Structuring Retrenchments in B by Decomposition

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
[2] Department of Computer Science, Manchester University,
Manchester M13 9PL, UK,
banach@cs.man.ac.uk

Abstract. Simple retrenchment is briefly reviewed in the B language of J.-R. Abrial [1] as a liberalization of classical refinement, for the formal description of application developments too demanding for refinement. This work initiates the study of the structuring of retrenchment-based developments in B by decomposition. A given coarse-grained retrenchment relation between specifications is decomposed into a family of more fine-grained retrenchments. The resulting family may distinguish more incisively between refining, approximately refining, and non-refining behaviours. Two decomposition results are given, each sharpening a coarse-grained retrenchment within a particular syntactic structure for operations at concrete and abstract levels. A third result decomposes a retrenchment exploiting structure latent in both levels. The theory is illustrated by a simple example based on an abstract model of distributed computing, and methodological aspects are considered.
Keywords decomposition, formal methods, refinement, retrenchment, structuring.

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 liberalization of refinement. Early work [7, 8] motivated such a liberalization in terms of the problems applying refinement to "difficult" applications such as radiation dosimetry and magnetohydrodynamics. Such problem domains include infinite sets or properties, or models in continuous mathematics or classical physics, which do not relate in a simple way to the finite, discrete computer. A simple example is the impossibility of refining element addition/subtraction on an infinite set to a finite one. Classical refinement also prohibits I/O type change between what are conventionally known as the *abstract* and *concrete* models. By weakening the abstraction relation over the operation step, retrenchment allows concrete non-simulating behaviour to be described in, and related back to corresponding abstract behaviour. Concrete I/O may have different type to the abstract counterpart, and moreover the retrenchment relation may accomodate fluidity between state and I/O components across the development step from abstract to concrete model. [17, 20] developed a calculus of retrenchment in B, proved transitivity, and showed all primitive operators of the B generalized Substitution Language (GSL) to be monotonic with respect to retrenchment. [8, 9] explored the landscape between refinement, simulation and retrenchment. [5] addressed the integration of refinement and retrenchment from a methodological perspective. [18, 6] present two generalizations, evolving and output retrenchment respectively. The latter of these is used in this paper.

To provide application motivation for this liberalizing enterprise, a number of more substantial case studies of retrenchment have been presented. [19] gives a retrenchment model of the conventional approximating design step from an analogue linear control system to its discrete-time zero-order hold counterpart. Telephony feature interaction is a major application area characterised by requirements features which are in general mutually inconsistent and not simply composable; the utility of retrenchment was shown in a simple feature interaction case study [10]. This case study was developed to show how application domain knowledge could strengthen a retrenchment description [12].

Various methodological issues need to be addressed to support the effective use of retrenchment in practice: the choice of abstractions and designs for understandable and mechanisable retrenchment proof 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. [11] makes some commentary on the first issue, and the second is becoming better understood [5]. [20] gave the monotonicity results on which to base a study of composability. This paper is concerned with the fourth issue, the decomposition of a given retrenchment.

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 family 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 coarse-grained retrenchment. In this work we employ the output form of retrenchment [Op. cit.], which provides equipment for certain algebraic issues that arise.

The paper proceeds as follows. Section 2 briefly recalls the B GSL. Section 3 recaps syntactic and semantic definitions for retrenchment in GSL, extending them for output retrenchment. We extend the transitivity theorem of [20, 17] to provide the composition of two output retrenchments [Op. cit.]. Section 4 presents a running example to motivate the discussion, and demonstrates how the disjunctive shape of the retrenchment obligation is coarser and less descriptive than may desirable for certain purposes. Section 5 gives a number of retrenchment decomposition results. Three syntactic patterns are given for decomposing a single retrenchment into a 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 The B Language of generalized Substitutions

