Structuring Retrenchments in the small with B

Michael Poppleton[1] and Richard Banach[2]

[1] Department of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK,
mrp@ecs.soton.ac.uk,
WWW home page: http://www.ecs.soton.ac.uk/ ~mrp/
[2] Department of Computer Science, Manchester University,
Manchester M13 9PL, UK,
banach@cs.man.ac.uk,
WWW home page: http://www.cs.man.ac.uk/~banach/

November 21, 2002

Abstract. Simple *retrenchment* is briefly reviewed as a liberalisation of classical refinement, for the formal description of application developments too demanding for refinement. Two generalisations, *output* and *evolving* retrenchment, are presented. Simple monotonicity results for retrenchment are recalled, forming the basis of a piecewise development method.
This work then commences the study of the structuring of developments using retrenchment. The aim is to decompose a single retrenchment between specifications, over most or all of the combined frame, into a set of finer-grained (i.e. of smaller domain) retrenchments over partitioned subsets of the combined frame. The partition may distinguish *inter alia* between refining, approximately refining, and non-refining sets of behaviours, and the decomposed retrenchments should be correspondingly stronger and more informative. Two decomposition results are given, that each sharpen a coarse-grained retrenchment description with respect to a general operation syntactic structure at concrete and abstract levels respectively. A third result decomposes a retrenchment, simultaneously exploiting structure in both levels. The theory is motivated by and applied to a simple example from distributed computing, and methodological aspects are considered.

1 Introduction

From early concerns about proving correctness of programs such as Hoare's [15] and Dijkstra's [14], a mature *refinement calculus* of specifications to programs has developed. Thorough contemporary discussion can be found in [13, 2]. For model-based specifications the term "refinement" has a very precise meaning; according to Back and Butler [3] it is a "...correctness-preserving transformation...between (possibly abstract, non-executable) programs which is transitive, thus supporting stepwise refinement, and is monotonic with respect to program constructors, thus supporting piecewise refinement". A succinct characterisation of refinement is a relation between models where the precondition is weakened and the postcondition strengthened.

This work develops the *retrenchment* method, a liberalisation of refinement. We argued, when introducing the notion [7, 8], for a weakening of the retrieve relation over the operation step, allowing concrete non-simulating behaviour in retrenchment. Concrete I/O may have different type to the abstract counterpart, and moreover the retrenchment relation may define fluidity between state and I/O components across the development step from abstract to concrete model. [19] reported initial work, using transitivity and monotonicity arguments, on the development of a calculus of retrenchment in B. This calculus was completed in [22], which showed all primitive operators of the B GSL, including operation sequence, to be monotonic with respect to retrenchment. [9, 10] gave a substantial theoretical presentation and development of the retrenchment framework, exploring the landscape between refinement, simulation and retrenchment. [5] addressed the integration of refinement and retrenchment from a methodological perspective. [20, 6] present two generalisations, *evolving* and *output* retrenchment respectively, that are applied in this paper.

A number of textbook-scale application examples of retrenchment have been presented, among the more substantial being [21, 12]. This work considers some requirements of practical retrenchment work in specification. More than one aspect of methodological support is required: the choice of abstractions and designs for understandable and decidable retrenchment obligations, how best to integrate with refinement methods, how to compose atomic retrenchment steps up to the scale of realistic specifications, how to decompose coarse-grained "first-cut" retrenchments to improve descriptiveness. The second aspect has been well considered [op.cit.]. [11] makes some commentary on the first, largely in response to the lack of published methodological reflection on refinement. This paper is concerned with the third and fourth aspects, the composition and decomposition of retrenchments.

A typical style of operational specification partitions the state/input domain in order to process each part of the partition appropriately; in B this case analysis approach is structured using a bounded choice over guarded GSL commands. This paper will concentrate on this style. A retrenchment relation covering the whole domain of such an operation and its concrete counterpart will in general document the processing choices in terms of a disjunctive choice of outcomes in the postcondition. Since there will usually be case structure at both levels, this disjunctive weakening effect is exacerbated. Describing such a given retrenchment as "coarse-grained", in this work we seek a decomposition into a collection of retrenchments, each of which is restricted to one branch of the case structure (at abstract or concrete levels separately, or both levels simultaneously). Each such decomposed retrenchment should be "finer-grained" (i.e. have stronger postcondition on a restricted domain) in the sense of including only one or some of the disjunctive possibilities in the postcondition of the composite retrenchment.

The paper proceeds as follows. Section 2 recaps syntactic and semantic definitions for simple retrenchment and its evolving variant. On the way output retrenchment is introduced. We restate the transitivity theorem of [22, 19], used for composing two retrenchments. Section 3 recalls basic monotonicity results for retrenchment, which provide a baseline for structuring retrenchments in specification. Section 4 presents a running example retrenchment to motivate the discussion, and demonstrates how the disjunctive shape of the simple retrenchment obligation is coarser and less descriptive than is desirable. Section 5 gives a number of results for retrenchment decomposition. Three syntactic patterns are given for decomposing a single retrenchment into a more descriptive, finer-grained family of retrenchments. Each pattern is shown to be a valid decomposition in general. Section 6 applies the decomposition to the example to show its utility, and section 7 concludes.

2 Simple, Output and Evolving Retrenchment

2.1 Introduction

Simple retrenchment is a weakening of the refinement relation between two levels of abstraction; loosely speaking, it strengthens the precondition, weakens the postcondition, and introduces mutability between state and I/O at the two levels. The postcondition comprises a disjunction between a retrieve relation between abstract and concrete state, where refining behaviour is described, and a concession relation between abstract and concrete state and output. This concession (where non-refining concrete behaviour is related back to abstract behaviour) is the vehicle in the postcondition for describing I/O mutability. Use of the simple retrieve relation, however, precludes I/O mutability being described in the case of refining behaviour. In output retrenchment [6] we conjoin an "output" clause to the retrieve clause to deal with this.

2.2 Simple and Output Retrenchment

Figure 1 defines the syntax of output retrenchment in the B language of J.-R. Abrial [1]; it differs only from the simple form in the addition of the OUTPUT clause. Abstract machine M has parameter a, state variable u, and invariant predicate $I(u)$. Variable u is initialised by substitution $X(u)$, and is operated on by operation *OpName*, a syntactic wrapper for substitution $S(u, i, o)$, with input i and output o. Unlike a refinement, which in B is a construct derived from the refined machine, a retrenchment is an independent MACHINE. Thus N is a machine with parameter b (not necessarily related to a), state variable v, invariant $J(v)$, initialisation $Y(v)$, and operation *OpNameC* as wrapper for $T(v, j, p)$, a substitution with input j and output p.

```
MACHINE             $M(a)$        MACHINE           $N(b)$
                                  RETRENCHES        $M$
VARIABLES           $u$           VARIABLES         $v$
INVARIANT           $I(u)$        INVARIANT         $J(v)$
                                  RETRIEVES         $G(u, v)$
INITIALISATION      $X(u)$        INITIALISATION    $Y(v)$
OPERATIONS                        OPERATIONS
    $o \longleftarrow OpName(i) \widehat{=}$          $p \longleftarrow OpNameC(j) \widehat{=}$
        $S(u, i, o)$                      BEGIN
END                                         $T(v, j, p)$
                                        WITHIN
                                            $P(i, j, u, v)$
                                        OUTPUT
                                            $E(u, v, o, p)$
                                        CONCEDES
                                            $C(u, v, o, p)$
                                        END
                                  END
```

Fig. 1. Syntax of simple retrenchment

The RETRENCHES clause (similarly to REFINES) makes visible the contents of the retrenched construct. The name spaces of the retrenched and retrenching constructs are disjoint, but admit an injection of (retrenched to retrenching) operation names, allowing extra independent dynamic structure in the retrenching machine.

