieve relation would not be *guaranteed* by the logic. For this reason, we retained the output relation in this paper, even though we take it as read that it would be a seldom used feature.

A further interesting technical point concerns the granularity of naming of events. In Section 6 we encountered a situation in which it was reasonable to package newly introduced behaviour into new events, for requirements reasons. In Section 7 we had the converse situation. There, it was more natural to include the new behaviour as a possible outcome of an existing event, again for requirements reasons. So both approaches can be justified. The field here gets more muddied when we consider developing beyond the pure event world, getting closer to actual code. This requires the merging of finegrained events into pieces of sequential code (see [Abr]). The fact that retrenchment can equally easily handle both situations: ones in which new behaviour is separately packaged, and ones in which it is included with existing behaviour, becomes a strength in also dealing with these later phases of development.

It would of course be desirable to mechanise the technology introduced in this paper, and in the immediately preceding section, we delved into what the main challenges would be in terms of a Rodin implementation. For this, as well as the obvious tool development, it would be necessary to formulate precise Event-B versions of the theorems of [Jes05, BJ]. These would focus on the most useful cases of the tower constructions in a manner that made the subsequent mechanisations as straightforward as possible. All of this remains as work for the future.

[8] The clean state of affairs just noted gets a little blurred in considering the keep events of Section 5.2, and other techniques achievable via surreptitious modifications of the abstract machine.

[9] In fact the precursor of this paper [Ban08], did develop this possibility in detail.

[10] It might be imagined when output is done via state variables, that all such considerations could be incorporated into a more elaborate retrieve relation. However this is not the case when the considerations in question depend on the *name of the event being executed*. This is what happens in [BPJS07b].

References

[Abr] J.-R. Abrial. *Modeling in Event-B: System and Software Design*. In press.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACM05] J-R. Abrial, D. Cansell, and D. Méry. Refinement and Reachability in Event-B. In *ZB 2005: Formal Specification and Development in Z and B* [ZB-05], pages 222–241.

[Ban] R. Banach. Model Based Refinement and the Design of Retrenchments. Submitted.

[Ban95] R. Banach. On Regularity in Software Design. *Sci. Comp. Prog.*, 24:221–248, 1995.

[Ban08] R. Banach. UseCase-wise Development: Retrenchment for Event-B. In *Proc. ABZ-08, LNCS 5238*, pages 167–180, 2008.

[BF05] R. Banach and S. Fraser. Retrenchment and the BToolkit. In *ZB 2005: Formal Specification and Development in Z and B* [ZB-05], pages 203–221.

[BJ] R. Banach and C. Jeske. Retrenchment and Refinement Interworking: the Tower Theorems. Submitted.

[BJP08] R. Banach, C. Jeske, and M. Poppleton. Composition Mechanisms for Retrenchment. *J. Log. Alg. Prog.*, 75:209–229, 2008.

[BJPS05] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the Purse: Finite Exception Logs, and Validating the Small. In *Proc. IEEE/NASA SEW30-06*, pages 234–245, 2005.

[BJPS06] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties. In *Proc. IEEE ISOLA-06*, pages 82–90, 2006.

[BP98] R. Banach and M. Poppleton. Retrenchment: An Engineering Variation on Refinement. In D. Bert, editor, *2nd International B Conference*, volume 1393 of *LNCS*, pages 129–147, Montpellier, France, April 1998. Springer.

[BPJS] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenchment and Promotion in Z. Submitted.

[BPJS05] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Finite Sequence Numbers, and the Tower Pattern. In *Proc. FM-05, LNCS 3582*, pages 382–398, 2005.

[BPJS07a] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and Theoretical Underpinnings of Retrenchment. *Sci. Comp. Prog.*, 67:301–329, 2007.

[BPJS07b] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1,1) Forward Refinements. *Fund. Inf.*, 77:29–69, 2007.

[BS96] R.J.R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[Ecl] Eclipse. The Eclipse Project. `http://www.eclipse.org/`.

[FB07] S. Fraser and R. Banach. Configurable Proof Obligations in the Frog Toolkit. In *Proc. Fifth IEEE International Conference on Software Engineering and Formal Methods*, IEEE Computer Society Press, pages 361–370. IEEE, 2007.

[Fra08] Fraser, S. *Mechanized Support for Retrenchment*. PhD thesis, School of Computer Science, University of Manchester, 2008.

[Jes05] C. Jeske. *Algebraic Integration of Retrenchment and Refinement*. PhD thesis, University of Manchester, 2005.

[Kat93] S. Katz. A superimposition control construct for distributed systems. *ACM TPLAN*, 15(2):337–356, April 1993.

[Roda] Rodin. European project rodin (rigorous open development for complex systems) ist-511599 http://rodin.cs.ncl.ac.uk/.

[Rodb] Rodin. The Rodin Platform. http://sourceforge.net/projects/rodin-b-sharp/.

[SCW00] S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory, 2000.

[ZB-05] *Proc. ZB-05*, volume 3455 of *LNCS*. Springer, 2005.