Requirements Validation by Lifting Retrenchments in B

Michael Poppleton
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
mrp@ecs.soton.ac.uk

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

Abstract

Simple retrenchment is briefly reviewed in the B specification language of J.-R. Abrial [1] as a liberalization of classical refinement, for the formal description of application developments too demanding for refinement. The looser relationships allowed by retrenchment between adjacent models in the development process may capture some of the requirements information of the development. This can make requirements validation more difficult to understand since the locus of requirements should be the models, and not their interrelationships, as far as possible. Hence the universal construction of [6], originally proposed for simple transition systems, is reformulated in B, in order to "lift" a given retrenchment conceptually, thus retracting such requirements information back to the level of abstraction of the abstract, ideal model. Examples demonstrate the cognitive value of retracting requirements to the abstract level, articulated in a well-understood formal language. This is also seen to yield a more understandable way of comparing alternative retrenchment designs. Some new B syntax in the pre- and postcondition style is presented to facilitate expression of the lifted requirements.

1. Introduction

Refinement has for a long time been a well respected method for developing concrete models and executable code from (not necessarily executable) specifications which are expressed using mathematics that is unconcerned with the actual capabilities of any real computing device [4, 5, 13, 15]. The robustness and reliability of the refinement technique come from the relatively strong, precise conditions that have to hold before a refinement can be asserted between two models of a system. A *retrieve*, or *abstraction* relation is defined between the data types of the first, more abstract model, and the second, more concrete and algorithmic model. The retrieve relation defines how the concrete data type represents the abstract one. Essentially, refinement requires that any concrete behaviour *simulates* any corresponding abstract behaviour by preserving the retrieve relation. Therein lies a problem, because many situations in which developers of high consequence and complex systems might wish to faithfully use refinement to express (and verify) their development route, feature a series of models whose desired relationships (desired that is, from an engineering perspective) do not satisfy the exacting conditions for refinement. Examples of such model combinations are easily found in modern distributed systems, e.g. [16], and have been examined in a retrenchment setting: [12] considers the construction of feature-oriented specifications with inconsistency, and [19] is a control engineering example which describes the inherently approximate and varying relationship between continuous- and discrete-time models. In the face of this, developers either abandon any attempt to use refinement, contenting themselves with less formal but more flexible (and unfortunately less rigorous) techniques, or they deliberately choose to fail to meet the criteria for refinement in various ways, perhaps by working with models that represent reality less faithfully than might otherwise be desired.

Retrenchment [9, 10, 11, 8] was introduced to make available a formal technique for addressing such situations, which while lacking the guarantees that refinement offers (since one cannot have both the rigour of refinement and the flexibilty that developers crave), is nevertheless still formal, and thus allows some formal statements to be made about scenarios in which such a level of rigour would otherwise be unavailable. Retrenchment achieves this goal by weakening the main proof obligation (PO) of refinement in a way that enables it to express relevant facts about the more general situations envisaged. The weakening involves introducing extra relations into the PO (which, for refinement is roughly speaking expressed via a single relation, the *retrieve* relation). These are the *within* and *concedes* relations, whose job it is to accomodate the lack of precise adherence to the refinement PO criteria.

Of key importance is the need for retrenchment and refinement to work smoothly together, so that the power of the developer's arsenal of tools amounts to more than the sum of its parts, and this paper is concerned with some aspects of this interworking. In [6], it is shown that an arbitrary retrenchment, from an abstract model *Abs* to a concrete model *Conc*, can be factored into firstly a retrenchment from *Abs* to a model *Univ* that preserves the level of abstraction of *Abs* (in a certain well defined sense), and secondly a refinement from *Univ* to *Conc*, that bridges the abstraction gap of the original retrenchment, such that the composition of the two recovers the original retrenchment from *Abs* to *Conc*. This achieves a useful separation of concerns given the great flexibility of the retrenchment notion.

To state this more precisely, for a putative development step to be captured via a retrenchment from *Abs* to *Conc*, one can start by concentrating on the requirements[1] that the development step is intended to address, without worrying about issues around levels of abstraction. It will not be necessary to avoid requirements information appearing in the within and concedes relations of the retrenchment, which give the semantic relationship between the models. One can use the construction in [6] to factorize the given retrenchment. The factorization lifts this requirements information to the original level of abstraction and includes it explicitly in the syntax of the constructed operation specification in *Univ*. It is well understood that requirements can reside in a refinement relation: for example, [11] shows how refinement of a specification by reduction of nondeterminism can introduce behavioral properties not explicitly specified. The requirement of defined approximate representation of abstract (real-world) by concrete (computer) state, carried by evolving retrieve and concedes relations in retrenchment, was demonstrated in arithmetic [18] and control engineering [19] applications.