The relationship between concrete and abstract state is fundamentally different *before* and *after* the operation. We model this by distinguishing between a strengthened before-relation between abstract and concrete states, and a weakened after-relation. Thus the syntax of the concrete operation *OpNameC* in N is precisely as in B, with the addition of the *ramification*, a syntactic enclosure of the operation. The precondition is strengthened by the WITHIN condition $P(u, v, i, j)$ which may change the balance of components between input and state. In the postcondition, the RETRIEVES clause $G(u, v)$ is weakened by the CONCEDES clause (the *concession*) $C(u, v, o, p)$, which specifies what the operation guarantees to achieve (in terms of after-state and output) if it cannot maintain the retrieve relation G. Since in simple retrenchment, the RETRIEVES clause gives no information about the relation between concrete and abstract output, we conjoin onto that clause an OUTPUT clause $E(u, v, o, p)$. This means that, should any change occur in the balance of components between abstract and concrete state and output, the change is fully described both for refining and non-refining behaviour. We will see how the need for the OUTPUT clause arises in calculating certain compositions.

Retrenchment has the same initialisation requirements as refinement, i.e. that the retrieve relation be established:

$$[Y(v)] \neg [X(u)] \neg G(u, v) \tag{1}$$

Output retrenchment is defined[1] as follows; for simple retrenchment, simply remove the E clause:

$$\begin{aligned} & I(u) \land G(u,v) \land J(v) \land P(i,j,u,v) \land \mathsf{trm}(T(v,j,p)) \\ & \;\Rightarrow \mathsf{trm}(S(u,i,o)) \land [T(v,j,p)]\neg\ [S(u,i,o)]\neg \\ & \qquad\qquad\qquad\qquad ((G(u,v) \land E(u,v,o,p)) \lor C(u,v,o,p)) \end{aligned} \tag{2}$$

From this point we will refer to "retrenchment" where we actually mean "output retrenchment" The following shorthand for (2) will be used:

$$S \lesssim_{G,P,E,C} T \tag{3}$$

2.3 Transitivity: Composing Output Retrenchments

For transitivity, it is straightforward to generalise the theorem for simple retrenchments [9, 22]. We assume as in section 2.2 that machine N RETRENCHES M, and further that machine O RETRENCHES N. Define machine O syntactically as a "lexicographic increment" on N, schematically replacing occurrences of *N,b,M,v,J,G,Y,p,j,T,P,E,C* in N by *O,c,N,w,K,H,Z,q,k,U,Q,F,D*, respectively. Thus operation S in machine M is retrenched by operation T in machine N (w.r.t. G, P, E, C), which is in turn retrenched by operation U in machine O (w.r.t. H, Q, F, D).
Theorem

$$\begin{aligned} & \text{If } S \lesssim_{G,P,E,C} T \text{ and } T \lesssim_{H,Q,F,D} U \text{ then...} \\ & I(u) \land \exists\, v \bullet (G(u,v) \land J(v) \land H(v,w)) \land K(w) \\ & \land \exists\, v, j \bullet (G(u,v) \land J(v) \land H(v,w) \land P(i,j,u,v) \land Q(j,k,v,w)) \\ & \land \mathsf{trm}(U(w,k,q)) \\ & \Rightarrow \mathsf{trm}(S(u,i,o)) \land [U(w,k,q)]\neg\ [S(u,i,o)]\neg \\ & \quad \big(\exists\, v,p \bullet (G(u,v) \land J(v) \land H(v,w) \land E(u,v,o,p) \land F(v,w,p,q)) \\ & \quad \lor \exists\, v,p \bullet (G(u,v) \land E(u,v,o,p) \land D(v,w,p,q)) \\ & \quad \lor \exists\, v,p \bullet (C(u,v,o,p) \land H(v,w) \land F(v,w,p,q)) \\ & \quad \lor \exists\, v,p \bullet (C(u,v,o,p) \land D(v,w,p,q))\big) \end{aligned} \tag{4}$$

[1] See [6] for a more general definition and full development of output retrenchment.

The result is intuitively satisfying, albeit weak in the explosion of disjuncts in the after-frame. The RETRIEVES clause $\exists\, v \bullet (G \wedge J \wedge H)$ combines component RETRIEVES clauses and intermediate invariant. The WITHIN clause $\exists\, v, j \bullet (G \wedge \cdots \wedge Q)$ combines all component before-state RETRIEVES and WITHIN constraints. The OUTPUT clause $E \wedge F$ combines the component OUTPUT clauses. The concession is a cross-product of component RETRIEVES clauses, their associated OUTPUT clauses, and concessions. It can be shown that the transitivity property associates.

2.4 Evolving Retrenchment

[20, 21] defined *evolving* retrenchment, which we recap very briefly. This variant uses the device of adding a "precision" model parameter to the retrieve relation, say G_α, and then exploiting that relation as a concession by varying the parameter. That is, $G_\alpha(u, v)$ is required to start the retrenchment step, and $G_\beta(u', v')$ is a possible outcome. This formulation can describe a typical precision-decay situation over a simulation, for example in simulating (real to floating point) a sequence of arithmetic steps. In general precision of representation may increase, decrease, or not change over a simulation step.

Whereas intuition here is to use weakened, or evolved RETRIEVES clauses as concessions, we define *evolving output retrenchment* in full generality[2]:

$$\begin{aligned}&I(u) \wedge G_\alpha(u,v) \wedge J(v) \wedge P_\alpha(i,j,u,v,A) \wedge \mathsf{trm}(T)(v,j)\\ &\Rightarrow \mathsf{trm}(S)(u,i) \wedge [T(v,j,p)]\neg\, [S(u,i,o)]\\ &\qquad\qquad \neg\, ((G_\beta(u,v) \wedge E_\beta(u,v,o,p)) \vee C_\beta(u,v,o,p)) \qquad (5)\end{aligned}$$

Although we use the parameterised retrieve relation device in this paper, the analysis uses (non-evolving) output retrenchment for reasons of simplicity and space.