The B language was defined by [1] and is disseminated by textbooks such as [21]. B has as its central construct the *generalized substitution*: $[S]R$ (more conventionally written $wp(S, R)$) describes the weakest precondition under which program S is guaranteed to terminate satisfying postcondition R. generalized substitution distributes over conjunction and is monotonic w.r.t. implication. Programs (in general nondeterministic) are written using constructors inspired by Dijkstra's Guarded Command Language, called the generalized Substitution Language (GSL). The basic operation is the *simple substitution* (assignment, in procedural programming terms). For replacement of free variable x in formula R by expression E we write $[x := E]R$. The remaining simple constructors of B are axiomatised (for unbounded choice z is nonfree in R; this is written $z \setminus R$):

$$\begin{array}{rll} [\mathsf{skip}]R \equiv R & \text{skip} & \\ [P \mid S]R \equiv P \wedge [S]R & \text{precondition} & \\ [S \,[]\, T]R \equiv [S]R \wedge [T]R & \text{bounded choice} & \\ [P \Longrightarrow S]R \equiv P \Rightarrow [S]R & \text{guard} & \\ [@z \bullet S]R \equiv \forall\, z \bullet [S]R z \setminus R & \text{unbounded choice} & (1) \end{array}$$

The precondition constructor explicitly strengthens the termination set, guard strengthens the feasibility set, bounded choice gives demonic nondeterministic choice between two operations, and unbounded choice a universally quantified demonic choice over all operations indexed on some (external) variable.

The action of an operation S, with state variable (list) x, on predicate $R(x)$ can be expressed in the following *normalised* form, where P is a predicate in variable x, Q is a predicate in variables x and x' (x' distinct from x):

$$[S]R \equiv P \wedge \forall x' \bullet (Q \Rightarrow [x := x']R) \tag{2}$$

This decomposition into predicates P and Q is unique (modulo logical equivalence of predicates), and these are called $\mathsf{trm}(S)$ (termination: before-states from which S is guaranteed to terminate) and $\mathsf{prd}_x(S)$ (before-after transition) respectively. Theorem (2) interprets S as a predicate transformer: from initial state x, S establishes R precisely when S terminates at x and every x' reachable from x under S satisfies R. These predicates can be explicitly defined:

$$\mathsf{trm}(S) \equiv [S]\mathsf{true} \qquad\qquad \mathsf{prd}_x(S) \mathrel{\widehat{=}} \neg\, [S](x' \neq x)$$

The abstract syntax of the GSL is complemented by the concrete syntax of the Abstract Machine Notation (AMN), which includes constructs for modular structuring. The unit of modularity is the *machine*, which contains inter alia a state *variable* (list), an *invariant* predicate expressing type and other required state constraints, an *initialisation*, and a set of *operations*, which are expressed in terms of state, input and output variables. Fig. 1 shows an abstract machine and a refinement. The latter is a derivative construct: invariant clause $J(u, v)$ provides local variable type and constraint information, and the retrieve relation from concrete to abstract state variable.

MACHINE	$M(a)$	REFINEMENT	N
		REFINES	M
VARIABLES	u	VARIABLES	v
INVARIANT	$I(u)$	INVARIANT	$J(u, v)$
INITIALISATION	$X(u)$	INITIALISATION	$Y(v)$
OPERATIONS		OPERATIONS	
$\quad S(u, i, o) \mathrel{\widehat{=}} \cdots$		$\quad T(v, i, o) \mathrel{\widehat{=}} \cdots$	
END		END	

Fig. 1. B machine and refinement syntax

The basic machine consistency proof obligations are *initialisation* (the initialisation establishes the invariant) and *operation consistency* (given invariant and operation termination, then the operation establishes the invariant):

$$[X]I \qquad\qquad I \wedge \mathsf{trm}(S) \Rightarrow [S]I \tag{3}$$

The refinement proof obligations in B are equivalent to the classical forward simulation rules and are expressed as follows. Two abstract machines M and N are defined on state spaces u and v respectively, with a total relation J from v to u. There is a bijection between the operations of M and N (say, every operation

S of machine M corresponds to exactly one operation T of N). If for every such pair (S, T) the following proof obligations (POBs) hold, then M is refined by N (written $M \sqsubseteq N$): *initialisation refinement* (for every concrete initial step, there is an abstract initial step that establishes the retrieve relation) and *operation refinement* (for any concrete step of T, there is some abstract step of S that establishes the retrieve relation):

$$
\begin{aligned}
&[Y]\neg\,[X]\neg\,J \\
&I \wedge J \wedge \mathsf{trm}(S) \Rightarrow [T]\neg\,[S]\neg\,J
\end{aligned}
\tag{4}
$$

3 Retrenchment

In its simple form, retrenchment weakens 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 effectively in the case of refining behaviour.

Output retrenchment [6] improves matters by having an additional output conjunct specifically to cover this case. The ensuing tradeoff between additional syntactic complexity in the retrenchment and ease of use in discussing structural and algebraic aspects of retrenchment proves to be a big win technically. This paper will work with output retrenchment.