In this paper, we reinterpret the factorization construction of [6], originally developed for simple transition systems, within the context of the B-Method. This means paying attention to a number of matters of technical detail in which the two approaches differ. For instance the meaning of the retrieve relation needs to be reexamined, and the distinctive role of the B-Method's machine invariants must be incorporated. The meaning of a transition step needs to be interpreted within B.

Section 2 gives a brief introduction to the syntax, semantics and refinement method of B. In Section 3, the rationale for and definition of retrenchment is reviewed, as a liberalization of refinement. Section 4 gives some new syntax for B specification in a pre-postcondition style. The lifting construction of [6] is reformulated in the B terminology in Section 5. In Section 6, we consider two small example retrenchments which we use to illustrate the requirements validation ideas above. Section 7 concludes.

1 Our use of the term "requirements" is general: it comprises, and does not distinguish between, functional and nonfunctional requirements, implementation and environmental constraints.

2. Specification and refinement in B

The B language was defined by the B-Book [1] and is disseminated by textbooks such as [21]. A wide-spectrum language covering specification and programming, it is supported by full-lifecycle verification toolkits such as *Atelier B* [2], and has been instrumental in successful safety-critical system developments such as signalling on the Paris Metro [14].

B has as its central construct the *generalized substitution*: $[S]R$ (read "S establishes R", and more conventionally written $wp(S,R)$) is the weakest precondition under which program S is guaranteed to terminate satisfying postcondition R. Specifications, in the style of nondeterministic programs, 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 does not appear bound, i.e. is *nonfree* in R; written $z \setminus R$):

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

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 *normal form* theorem, where P is a predicate in variable x, Q is a predicate in variables x and x' (x' distinct from x):

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

This decomposition into predicates P and Q is unique (modulo logical equivalence of predicates). P is called trm_S (the *termination* predicate: before-states from which S is guaranteed to terminate) and we define two transition predicates (called the *transition* and *step* predicates respectively):

$$\mathsf{trm}_S \mathrel{\widehat{=}} P \qquad \mathsf{prd}_S \mathrel{\widehat{=}} P \Rightarrow Q \qquad \mathsf{stp}_S \mathrel{\widehat{=}} P \wedge Q \quad (3)$$

Theorem (2) gives a conventional, relational expression of the effect of a GSL operation $[S]$. It interprets predicate transformer S: from initial state x, S establishes R precisely when S terminates at x and every x' reachable from x under S satisfies R. The B-Book gives $\mathsf{trm}, \mathsf{prd}$ equivalences (straightforwardly extensible to stp) for the GSL constructors. Conversely, since

$$\begin{aligned}[S]R &\equiv \mathsf{trm}_S \wedge \forall x' \bullet (\mathsf{prd}_S \Rightarrow [x := x']R) \\ &\equiv \mathsf{trm}_S \wedge \forall x' \bullet (\mathsf{stp}_S \Rightarrow [x := x']R) \end{aligned} \tag{4}$$

constructors of the GSL can be specified in terms of arbitrary predicates: in this paper we will interpret selected predicates as trm and stp.

The abstract syntax of the GSL is complemented by the concrete syntax of the Abstract Machine Notation (AMN), which includes constructs for modular structuring. Fig. 1 gives the B syntax for an abstract machine and a refining machine. We see that the unit of modularity is the *machine*, which contains inter alia a state *variable* (list) u, an *invariant* predicate $Inv_A(u)$ expressing type and other required state constraints, an *initialisation* $Init_A(u)$, and a set of *operations* $Op_A(u,i,o)$, which are expressed in terms of state, input and output variables.

Refinement is the classic model-based formal method for the verifiably correct stepwise construction of a program from its specification; see [13] for a thorough historical account. Refinement in B (Fig. 1) constitutes a change to more concrete, program-oriented state data types, and the addition of algorithmic structure to the operations. We give a generalization of standard B refinement, called *IO-filtered refinement* from [10], in which the types of IO elements may change across the refinement step: machine *Abs* with state $u \in \mathrm{U}$ has local invariant Inv_A; refining machine *Conc* has state $w \in \mathrm{W}$ and local invariant Inv_C. State w represents state u through *retrieve* relation K. For the operation refinement of Op_A to Op_C, *within* relation $R_{Op}(i,k)$ relates concrete inputs $k \in \mathrm{K}$ to abstract inputs $i \in \mathrm{I}$, and *output* relation $V_{Op}(o,q)$ relates outputs $o \in \mathrm{O}$ to $q \in \mathrm{Q}$.