3 Basic Monotonicity Results for Retrenchment

[22] showed the GSL constructors of B to be monotonic w.r.t. simple retrenchment, in the weak sense of the following theorem, which is extended here for output retrenchment.

Theorem

Retrenchment is monotonic w.r.t. the precondition, guard, bounded and unbounded choice constructors of B.

Case $\mid$ **:** Assume $S(u,i,o) \lesssim_{G,P,O,C} T(v,j,p)$. Assume Q is a predicate either on a common input variable or some shared external variable unaffected by either operation. Provided u, v, o, p are nonfree in Q, then

$$(Q \mid S) \lesssim_{G,P,O,C} (Q \mid T)$$

[2] As before in (2), omit the E clause for evolving simple retrenchment.

Case $\implies$: Assume $S(u,i,o) \lesssim_{G,P,O,C} T(v,j,p)$. Assume Q as for case $|$ above. Then

$$(Q \implies S) \lesssim_{G,P,O,C} (Q \implies T)$$

Case [] : Assume $S_1(u,i,o) \lesssim_{G,P_1,O_1,C_1} T_1(v,j,p)$ and $S_2(u,i,o) \lesssim_{G,P_2,O_2,C_2} T_2(v,j,p)$. Then

$$S_1 [] S_2 \lesssim_{G,P_1 \wedge P_2, O_1 \vee O_2, C_1 \vee C_2} T_1 [] T_2$$

Case @ : Now assume that $S(u,i,o,x) \lesssim_{G,P,O,C} T(v,j,p,x)$ for some fresh free external variable x (distinct from u, i, v, j, p). Then

$$@x \bullet S \lesssim_{G,P,O,C} @x \bullet T \tag{6}$$

Monotonicity is a fundamental requirement in order that retrenchment be a piecewise development method. The theorem is weak in the sense that the predicates added by the constructors are constrained to say little about the variables in the frame of the retrenchment; this work will contribute some more incisive structure to composite retrenchments.

We have omitted the parallel constructor from theorem (6); this structuring mechanism will be the subject of future work.

4 Example: Resource Allocation

For brevity in this paper we will use the abstract syntax of B. We use the following shorthand in a choice of guarded commands, where ELSE denotes the complement of the disjoined guards $\neg \exists z \bullet (P \vee Q \vee \cdots)$:

$$\begin{aligned}
&@z \bullet (P \implies S) \\
&[]\ @z \bullet (Q \implies T) \\
&\cdots \\
&[]\ \text{ELSE} \implies W
\end{aligned}$$

Our example is part of a resource allocation and management system in some distributed environment: resources must be acquired, scheduled for processing, and released in some prespecified way. We do not go into the nature of such scheduling or processing, simply focussing on the operation of allocation, or acquisition. What is of interest for our purposes is the degree of non-idealism in distributed, perhaps Internet, applications.

An ideal abstract description might require instantaneous allocation of a specified resource on the basis of a simple, decidable set membership test. In the real world of implementation, there are a number of constraints on acquisition of software resources in a distributed environment. Timeliness is first: how long will it take to find? Contractual issues arise in an open multi-vendor environment: do we have a contract to acquire repository X's resource? Quality of service, or trust: are we prepared to accept a resource that falls short of our requirements to a greater or lesser degree, in terms of functionality, security, capacity, or performance?

We prefer, by the principle of separation of concerns, not to clutter the abstract model with these constraints, but rather to describe them in a separate, more concrete model. In refinement practice, this is not possible: any exceptional, or constrained behaviour must be present in the abstract description, or be indistinguishable (through refinement steps) from behaviour in that description. This necessity in refinement to breach the separation principle is a key part of the case for the generalisation that is retrenchment [7, 9]. The retrenchment description of a design for the example incorporates such constraints and approximations of the idealised requirement. Our separation of concerns is reminiscent of the *superimposition* [18] and *superposition refinement* [4] approaches.

Figure 2 specifies part of an abstract resource management machine $RsAlloc$, with allocation operation $Alloc$. $SPEC$ is the set of all resource specifications, and $spec_u$ a function returning the specification for any given resource from environment RSS. State variable u records all resources already allocated. Operation $Alloc$ allocates any resource available in the RSS whose specification meets requirement rqt. The operation tests only for availability, somewhere in the world, of the resource, abstracting over the real-world constraints already mentioned.

```
MACHINE           $RsAlloc$
SETS              $RSS, SPEC$
CONSTANTS         $spec_u$
PROPERTIES        $spec_u : RSS \rightarrow SPEC$
VARIABLES         $u$
INVARIANT         $u \subseteq RSS$
INITIALISATION    $u := \varnothing$
OPERATIONS
    $Alloc(rqt) \mathrel{\widehat{=}}$
        $rqt \in SPEC \mid$
        $@x \bullet (x \in RSS - u \wedge spec_u(x) = rqt \Longrightarrow u := u \cup \{x\})$
        ELSE skip
    ...
END
```

Fig. 2. Resource allocation: specification

The specification of concrete machine $DRsAlloc$ in Fig. 3 contains the simple distributed resource allocation operation $DAlloc$. Abstract sets $SPEC, RSS$ are visible here. $DRSS$ is the concrete environment of distributed resources. $spec_v$ returns the specification of any given resource in $DRSS$ and a trust function tr is defined over $DRSS$. v is the concrete state variable, recording resources allocated. Retrieve relation $G_{\delta,n}$, defined by (7) below, relates concrete state to abstract.

DAlloc retrenches *Alloc*, being a step towards implementation by adding some of the real-world constraints. We assume (for simplicity) that trust ratings of 0, 1 or 2 can be assigned to each candidate resource available for allocation. Trust is a generic rating here, an abstraction over the compromises to be made over the availability, trustedness, or quality of the resources in question. Trust level 2 indicates that requirements are fully met, level 1 that they are partially met, and level 0 indicates that an appropriate resource is available, but that the degree to which it meets requirements is unknown. This concrete specification allocates level 2 and 1 resources from *DRSS* to v under separate guards, and skips for level 0 or no resource available[3]. The relevance of trust level 0 is seen if we imagine a concrete operation *DModifyTrust*, which may dynamically change the trust level of a resource in the environment. With no effect through the retrenchment on the abstract model, this operation retrenches (indeed, refines) abstract skip.