3.1 Retrenchment Defined

Figure 2 defines the syntax of output retrenchment in B, based on Fig. 1; it differs only from the simple form in the addition of the OUTPUT clause. 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, local invariant $J(v)$, initialisation $Y(v)$, and operation *OpNameC* as wrapper for $T(v, j, p)$, a substitution with input j and output p. The RETRENCHES clause (replacing REFINES) makes visible the lexical environment of the retrenched construct. The RETRIEVES clause names the retrieve relation, from which the local invariant conjunct $J(v)$ has been separated syntactically into the INVARIANT clause. 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. This is reasonable in the light of the likelihood of machine N having a lower level and more detailed structure, possibly incorporating aspects that have no place in a cleaner, higher level model.

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. 2. Syntax of output retrenchment

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 $P(i,j,u,v)$ 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, where the latter expresses the global relationship between abstract and concrete state variables. Since in simple retrenchment, the RETRIEVES clause gives no information about the relationship between concrete and abstract output, we conjoin to that clause an OUTPUT clause $E(u,v,o,p)$ in the postcondition. 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:

$$[Y(v)]\,\neg[X(u)]\,\neg\, G(u,v) \tag{5}$$

Output retrenchment is defined[1] by all of the above together with the following operation proof obligation:

$$\begin{aligned}
&I(u) \wedge G(u,v) \wedge J(v) \wedge P(i,j,u,v) \wedge \mathsf{trm}(T(v,j,p)) \\
&\Rightarrow \mathsf{trm}(S(u,i,o)) \wedge [T(v,j,p)]\neg\, [S(u,i,o)]\neg \\
&\qquad\qquad\qquad ((G(u,v) \wedge E(u,v,o,p)) \vee C(u,v,o,p))
\end{aligned} \tag{6}$$

It is easy to see that retrenchment generalizes refinement[2]: choose $P \mathrel{\widehat{=}} \mathsf{trm}(S)$, $E \mathrel{\widehat{=}} \mathsf{true}$ and $C \mathrel{\widehat{=}} \mathsf{false}$ in (6). From this point we will refer to "retrenchment" where we actually mean "output retrenchment", and will use the following shorthand for (6): $S \lessapprox_{G,P,E,C} T$.

3.2 Composing Output Retrenchments

It is straightforward to generalize the composition theorem for simple retrenchments [8, 20]. We assume as in section 3.1 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 If $S \lessapprox_{G,P,E,C} T$ and $T \lessapprox_{H,Q,F,D} U$ then $S \lessapprox_{GJH,PQ,EF,CD} U$

$$\begin{aligned}
\text{where } GJH &= \exists\, v \bullet (G(u,v) \wedge J(v) \wedge H(v,w)) \\
PQ &= \exists\, v,j \bullet (G(u,v) \wedge J(v) \wedge H(v,w) \wedge P(i,j,u,v) \wedge Q(j,k,v,w)) \\
EF &= \exists\, v,p \bullet (E(u,v,o,p) \wedge F(v,w,p,q)) \\
CD &= \exists\, v,p \bullet (G(u,v) \wedge E(u,v,o,p) \wedge D(v,w,p,q)) \\
&\quad \vee \exists\, v,p \bullet (C(u,v,o,p) \wedge H(v,w) \wedge F(v,w,p,q)) \\
&\quad \vee \exists\, v,p \bullet (C(u,v,o,p) \wedge D(v,w,p,q))
\end{aligned} \tag{7}$$

The result is intuitively satisfying. The RETRIEVES clause GJH combines component RETRIEVES clauses and intermediate invariant. The WITHIN clause PQ combines all component before-state RETRIEVES and WITHIN constraints to ensure common v, j witnesses can be found for all the constituent terms. The OUTPUT clause EF combines the component OUTPUT clauses. The concession comes from a distribution of the disjunctions in the conjunction of the two postconditions $((G \wedge E) \vee C) \wedge ((H \wedge F) \vee D)$ over the conjunction, with the term corresponding to the combined RETRIEVES clause removed. It can be shown that the above definition of composition of retrenchments is associative.

[1] For simple retrenchment, simply remove the E clause.

[2] In its I/O modulated form [8], which permits I/O type change.