The syntactic shorthand above is the RETRIEVES clause, defined

$$\begin{aligned} G_{\delta,n}(u,v) \mathrel{\widehat{=}} \exists f \in v \rightarrowtail u \bullet \mathit{spec}_u \circ f = \mathit{spec}_v \\ \wedge \#(u - v) = \delta \\ \wedge \#(v \lhd \mathit{tr} \rhd \{1\}) = n \end{aligned} \tag{7}$$

The example is, informally speaking, an evolving retrenchment, where the concession is defined in terms of the retrieve clause $G_{\delta,n}(u,v)$, weakened by varying the meta-parameters δ, n. To understand the $G_{\delta,n}$ clause, briefly consider the pattern of resource allocation by the two operations. Abstractly, *Alloc* allocates if a resource is available, and otherwise skips. Concretely, *DAlloc* allocates a fully trusted resource if one is available, otherwise allocates a partially trusted resource if one is available, and otherwise skips. That is, *DAlloc* may (i)[4] exactly simulate the behaviour of *Alloc* (either in allocating a trusted resource, or not allocating when none are available), may (ii) approximate it in allocating a partially trusted resource, or may (iii) (more coarsely) approximate it in simply doing nothing. This approximating behaviour is recorded by parameters δ, the number of times allocation fails (iii), and n, the number of partially trusted resources allocated (ii).

Thus consider the representation of abstract resource set u by concrete set v. $G_{\delta,n}$ states that (a) there is an injection from v to u which uniquely identifies corresponding resource pairs, (b) each resource pair has matching specifications, (c) u has exactly δ more elements than v, and (d) that the number of partially trusted concrete resources is exactly n.

The retrenchment initialisation POB is clearly $[\mathit{Init}_C] \neg [\mathit{Init}_A] \neg G_{0,0}$: no resources exactly represents no resources. The operation retrenchment POB formalises the increasing approximation or decay over time of the (initially exact)

[3] We assume these guards are mutually disjoint and exhaustive. Formally, this would require the conjunction of each guard with the negation of each other guard, and so on, but we do not write this explicitly since to do so adds nothing to the discussion at this point. For the example application, this is admittedly simplistic.

[4] (i-iii) are annotations in Fig. 3

```
MACHINE           $DRsAlloc$
RETRENCHES        $RsAlloc$
SETS              $DRSS$
CONSTANTS         $spec_v, tr$
PROPERTIES        $spec_v : DRSS \rightarrow SPEC \wedge tr : DRSS \rightarrow \{0,1,2\}$
VARIABLES         $v$
INVARIANT         $v \subseteq DRSS$
RETRIEVES         $G_{\delta,n}(u,v)$
INITIALISATION    $v := \varnothing$
OPERATIONS
    $DAlloc(rqt) \mathrel{\widehat{=}}$
        BEGIN
        $rqt \in SPEC \mid$
        $@y \bullet (y \in DRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 2$
                                          $\Longrightarrow v := v \cup \{y\})$          (i)
        $@y \bullet (y \in DRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 1$
                                          $\Longrightarrow v := v \cup \{y\})$          (ii)
        ELSE skip                                                    (i,iii)
        WITHIN true
        OUTPUT true
        CONCEDES $G_{\delta,n}(u,v) \vee G_{\delta,n+1}(u,v) \vee G_{\delta+1,n}(u,v)$
        END
    ...
END
```

Fig. 3. Resource allocation: retrenchment

representation: each operation either maintains precision of representation, or weakens it by incrementing either δ or n. This is a weak and general statement, as one would expect from the disjunctive shape of the operation retrenchment obligation. Evidently, a sharper description of the relationship between the two models would detail the three retrenchment cases (i-iii) above, establishing $G_{\delta,n}(u,v)$, $G_{\delta,n+1}(u,v)$ or $G_{\delta+1,n}(u,v)$ respectively. Case (i) is clearly a refinement, in one layer of a partition of the retrenchment frame, where subsequent layers are domains of weakening, approximate relationships between the models (and ultimately, non-refining or exception layers).

4.1 Discussion

It is useful briefly to consider the example more fully. The resource allocation of operations $(Alloc, DAlloc)$ is clearly precursor to the main business of the example system, i.e. processing those resources in some fashion. Let us call the abstract/ concrete operation pair for processing $(Proc, DProc)$. This could represent initiation of communicating agents, or it could be exploitation of remote file and processing facilities. The system state u, v in the two models must be augmented with state components, say u_d, v_d, which will be dynamically transformed by this processing. This augmented state is subject to a retrieve relation augmented by a conjunct, say $G^d_\gamma(u_d, v_d)$ with parameter γ modelling the varying approximation of u_d by v_d. We expect the level of approximation under $(Proc, DProc)$ to remain constant where a fully trusted concrete resource is available to represent the abstract one in question, and to decay otherwise.

Since $DAlloc$ incorporates constraints of timely availability and quality of the desired resource, in general it is possible that a resource not available at some time t_0 may become available later, say at $t_1 > t_0$. Thus execution of $(Alloc, DAlloc)$ with input rqt (allocating abstract resource x, but no concrete resource) at t_0 will increment δ. Provided x is not processed before t_1, if we then execute $(\mathsf{skip}, DAlloc)$ with input rqt at t_1, we can guarantee improved approximation in postcondition $G_{\delta,n}(u, v)$ by decrementing δ. That is, we could treat $DAlloc$ as an evolving retrenchment of skip, with the following WITHIN clause. This clause is taken from $G_{\delta,n}$ to define the injection, establish the existence of a concretely unwitnessed abstract resource x of spec rqt, and the existence of a corresponding concrete witness resource for allocation:

$$\begin{aligned} P_{\delta,n}(u, v, rqt) \mathrel{\widehat{=}} \exists f : v \rightarrowtail u \bullet f \circ spec_u = spec_v \\ \wedge \#(u - v) = \delta \\ \wedge \# \operatorname{ran}(v \lhd tr \rhd \{1\}) = n \\ \wedge \exists x : RSS \bullet (x \in u - \operatorname{ran} f \wedge spec_u(x) = rqt) \\ \wedge \exists y : DRSS \bullet (y \notin v \wedge spec_v(y) = rqt \wedge tr(y) = 2) \end{aligned} \tag{8}$$

The retrenchment, with false concession and stronger than a refinement, is thus

$$\mathsf{skip} \lesssim DAlloc \ \ \text{w.r.t.} \ \ (G_{\delta,n} \wedge G^d_\gamma, P_{\delta,n} \longrightarrow G_{\delta-1,n} \wedge G^d_\gamma, \mathsf{false})$$

Whereas retrenchment is defined in terms of the classical forward simulation rule for refinement [16, 13] with its one-to-one mapping between operation steps, this example scenario brings to mind the more flexible m-to-n generalised refinement rules of Schellhorn et al [23]. In that formulation refinement is proved where m_i abstract steps correspond to n_i concrete steps between "synchronisation" points (where the retrieve relation is established), for two execution traces with $i \in \mathbb{N}$. One can imagine such flexible formulation of evolving retrenchment models for the example: we have seen defferred allocation of resources improving the $G_{\delta,n}$ approximation, and can imagine other scenarios. A $Proc$-step might correspond to a skipping $DProc$-step, where no corresponding concrete resource is available, decaying the G^d_γ conjunct. Once we perform the deferred concrete allocation by $\mathsf{skip} \lesssim DAlloc$, we might improve G^d_γ by deferred concrete execution $\mathsf{skip} \lesssim DProc$. A $Proc$-step might on the other hand correspond to a $DProc$ exception, decaying the G^d_γ, followed by concrete recovery and improvement of G^d_γ.

The idea of improvement of the level of approximation in evolving retrenchment is one that has emerged from our case study work [20, 21]. Although simple retrenchment is intrinsically a weakening relation between before- and after-frame, evolving retrenchment can model a strengthening retrieve relation by virtue of the parameterisation of the relation. In a control systems scenario [op.cit.], for example, improvement in approximation can happen after sensor failure is followed later by sensor recovery.

5 Decomposing a Retrenchment

We have seen that the retrenchment of Fig. 3 is weaker than we might wish. It represents a broad design view of the problem, relating case splits in abstract and concrete models through the retrenchment. This broad view includes all alternative behaviour cases, resulting in a weak and coarse-grained retrenchment picture, with a number of disjuncts in the concession. We need a systematic way to decompose this into a number of stronger-concession, thus finer-grain retrenchments that more sharply describe the partition (i-iii) of Fig. 3 of distinct relationships between the models.

Three approaches will be needed: (a) decomposing w.r.t. given concrete structure, (b) decomposing w.r.t. given abstract structure, and (c) decomposing w.r.t. given structure at both levels together.

For each of three decomposition results, a corresponding result enriched with nondeterministic choice will be given. This is both for generality as well as to support the example of section 4. Each of these six results is followed by a corollary which re-composes the retrenchment decomposition of that result. Note that we will make a distinction between bounded choice indexed over guards and substitutions, and nondeterministic (unbounded) choice. Although these two forms of choice are the same thing in the weakest precondition semantics, they are quite distinct in the world of the specification.

5.1 Decomposition - The Concrete Level

Where the concrete operation is one of guarded choice over an l-indexed collection of nested substitutions COp_l, we seek to decompose the single coarse-grained retrenchment

$$AOp \precsim_{G,P,O,C} \underset{l}{[]} (R_l \Longrightarrow COp_l) \tag{9}$$

into a finer-grained collection of retrenchments between the same two operations. For each choice branch in turn, given guard R_k, we seek a retrenchment

$$AOp \precsim_{G,P \wedge R_k,O_k,C_k} \underset{l}{[]} (R_l \Longrightarrow COp_l) \tag{10}$$

Each retrenchment here is intended to describe partition layer k of the abstract/concrete frame (the case concretely guarded by R_k) by strengthening the WITHIN and CONCEDES clauses to $P \wedge R_k$ and C_k respectively, where for each k we expect that $C_k \Rightarrow C$.

We show that (9) can be decomposed into (10) in two steps, by showing that (10) is the composition as per (4) of two retrenchments. The second of these is given by a lemma to show that a guarded command is retrenched by an indexed choice of guarded commands. The apparent increase in nondeterminism in this retrenchment is avoided by the assumption of mutual exclusivity of the guards. This is a strong assumption; the question of nondeterministically overlapping guards is addressed in section 5.2.

Lemma

For each k in turn, given Q_k (where k and l independently index the same family of substitutions), we have

$$R_k \Longrightarrow COp_k(\tilde{v}, \tilde{j}, \tilde{p}) \lesssim_{\tilde{v}=v, Q_k, \tilde{p}=p, \mathsf{false}} \underset{l}{[]} (R_l \Longrightarrow COp_l(v, j, p))$$
$$\text{where } Q_k \mathrel{\widehat{=}} \tilde{j} = j \wedge R_k(\tilde{v}, \tilde{j}) \wedge \bigwedge_{l \neq k} \neg\, R_l(\tilde{v}, \tilde{j}) \qquad (11)$$

Proof is by writing out and manipulating the retrenchment POB (2):

$$J(\tilde{v}) \wedge \tilde{v} = v \wedge J(v) \wedge \tilde{j} = j \wedge R_k(\tilde{v}, \tilde{j}) \wedge \bigwedge_{l \neq k} \neg\, R_l(\tilde{v}, \tilde{j}) \wedge \mathsf{trm}(\underset{l}{[]} R_l \Longrightarrow COp_l)$$
$$\Rightarrow \mathsf{trm}(R_k \Longrightarrow COp_k) \wedge [\underset{l}{[]} R_l \Longrightarrow COp_l]\,\neg[R_k \Longrightarrow COp_k]\,\neg(\tilde{v} = v \wedge \tilde{p} = p)$$

The RETRIEVES and WITHIN assumptions identify state and input in the two models. Also the mutual exclusivity of the guards ensures that this retrenchment is effectively an identity refinement. By the algebra of the GSL we have $\mathsf{trm}(\underset{l}{[]} R_l \Longrightarrow COp_l) \equiv \bigwedge_l (R_l \Rightarrow \mathsf{trm}(COp_l))$, and the consequent termination clause follows. The consequent simulation clause reduces to

$$\bigwedge_l (R_l \Rightarrow [COp_l]\neg\, (R_k \Rightarrow [COp_k]\neg\, (\tilde{v} = v \wedge \tilde{p} = p)))$$
$$\equiv \bigwedge_l (R_l \Rightarrow (R_k \wedge [COp_l]\,\neg[COp_k]\,\neg(\tilde{v} = v \wedge \tilde{p} = p)))$$

Syntactically, $R_k(\tilde{v}, \tilde{j})$ distributes through $COp_l(v, j, p)$ since they are over disjoint variable spaces. The mutual exclusivity premise Q_k ensures that R_l only holds for $l = k$, and the clause follows by identity refinement. **QED**

Theorem

Each retrenchment of index k may be transformed as follows:

$$AOp \lesssim_{G, P_k, O_k, C_k} R_k \Longrightarrow COp_k \vdash AOp(u, i, o) \lesssim_{G, P'_k, O_k, C_k} \underset{l}{[]} (R_l \Longrightarrow COp_l(v, j, p))$$
$$\text{where } P'_k \mathrel{\widehat{=}} P_k \wedge R_k \wedge \bigwedge_{l \neq k} \neg\, R_l \qquad (12)$$

Proof Here the abstract model is in variables u, i, o and the intermediate and lower models have variables as per lemma (11). Proof is by transitive composition of the left-hand retrenchment in (12) with that in the lemma, as per (4). This is straightforward; as again per (4) we have the composed postcondition clause in the form $(G \wedge O) \vee C$:

$$\begin{aligned}&\exists\, \tilde{v}, \tilde{p} \bullet (G(u, \tilde{v}) \wedge J(\tilde{v}) \wedge v = \tilde{v} \wedge O_k(u, \tilde{v}, o, \tilde{p}) \wedge \tilde{p} = p)\\ &\qquad \vee \exists\, \tilde{v}, \tilde{p} \bullet (C_k(u, \tilde{v}, o, \tilde{p}) \wedge \tilde{v} = v \wedge \tilde{p} = p)\end{aligned}$$