4 Example: Resource Allocation

For brevity we use the abstract syntax of B GSL for operation bodies rather than the more verbose concrete B AMN syntax. We adopt the shorthand of an 'ELSE' clause in a choice of guarded commands, where ELSE denotes the complement of disjoined guards $\neg \exists z \bullet (P \vee Q \vee \cdots)$ in the following expression:

$$@z \bullet (P \Longrightarrow S) \; [] \; @z \bullet (Q \Longrightarrow T) \; \cdots \; [] \; \text{ELSE} \Longrightarrow W$$

Our example is a partial abstract model of a resource allocation and management system in a distributed environment: resources must be acquired, scheduled for processing, and released. In a centralised environment, functional requirements such as resource acquisition can be viewed as atomic until we descend to a fairly low level of abstraction, because the centralised scheduler in effect has all the aspects involved under its direct control. In a distributed environment, this is much less the case because of ignorance about what is going on at remote locations. Methodologically, we seek to separate concerns of functionality from those of distribution. Thus the abstract description models instantaneous allocation or not of a specified resource on the basis of a simple test.

In the concrete world, a number of lower level issues intrude to influence the success or otherwise of allocation. We could mention timeliness, contractual issues, quality of service — these relating to the requesting system's knowledge of the providing system's capabilities at that time — as well as the simple availability of what is requested. The situation is simplified here by modelling even the distributed allocation as an atomic process (i.e. described within a single syntactic entity), but entertaining nonetheless the possibility of outcomes displaying different degrees of success, in line with what can happen in real distributed systems.

We separate specification concerns by restricting consideration of such issues to the concrete level. This raises the question of how the abstract and concrete levels relate to each other – ideally, by refinement. But whether this is true or not depends strongly on how the extra concrete features fit together with the concrete description of the purely abstract model. If all goes well then the situation can be elegantly captured within a superposition refinement [16, 4]. But all is by no means guaranteed to go well. To address these less convenient situations, which are nevertheless prone to occur in practice, the authors introduced retrenchment, with its more forgiving operation proof obligation.

The example provides a simple vehicle for the contribution of this work – we restrict ourselves to little more than the distinct case splits in the two operation models to illustrate our contribution – and further motivates the utility of retrenchment.

Figure 3 specifies part of an abstract resource management machine *RsAlloc*, with allocation operation *Alloc*. *SPEC* is the set[3] of all resource specifications, and $spec_u$ is a static function returning the specification for any given resource

[3] In this model resource specifications are unstructured, abstract entities, elements of the set *SPEC*, which in a real specification would be defined elsewhere.

from the universe of allocatable resources RSS. The state variable u records all resources already allocated. Operation *Alloc* allocates any resource not yet allocated in the set RSS whose specification meets the requirement rqt of the *Alloc* call. The operation tests only for availability 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	

$Alloc(rqt) \mathrel{\widehat{=}}$
$\quad rqt \in SPEC \mid$
$\quad @x \bullet (x \in RSS - u \wedge spec_u(x) = rqt \Longrightarrow u := u \cup \{x\})$
$\quad []$ ELSE skip
$\ldots$
END

Fig. 3. Resource allocation: specification

Concrete machine *CRsAlloc* in Fig. 4 is the concrete counterpart of machine *RsAlloc*. In particular it contains the simple distributed resource allocation operation *CAlloc* (distributed only to the extent that the atomic operation exhibits some characterisitics normally associated with genuinely distributed allocation operations, in line with the remarks above).

Thus $CRSS$ is the set of concrete distributed resources and $spec_v$ returns the specification of any given concrete resource in $CRSS$, with values in $SPEC$. There is also a trust function tr, defined over $CRSS$. This yields an abstract measure of the quality of the resource acquired in the case of successful allocation. v is the concrete state variable, recording resources allocated.

CAlloc retrenches *Alloc* 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 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. Thus the concrete operation allocates level 2 and 1 resources from $CRSS$ to v under separate guards, and skips for level 0 or no resource available[4]. Output res from *Calloc* reports the degree of success in matching an abstract allocation *Alloc*. There is no matching output from *Alloc*; this shows

[4] 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

the I/O mutability possible in retrenchment. For simplicity we choose not to exploit such mutability further in this discussion.

Trust level 0 resources are strictly redundant here since we never do anything with them. However, the utility of trust level 0 is clear if we consider an additional concrete operation *CModifyTrust*, which can dynamically change the trust level of a resource in the environment in response to information received. Such an operation would have no abstract counterpart in line with the possibility admitted by the retrenchment formalism. We retain trust level 0 but do not discuss *CModifyTrust* further.