This gives composite WITHIN $\equiv G$, OUTPUT $\equiv O_k$, and CONCEDES $\equiv C_k$. We see here the need for output retrenchment: without the second-step OUTPUT clause $\tilde{p} = p$, the concrete p output would be completely unconstrained in the composite concession, which would be $\exists\, \tilde{p} \bullet C_k$. **QED**

Corollary

Given a decomposition of retrenchments (12), the following retrenchment holds[5]:

$$AOp \precsim_{G, \bigvee_k P'_k, \bigvee_k O_k, \bigvee_k C_k} \underset{l}{[]} (R_l \Longrightarrow COp_l) \tag{13}$$

Proof We use the facts that (i) if $A \Rightarrow B$ and $C \Rightarrow D$ then $A \vee C \Rightarrow B \vee D$ and (ii) the modal operator $[\;]\neg\,[\;]\neg$ is semidistributive over disjunction[6]. Take the disjunction over all k sets of hypotheses, infer the disjunction of the k consequents, and thus the composite consequent. **QED**

The example of section 4 includes nondeterministic choice, so the results of this section all need to be modified accordingly. Thus we have

Lemma

For each k in turn, given Q_k, we have

$$@z \bullet (R_k \Longrightarrow COp_k(\tilde{v}, \tilde{j}, z, \tilde{p})) \precsim_{\tilde{v}=v, Q'_k, \tilde{p}=p, \mathsf{false}} \underset{l}{[]} @z \bullet (R_l \Longrightarrow COp_l(v, j, z, p))$$

$$\text{where } Q'_k \mathrel{\widehat{=}} \tilde{j} = j \wedge \exists\, z \bullet R_k(\tilde{v}, \tilde{j}, z) \wedge \bigwedge_{l \neq k} \neg\, \exists\, z \bullet R_l(\tilde{v}, \tilde{j}, z) \tag{14}$$

Proof is as for lemma (11), with guard mutual exclusivity strengthened to include the choice variable z: given $\tilde{v}, \tilde{j}$, if any z satisfies R_k then *no* z satisfies any other guard R_l at $\tilde{v}, \tilde{j}$. The termination consequent follows as before. The simulation consequent reduces to

$$\begin{aligned}&\bigwedge_l \forall\, z \bullet (R_l(v, j, z) \Rightarrow (\exists\, \tilde{z} \bullet R_k(\tilde{v}, \tilde{j}, \tilde{z})\\ &\qquad\qquad \wedge [COp_l(v, j, z, p)]\, \neg [COp_k(\tilde{v}, \tilde{j}, \tilde{z}, \tilde{p})]\, \neg (\tilde{v} = v \wedge \tilde{p} = p)))\end{aligned}$$

[5] The alert reader may be confused when constrasting the disjunctive WITHIN hypothesis here with the conjunctive one in the apparently similar $[]$ result of (6). The latter result combines given retrenchments between *different* operation pairs, where this result combines given retrenchments over the *same* operation pair. If (6) is applied to two retrenchments over the same operation pair, then it does indeed apply with a disjunction of given WITHIN clauses.

[6] That is, $[T(v)]\, \neg [S(u)]\, \neg\, C(u, v) \vee [T]\, \neg [S]\, \neg\, D(u, v) \Rightarrow [T]\, \neg [S]\, \neg (C \vee D)$

The WITHIN clause ensures that the $\forall$-quantified expression is vacuously true for guards other than R_k , and any z satisfying $R_k(v, j, z)$ can be used as the existential witness $\tilde{z}$. **QED**

The following results are proved as before.

Theorem
Each retrenchment of index k may be transformed as follows:

$$\begin{aligned}
&AOp \lesssim_{G,P_k,O_k,C_k} @z \bullet (R_k \Longrightarrow COp_k) \\
&\vdash AOp(u, i, o) \lesssim_{G,P_k^{\forall},O_k,C_k} \underset{l}{[]} @z \bullet (R_l \Longrightarrow COp_l(v, j, z, p)) \\
&\text{where } P_k^{\forall} \mathrel{\widehat{=}} P_k \wedge \exists\, z \bullet R_k \wedge \bigwedge_{l \neq k} \neg\, \exists\, z \bullet R_l
\end{aligned} \tag{15}$$

Corollary
Given a decomposition of retrenchments (15), the following retrenchment holds:

$$AOp \lesssim_{G,\bigvee_k P_k^{\forall},\bigvee_k O_k,\bigvee_k C_k} \underset{l}{[]} @z \bullet (R_l \Longrightarrow COp_l) \tag{16}$$

5.2 Mutual Exclusivity Considered Harmful ?

The mutual exclusivity restriction of the above results seem very constraining. Particularly so, considering that retrenchment is an early-specification activity, intended to separate out concerns of architecture and information loss in the reification of a rich model down to a discrete, finite computer program. Nondeterminism is an intrinsic feature of abstract descriptions.

It is possible to weaken retrenchment (11) by allowing nondeterministically overlapping guards in the WITHIN clause, and weakening the concession. However, a rather baroque picture results which we choose not to pusue here, not least for reasons of space.

Methodologically, the assumption of mutual exclusivity will not prove to be a serious restriction. A nondeterministic guarded choice operation is always refinable to a deterministic one, by removing excess transitions. This amounts to refinement to an IF-THEN-ELSIF nesting, with precedence ordering of guards a design decision. A refinement is always expressible as a false-concession retrenchment, by including abstract termination as a WITHIN conjuct if necessary [9]. It is thus trivial to see that the following retrenchments compose, where $R_k' \Rightarrow R_k$:

$$\begin{aligned}
&AOp \lesssim_{G,P,O_k,C_k} R_k \Longrightarrow COp_k \quad , \\
&R_k \Longrightarrow COp_k \lesssim_{\tilde{v}=v,\tilde{j}=j \wedge R_k',\tilde{p}=p,\mathsf{false}} R_k' \Longrightarrow COp_k \\
&\vdash AOp \lesssim_{G,P \wedge R_k',O_k,C_k} R_k' \Longrightarrow COp_k
\end{aligned} \tag{17}$$

Thus guard-strengthening retrenchments compose seamlessly. We simply retrench away the nondeterminism until mutual exclusivity obtains, and then apply the relevant decomposition theorem. Since guard strengthening should be designed to eliminate nondeterminism, the overall operation guard ought not to strengthen; it should remain exhaustive, if the original overall guard is.

5.3 Decomposition - The Abstract Level

We now seek the complementary decomposition to that of section 5.1; i.e. to decompose the single retrenchment $\underset{l}{[]}\,(R_l \Longrightarrow AOp_l) \precsim_{G,P,O,C} COp$ into a finer-grained collection.