MACHINE	$CRsAlloc$
RETRENCHES	$RsAlloc$
SETS	$CRSS, RESULT$
CONSTANTS	$spec_v, tr$
PROPERTIES	$spec_v : CRSS \to SPEC \wedge tr : CRSS \to \{0,1,2\} \wedge$ $RESULT = \{Succ, Partial, Fail\}$
VARIABLES	v
INVARIANT	$v \subseteq CRSS$
RETRIEVES	$G_{\delta,n}(u,v)$
INITIALISATION	$v := \varnothing$
OPERATIONS	

$res \longleftarrow CAlloc(rqt) \mathrel{\widehat=}$
BEGIN
$rqt \in SPEC \mid$
$@y \bullet (y \in CRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 2$
$\qquad \Longrightarrow v := v \cup \{y\}) \mid\mid res := Succ$ (i)
$[]\ @y \bullet (y \in CRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 1$
$\qquad \Longrightarrow v := v \cup \{y\}) \mid\mid res := Partial$ (ii)
$[]$ ELSE $res := Fail$ (i,iii)
WITHIN true
OUTPUT true
CONCEDES $G_{\delta,n+1}(u,v) \vee G_{\delta+1,n}(u,v)$
END
...
END

Fig. 4. Resource allocation: retrenchment

Retrieve relation $G_{\delta,n}$, defined by (8) below, relates concrete states to abstract ones. It is parameterised by δ, quantifying the maximum acceptable difference in numbers of resources allocated at the two levels, and n, the maximum number of partially-trusted resources that can be concretely allocated.

$$G_{\delta,n}(u,v) \mathrel{\widehat=} \exists f \in v \rightarrowtail u \bullet spec_u \circ f = v \lhd spec_v \\ \wedge \#(u - v) \leq \delta \wedge \#(v \lhd tr \rhd \{1\}) \leq n \qquad (8)$$

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.

To understand the RETRIEVES clause $G_{\delta,n}$, consider the pattern of resource allocation by the two operations. Abstractly, *Alloc* allocates if a resource is available, or otherwise skips. Concretely, *CAlloc* allocates a fully trusted resource if one is available, or allocates a partially trusted resource if one is available, and otherwise skips. That is, *CAlloc* may (i)[5] exactly simulate the behaviour of *Alloc* (either in allocating a trusted resource, or not allocating), may (ii) approximate it in allocating a partially trusted resource, or may (iii) more coarsely approximate it by simply doing nothing. This approximating behaviour is recorded by parameters δ, the maximum number of times allocation may fail, and n, the maximum number of partially trusted resources that may be allocated.

$G_{\delta,n}$ thus states that (a) there is a total injection from v to u which uniquely identifies corresponding resource pairs, (b) each resource pair shares a specification, (c) u has at most δ more elements than v, and (d) that the number of partially trusted concrete resources is at most n. The operation retrenchment POB formalises a varying representation: each allocation either (i) maintains precision of representation in retrieve relation $G_{\delta,n}$, or weakens it by establishing as concession either (ii) $G_{\delta,n+1}$ or (iii) $G_{\delta+1,n}$.

5 Decomposing Retrenchment

The retrenchment of Fig. 4 represents a first-cut design view of the problem, relating the abstract to the concrete allocation operation without exploiting the case structure at either level. It is thus a coarse-grained retrenchment picture, with a number of disjuncts in the postcondition, and no *a priori* guarantee as to which might be established. A systematic way is required to decompose this single retrenchment into a family of stronger-concession, thus finer-grained retrenchments. These will more sharply describe the partition (i-iii) of Fig. 4 of distinct relationships between the models.

Three approaches, shown schematically in Fig. 5, will be needed to decompose (a) w.r.t. given concrete structure, (b) w.r.t. given abstract structure, and (c) w.r.t. given structure at both abstract and concrete levels together. Approach (a), for example, in theorem 12 needs a k-indexed family of "component" retrenchments $AOp \lesssim R_k \implies COp_k$ to be read from the specification (and proved). Each such retrenchment is composed as per (7) with the corresponding retrenchment $R_k \implies COp_k \lesssim \underset{l}{[]} (R_l \implies COp_l)$ given by lemma 11. Approach (b) is the converse of (a), and for (c) a three-step composition is required. To simplify matters, in each case the retrenchments linking a guarded command to a bounded choice of guarded commands are in fact I/O modulated refinements [Op.Cit.]. We will also see the algebraic necessity for the output, rather than simple, form of retrenchment in the proof of theorem (12).

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

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

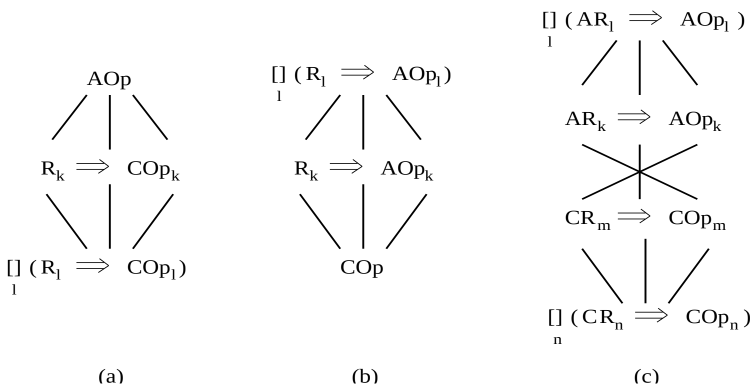

Fig. 5. Three patterns for decomposing a retrenchment

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.

5.1 Decomposition - The Concrete Level

We seek a retrenchment decomposition as per Fig. 5(a), where the abstract operation is atomic, and 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

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

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

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

Each retrenchment in the family is intended to describe partition part 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. For each k we expect that $C_k \Rightarrow C$. The specifier will be free to choose C_k and O_k, which are expected to arise naturally from the specification.

We show that the single retrenchment (9) can be decomposed into the family (10) in two steps, by showing that, for each k, (10) is the composition as per (7) of two retrenchments. The second of these is given by a lemma to show that a guarded command is retrenched[6] by an indexed choice of guarded commands. The apparent increase in nondeterminism in this retrenchment is avoided by the

[6] Lemma (11) is in fact an I/O-modulated refinement, as mentioned in section 3.1.

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

$$R_k(\tilde{v}, \tilde{j}) \Longrightarrow COp_k(\tilde{v}, \tilde{j}, \tilde{p}) \lesssim_{\tilde{v}=v,\, Q_k,\, \substack{\tilde{v}=v\wedge\\ \tilde{p}=p},\, \mathsf{false}} \underset{l}{[]}\, (R_l(v, j) \Longrightarrow COp_l(v, j, p))$$

$$\text{where } Q_k \mathrel{\widehat{=}} \tilde{j} = j \wedge R_k(\tilde{v}, \tilde{j}) \wedge \bigwedge_{l \neq k} \neg\, R_l(\tilde{v}, \tilde{j}) \tag{11}$$

Proof is by writing out and manipulating the retrenchment POB (6):

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

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

The RETRIEVES and WITHIN assumptions identify state and input respectively in the two models. 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 (12) decomposes a retrenchment in terms of given concrete case structure:

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

$$AOp \lesssim_{G, P_k, O_k, C_k} R_k \Longrightarrow COp_k \vdash AOp(u, i, o) \lesssim_{G, P'_k, O_k, C_k} \underset{l}{[]} (R_l \Longrightarrow COp_l(v, j, p))$$

$$\text{where } P'_k \mathrel{\widehat{=}} P_k \wedge R_k \wedge \bigwedge_{l \neq k} \neg\, R_l \tag{12}$$

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 (7). This is straightforward; as again per (7) we have the composed postcondition clause in the form $(G \wedge O) \vee C$. Note that two of the three C disjuncts collapse to false because the second-step concession is false:

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

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**

The corollary recomposes the original coarse-grained retrenchment (9):

Corollary Given a decomposition of retrenchments (12), the following holds:

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

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[7]. 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

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

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

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

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

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 decomposition and recomposition results (15 - 16) with nondeterministic choice are proved as before.

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

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

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

Corollary Given a decomposition of retrenchments (15), the following 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 is at first sight 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 make retrenchment (11) more expressive by allowing nondeterministically overlapping guards in the WITHIN clause, and weakening the concession from false. 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, as shown in section 3.1. 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

Here we seek a retrenchment decomposition as per Fig. 5(b), where the abstract operation is one of guarded choice over an l-indexed collection of nested substitutions AOp_l, and the concrete operation is atomic. This is the complementary decomposition to that of section 5.1; i.e. to decompose the single retrenchment $[]\,(R_l \Longrightarrow AOp_l) \lesssim_{G,P,O,C} COp$ into a finer-grained family. Proofs are omitted in this section; the first proof straightforwardly rewrites a refinement as a retrenchment, and the rest are as before.