Lemma

For each k in turn, given Q_k, we have

$$\underset{l}{[]}\,(R_l \Longrightarrow AOp_l(u,i,o)) \precsim_{u=\tilde{u},P,o=\tilde{o},\mathsf{false}} R_k \Longrightarrow AOp_k(\tilde{u},\tilde{i},\tilde{o})$$
$$\text{where } P \mathrel{\widehat{=}} i = \tilde{i} \wedge \bigwedge_l (R_l \Rightarrow \mathsf{trm}(AOp_l)) \tag{18}$$

Proof is trivial: this is the conventional rewriting of a refinement as a retrenchment, with abstract termination as part of the WITHIN clause and a false concession. The simulation consequent reduces to

$$R_k \Rightarrow [AOp_k](\bigvee_l (R_l \wedge \neg\, [AOp_l]\neg\, (u = \tilde{u} \wedge o = \tilde{o})))$$

which is satisfied for $l = k$ as for lemma (11). The disjunctive expression here means that guard mutual exclusivity is not required. **QED**

Theorem

Each retrenchment of index k may be transformed as follows:

$$R_k \Longrightarrow AOp_k \precsim_{G,P_k,O_k,C_k} COp_k$$
$$\vdash \underset{l}{[]}\,(R_l \Longrightarrow AOp_l(u,i,o)) \precsim_{G,P'_k,O_k,C_k} COp(v,j,p)$$
$$\text{where } P'_k \mathrel{\widehat{=}} P_k \wedge \bigwedge_l (R_l \Rightarrow \mathsf{trm}(AOp_l)) \tag{19}$$

Proof , as before, is by transitive composition of the left-hand retrenchment in (19) with that in lemma (18), as per (4). **QED**

Corollary

Given a decomposition of retrenchments (19), the following retrenchment holds:

$$\underset{l}{[]}\,(R_l \Longrightarrow AOp_l) \precsim_{G,\bigvee_k P'_k,\bigvee_k O_k,\bigvee_k C_k} COp \tag{20}$$

Via the appropriate lemma, the analogue of (19) including nondeterministic choice is

Theorem

Each retrenchment of index k may be transformed as follows:

$$@z \bullet (R_k \Longrightarrow AOp_k) \precsim_{G,P_k,O_k,C_k} COp_k$$
$$\vdash \underset{l}{[]}\,@z \bullet (R_l \Longrightarrow AOp_l(u,i,o,z)) \precsim_{G,P^{\forall}_k,O_k,C_k} COp(v,j,p)$$
$$\text{where } P^{\forall}_k \mathrel{\widehat{=}} P_k \wedge \bigwedge_l \forall\, z \bullet (R_l \Rightarrow \mathsf{trm}(AOp_l)) \tag{21}$$

Corollary
Given a decomposition of retrenchments (21), the following retrenchment holds:

$$\underset{l}{[]}\ @z \bullet (R_l \Longrightarrow AOp_l) \lesssim_{G, \bigvee_k P_k^{\forall}, \bigvee_k O_k, \bigvee_k C_k} COp \tag{22}$$

5.4 Decomposition - Both Levels Together

The two sections above show how to decompose a coarse-grained retrenchment by exploiting concrete and abstract model structure respectively. An even more finely grained picture should be obtainable by considering all such structure simultaneously. That is, given an abstractly decomposed retrenchment (19) achieving $(G \wedge O) \vee C_k$ under assumptions H_k, and a concretely decomposed retrenchment (12) between the same operations achieving $(G \wedge O) \vee C_l$ under assumptions H_l, we seek a retrenchment achieving $(G \wedge O) \vee (C_k \wedge C_l)$ under assumptions $H_k \wedge H_l$[7]. Unfortunately, the modal simulation operator $[\,] \neg [\,] \neg$ is not conjunctive. It is necessary to perform the full decomposition from first principles, as the application of three transitive composition steps (4) combining those of the two above theorems.

We omit proofs in this section because of their similarity with previous proofs.
Theorem
Each of a collection of retrenchments, with abstract and concrete models indexed separately by k and m, can be transformed as follows:

$$\begin{aligned} & AR_k \Longrightarrow AOp_k \lesssim_{G, P_{km}, O_{km}, C_{km}} CR_m \Longrightarrow COp_m \\ & \vdash \underset{l}{[]}\,(AR_l \Longrightarrow AOp_l(u,i,o)) \lesssim_{G, P'_{km}, O_{km}, C_{km}} \underset{n}{[]}\,(CR_n \Longrightarrow COp_n(v,j,p)) \\ & \text{where } P'_{km} \mathrel{\widehat=} P_{km} \wedge \bigwedge_l (AR_l \Rightarrow \mathsf{trm}(AOp_l)) \wedge CR_m \wedge \bigwedge_{n \neq m} \neg\, CR_n \end{aligned} \tag{23}$$

We note the following points about this result. This fine-grained collection of retrenchments fully exploits the structure in both models, meeting the goal discussed at the beginning of this section. Usually we will have $P_{km} \Rightarrow AR_k \wedge CR_m$, i.e. each retrenchment layer will be defined within the subdomain where both abstract and concrete guards hold. Guards may overlap nondeterministically in the abstract model, and, should they do so in the concrete model, the latter can be "retrenched down" seamlessly to the required mutual exclusivity of guards.
Corollary
Given a decomposition of retrenchments (23), the following retrenchment holds:

$$\underset{l}{[]}\,(AR_l \Longrightarrow AOp_l) \lesssim_{G, \bigvee_k P'_{km}, \bigvee_k O_{km}, \bigvee_k C_{km}} \underset{n}{[]}\,(CR_n \Longrightarrow COp_n) \tag{24}$$

Note that where the corollary is indexed over k, it is of course applicable over various disjunctions: over all abstract guards l, over all concrete guards n, or over all guards at both levels together.

[7] Note that here the two retrenchments share the OUTPUT clause O.

Finally, the analogue of (23) and (24) including nondeterministic choice is
Theorem
Each of a collection of retrenchments, with abstract and concrete models indexed separately by k and m, can be transformed as follows:

$$
\begin{aligned}
&@z \bullet (AR_k \Longrightarrow AOp_k) \precsim_{G,P_{km},O_{km},C_{km}} @z \bullet (CR_m \Longrightarrow COp_m)\\
&\vdash \underset{l}{[]}\, @z \bullet (AR_l \Longrightarrow AOp_l(u,i,o)) \precsim_{G,P^{\forall}_{km},O_{km},C_{km}} \underset{n}{[]}\, @z \bullet (CR_n \Longrightarrow COp_n(v,j,p))\\
&\text{where } P^{\forall}_{km} \mathrel{\widehat{=}} P_{km} \wedge \bigwedge_{l} \forall\, z \bullet (AR_l \Rightarrow \mathsf{trm}(AOp_l)) \wedge\\
&\qquad\qquad \exists\, z \bullet CR_m \wedge \bigwedge_{n \neq m} \neg\, \exists\, z \bullet CR_n
\end{aligned}
\tag{25}
$$