Lemma For each k in turn, given Q_k, we have

$$\underset{l}{[]}\,(R_l \Longrightarrow AOp_l(u,i,o)) \lesssim_{u=\tilde{u}, P, o=\tilde{o}, \mathsf{false}} R_k \Longrightarrow AOp_k(\tilde{u}, \tilde{i}, \tilde{o})$$

$$\text{where } P \mathrel{\widehat{=}} i = \tilde{i} \wedge \bigwedge_l (R_l \Rightarrow \mathsf{trm}(AOp_l)) \tag{18}$$

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

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

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

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

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

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

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

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

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

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, as per Fig. 5(c). That is, given an abstractly decomposed retrenchment family (19) achieving $(G \wedge O) \vee C_k$ under assumptions H_k, and a concretely decomposed retrenchment family (12) between the same operations achieving $(G \wedge O) \vee D_l$ under assumptions H_l, we seek a retrenchment family (indexed on k and l) achieving $(G \wedge O) \vee (C_k \wedge D_l)$ under assumptions $H_k \wedge H_l$[8]. 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 (7) combining those of

[8] Note that here the two retrenchment families share the OUTPUT clause O.

theorems (12), (19). We omit proofs in this section because of their similarity with previous proofs.

Theorem Each of a family of retrenchments indexed on k, m, where abstract guards are k-indexed and concrete guards m-indexed, can be transformed as follows:

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

We note the following points about this result. This fine-grained family 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, as indicated in section 5.2.

Corollary Given a decomposition of retrenchments (23), the following holds:

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

Note that where the corollary is indexed over k (all abstract guards), it is of course applicable over m (all concrete guards), and indeed over k, m (all guards at both levels).

Finally, the analogue of (23) and (24) including nondeterministic choice is

Theorem Each of a family of retrenchments, with abstract and concrete models indexed separately by k and m, can be transformed as follows:

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

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

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

6 Decomposing The Example

We apply (25), (26) to the example retrenchment in order to extract a finer-grained family. Modulo comments in section 5.2 and footnote 4 about mutual exclusivity, from Figures 3, 4 we have guards

$AR_1 \mathrel{\widehat{=}} x \in RSS - u \wedge spec_u(x) = rqt$	abstract, alloc
$AR_2 \mathrel{\widehat{=}} \neg\, AR_1$	abstract, no-alloc
$CR_1 \mathrel{\widehat{=}} y \in CRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 2$	(i) concrete, alloc-tr=2
$CR_2 \mathrel{\widehat{=}} y \in CRSS - v \wedge spec_v(y) = rqt \wedge tr(y) = 1$	(ii) concrete, alloc-tr=1
$CR_3 \mathrel{\widehat{=}} \neg\, (CR_1 \vee CR_2)$	(i,iii) concrete, no-alloc

We employ the annotations (i - iii) from Fig. 4. We have $P_{1m} \mathrel{\widehat{=}} AR_1 \wedge CR_m$ for $m = 1 \mathinner{..} 3$, for the retrenchment of abstract allocation by cases (i, ii, iii) respectively. We have $P_{23} \mathrel{\widehat{=}} AR_2 \wedge CR_3$ for case (i) with no allocation at either level. There are no retrenchments for $k = 2, m = 1 \mathinner{..} 2$ in this model since we cannot relate abstract non-allocation to concrete allocation. All simple guarded substitutions here of form $R \Longrightarrow Op$ always terminate. Finally, we have $G \equiv G_{\delta,n}$ and for all indices $O_{km} \mathrel{\widehat{=}} \mathsf{true}$.

Thus for input to theorem (25) we have four component retrenchments between single-guarded commands, say r_{km} with WITHIN clauses P_{km} etc., for $k = 1 \mathinner{..} 2$ and $m = 1 \mathinner{..} 3$. r_{11} represents the refining case (i) of allocation at both levels, and thus achieves G with concession $C_{11} \mathrel{\widehat{=}} \mathsf{false}$. r_{12} achieves either G or concession $C_{12} \mathrel{\widehat{=}} G_{\delta,n+1}$, in the approximating case (ii) of trust 2 concrete allocation. r_{13} achieves either G or concession $C_{13} \mathrel{\widehat{=}} G_{\delta+1,n}$, in the case (iii) of no concrete allocation approximating abstract allocation. r_{23} achieves G with concession $C_{23} \mathrel{\widehat{=}} \mathsf{false}$, in the (i) case where both models fail to allocate.