Corollary
Given a decomposition of retrenchments (25), the following retrenchment holds:

$$
\underset{l}{[]}\, @z \bullet (AR_l \Longrightarrow AOp_l) \precsim_{G,\bigvee_k P^{\forall}_{km},\bigvee_k O_{km},\bigvee_k C_{km}} \underset{n}{[]}\, @z \bullet (CR_n \Longrightarrow COp_n) \tag{26}
$$

6 Decomposing The Example

We apply (25,26) to the example. Modulo comments in section 5 and footnote 3 about mutual exclusivity of guards, we read guards AR_1 and $AR_2 \equiv \neg\, AR_1$ from Fig. 2 and CR_1, CR_2, CR_3 from Fig. 3. We have $P_{1m} \equiv AR_1$ for $m = 1\,..\,3$, since the existence of an abstract resource is a necessary condition for there to exist a concrete approximating resource. We have no retrenchments for $k = 2, m = 1..2$, since we cannot (in this model) relate abstract resource non-existence to concrete resource existence. We have $P_{23} \equiv \neg\, AR_1$. All simple guarded substitutions here of form $R \Longrightarrow Op$ always terminate. Finally, we have $G \equiv G_{\delta,n}$ and for all indices $O_{km} \equiv \mathsf{true}$.

Thus for input to theorem (25) we have four component retrenchments, say r_{km}, for $k = 1\,..\,2$ and $m = 1\,..\,3$. r_{11} achieves concession $C_{11} \equiv \mathsf{false}$, since $G_{\delta,n}$ is guaranteed in this case. r_{12} achieves concession $C_{12} \equiv G_{\delta,n+1}$. r_{13} achieves concession $C_{13} \equiv G_{\delta+1,n}$. r_{23} achieves concession $C_{23} \equiv \mathit{false}$, since $G_{\delta,n}$ is guaranteed in this case where both models skip.

Applying (25) we have four fine-grained retrenchments r_{km} of *Alloc* to *Dalloc*, each qualified by WITHIN clause P'_{km} combining relevant abstract and concrete guard predicates, and CONCEDES clause C_{km}. Corollary (26) combines these retrenchments to recover the original coarse-grained retrenchment of Fig. 3.

7 Conclusion

The observant reader will by now have noticed the sleight-of-hand in the approach of this paper: formally speaking, our “decomposition” is in fact composition masquerading as decomposition. The decomposition problem we have ostensibly addressed, in its full generality, is "given a retrenchment r from abstract AOp to concrete COp, can we find two retrenchments r_1 from AOp to some intermediate IOp and r_2 from IOp to COp such that $r_1 \mathbin{⨾} r2 = r$?". We have in reality sidestepped this question by starting with a general syntactic form for the composite retrenchment, and a strong intuition about the finer-grained component retrenchments, in order to prove some simple *composition* results.

These will suffice for our purposes, because the generality of the composite form "covers most of the bases" required by practical specification work. Also, the natural (and traditional) approach to design is to deal with any case-split structure in the model under consideration, and to worry about its relationship to adjacent models later. A significant base not covered is of course the parallel substitution $||$; this remains for future work.

The obvious universality question related to the question above arises: "What are the 'best', i.e. weakest-WITHIN and strongest-CONCEDES component retrenchments r_1 and r_2?". Further work in the categorical style of the integration of refinement and retrenchment [5, 17] is indicated here. The suggestion of [22] of a lattice theory of retrenchment (over the collection of all WITHIN clauses that satisfy a given retrenchment, similarly all CONCEDES clauses) also needs pursuing to this end.

In the case of the more general decomposition problem where our approach cannot be applied, transitivity of retrenchment (4) gives some guidance: for the composite retrenchment r to follow from the decomposition $r_1 \mathbin{⨾} r2$ we must have

$$\begin{aligned}
&\mathit{RETRIEVES}(r) \equiv \mathit{RETRIEVES}(r_1 \mathbin{⨾} r2)\\
&\land\ \mathit{WITHIN}(r) \Rightarrow \mathit{WITHIN}(r_1 \mathbin{⨾} r2)\\
&\land\ \mathit{OUTPUT}(r) \equiv \mathit{OUTPUT}(r_1 \mathbin{⨾} r2)\\
&\land\ \mathit{CONCEDES}(r_1 \mathbin{⨾} r2) \Rightarrow \mathit{CONCEDES}(r)
\end{aligned} \tag{27}$$

References

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

[2] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998.

[3] R.J.R. Back and M. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35:921–949, 1998.

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

[5] R. Banach. Maximally abstract retrenchments. In *Proc. IEEE ICFEM2000*, pages 133–142, York, August 2000. IEEE Computer Society Press.

[6] R. Banach and C. Jeske. Output retrenchments, defaults, stronger compositions, feature engineering. 2002. submitted, http://www.cs.man.ac.uk/ banach/some.pubs/Retrench.Def.Out.pdf.

[7] 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.

[8] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. Technical Report Report UMCS-99-3-2, University of Manchester Computer Science Department, 1999.

[9] R. Banach and M. Poppleton. Sharp retrenchment, modulated refinement and simulation. *Formal Aspects of Computing*, 11:498–540, 1999.

[10] R. Banach and M. Poppleton. Retrenchment, refinement and simulation. In J. Bowen, S. King, S. Dunne, and A. Galloway, editors, *Proc. ZB2000*, volume 1878 of *LNCS*, York, September 2000. Springer.

[11] R. Banach and M. Poppleton. Engineering and theoretical underpinnings of retrenchment. submitted, http://www.cs.man.ac.uk/ banach/some.pubs/Retrench.Underpin.pdf, 2002.

[12] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Requirements Engineering Journal*, 2002. 22pp., to appear.

[13] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

[14] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[15] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[16] C.A.R. Hoare, Jifeng He, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

[17] C. Jeske and R. Banach. Reconciling retrenchments and refinements. *submitted*, 2002.

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

[19] M. Poppleton and R. Banach. Retrenchment: extending the reach of refinement. In *ASE'99: 14th IEEE International Conference on Automated Software Engineering*, pages 158–165, Florida, October 1999. IEEE Computer Society Press.

[20] M. Poppleton and R. Banach. Retrenchment: Extending refinement for continuous and control systems. In *Proc. IWFM'00*, Springer Electronic Workshop in Computer Science Series, NUI Maynooth, July 2000. Springer.

[21] M. Poppleton and R. Banach. Controlling control systems: An application of evolving retrenchment. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *Proc. ZB2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, Grenoble, France, January 2002. Springer.

[22] M.R. Poppleton. *Formal Methods for Continuous Systems: Liberalising Refinement in B.* PhD thesis, Department of Computer Science, University of Manchester, 2001.

[23] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *JUCS*, 7:952–979, 2001.