Applying (25) produces four fine-grained retrenchments r_{km} of *Alloc* to *Dalloc*, each qualified by RETRIEVES G, WITHIN P'_{km} combining relevant abstract and concrete guard predicates, OUTPUT true and CONCEDES C_{km}. Corollary (26) combines these retrenchments to recover the original coarse-grained retrenchment of Fig. 4. We see that two of the four retrenchments produced are in fact refinements, and the other two are each finer (have stronger concessions) than the original.

It is worth noting that there are further, finer decompositions possible of the example. By strengthening the guards with state information about approximation levels (e.g. how close $\#(u - v)$ is to δ), it is possible to tease out more retrenchments with stronger postconditions (e.g. $\neg\, G \wedge G_{\delta+1,n}$ when $\#(u - v) = \delta$ in WITHIN).

7 Conclusion

We have considered the problem of a first cut "coarse-grained" design of the abstract-to-concrete operation transformation $AOp \lesssim COp$ as a retrenchment r, say, and its decomposition into a finer-grained family of retrenchments $\{r_i\}$.

An approach of "decomposition by composition" was taken: using a general syntactic form for each of for the two operations, each member of the family was constructed as the transitive composition per theorem (7) of retrenchments via suitable intermediate operation fragments. Each component retrenchment in the family is stronger than the composite retrenchment in the sense that it delivers a stronger concession, i.e. guarantees more in the postcondition. Each component is also more restrictive in having a stronger WITHIN clause; moreover the WITHIN clauses of the family effectively partition the joint before-state/I/O frame of the composite retrenchment.

The general syntactic form used inductively covers all operations that may be specified using the primitive abstract syntax of B. We have not mentioned the precondition constructor, which factors through the theory trivially; in practice it is only used at the top level of an operation specification to type input parameters. Thus the results "cover most of the bases" required by practical specification work. We merely claim "most" since we have yet to address the parallel substitution || of B: this is the means by which multiple variables (and nontrivial transformations of such through retrenchment) and their dynamics are described.

Methodologically speaking, this work supports a natural (and traditional) approach to design. That is, one model at one abstraction level is developed, including choice, case-split and other structure. The next, more concrete model is then developed, bearing in mind the refinement or retrenchment abstraction to be used. Only then is the relation between the models examined; the retrenchment case, as we have seen, affording the option of further decomposition to a suitable granularity.

Finally, we briefly consider the theoretical decomposition question in its full generality: "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 $r_1 \mathbin{;} r_2 \Rrightarrow r$?". Transitivity of retrenchment (7) gives some guidance: for the composite retrenchment r to be a logical consequence of the decomposition $r_1 \mathbin{;} r_2$ we must have

$$\begin{aligned} &RETRIEVES(r) \equiv RETRIEVES(r_1 \mathbin{;} r_2) \wedge WITHIN(r) \Rrightarrow WITHIN(r_1 \mathbin{;} r_2) \\ &\wedge OUTPUT(r) \equiv OUTPUT(r_1 \mathbin{;} r_2) \wedge CONCEDES(r_1 \mathbin{;} r_2) \Rrightarrow CONCEDES(r) \end{aligned} \tag{27}$$

The obvious universality problem related to the full decomposition question above arises: "What are the 'best', i.e. weakest-WITHIN and strongest-concession component retrenchments r_1 and r_2?". Further work in the categorical style of the integration of refinement and retrenchment [5] is indicated here. The suggestion of [20] 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.

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. Sharp retrenchment, modulated refinement and simulation. *Formal Aspects of Computing*, 11:498–540, 1999.

[9] 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.

[10] R. Banach and M. Poppleton. Model based engineering of specifications by retrenching partial requirements. In *Proc. MBRE-01: IEEE Workshop on Model-Based Requirements Engineering*, University of California, San Diego, November 2001. IEEE Press.

[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*, 8(2), 2003. 22pp.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] S. Schneider. *The B-Method*. Palgrave Press, 2001.