MACHINE	$Abs(a)$	MACHINE	$Conc$
VARIABLES	u	VARIABLES	w
INVARIANT	$Inv_A(u)$	INVARIANT	$Inv_C(w)$
INITIALISATION	$Init_A(u)$	RETRIEVES	$K(u,w)$
OPERATIONS		INITIALISATION	$Init_C(w)$
$o \longleftarrow Op_A(u,i,o) \cdots$		OPERATIONS	
END		$q \longleftarrow Op_C(w,k,q) \cdots$	
		WITHIN	$R_{Op}(i,k)$
		OUTPUT	$V_{Op}(o,q)$
		END	

Figure 1. B machine and refinement syntax

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

$$[Init_A]Inv_A \qquad Inv_A \wedge \mathsf{trm}_{Op_A} \Rightarrow [Op_A]Inv_A \tag{5}$$

The refinement POs generalise the classical forward simulation rules [13] and are expressed as follows. Two abstract machines *Abs* and *Conc* have a total onto retrieve relation $K(u,w)$. For every refining operation pair (Op_A, Op_C) there are total onto relations $R_{Op}(i,k)$ (within) and $V_{Op}(o,q)$ (output). If for every such pair the following POs (6) hold, then *Abs* is refined by *Conc* (written $Abs \sqsubseteq Conc$): *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 Op_C, there is some abstract step of Op_A that establishes the retrieve and output relations). As shorthand we write $Op_A \sqsubseteq_{K,R_{Op},V_{Op}} Op_C$. The unfamiliar $[\cdots]\neg[\cdots]\neg$ syntax expands to a $\forall - \exists$ sequence of quantifiers as per (2):

$$\begin{aligned} &[Init_C]\neg\,[Init_A]\neg\,K \\ &Inv_A \wedge K \wedge Inv_C \wedge \mathsf{trm}_{Op_A} \wedge R_{Op} \\ &\ \Rightarrow \mathsf{trm}(Op_C) \wedge [Op_C]\neg\,[Op_A]\neg\,(K \wedge V_{Op}) \end{aligned} \tag{6}$$

As mentioned above, this work generalizes standard B refinement slightly, not least in allowing IO types to change. Moreover, standard B refinement has some redundancy: the usual REFINEMENT construct defines a machine in the product space $Abs \times Conc$. Our definition makes the refinement a distinct machine with access to abstract frame u, i, o. Since the construction we exploit is itself located in the product space $Abs \times Conc$, use of the standard definition in defining the refinement of *Univ* to *Conc* would result in unacceptable proliferation of product constructions.

3. From refinement to retrenchment

Simple retrenchment weakens the retrieve relation between two levels of abstraction: loosely speaking, it strengthens the precondition, weakens the postcondition, and introduces mutability between state and IO at the two levels. The precondition is strengthened by a *within* clause between abstract and concrete before-state and input, restricting the joint frame to the precise region where the retrenchment relation is to be posited. Compare this more expressive within clause P in (7) with R_{Op} in (6), which simply relates input types. The postcondition comprises a disjunction between a retrieve relation between after-state at both levels, where refining behaviour is described, and a *concession* relation between after-state and output at both levels. This concession (where non-refining concrete behaviour is related back to abstract behaviour) is the vehicle in the postcondition for describing state-output mutability. Before-state and input may also appear in the concession as “history” information. The disjunctive concession C in (7) can be contrasted with the conjunctive R_{Op} in (6), which simply relates output types.

The semantic definition of retrenchment is by analogy with refinement (6). The initialisation proof obligation is the same as for refinement. The retrenchment of Op_A by Op_C w.r.t. retrieve G, within P and concession C (as shorthand we write $Op_A \lesssim_{G,P,C} Op_C$), is defined by the following PO:

$$\begin{aligned} &Inv_A(u) \wedge G(u,w) \wedge Inv_C(w) \wedge P(i,k,u,w) \wedge \mathsf{trm}(Op_C(w,k,q)) \\ &\Rightarrow \mathsf{trm}(Op_A(u,i,o)) \wedge [Op_C(w,k,q)]\neg\, [Op_A(u,i,o)] \\ &\qquad \neg\, (G(u,w) \vee C(u,w,o,q;\ i,k,u_0,w_0)) \end{aligned} \tag{7}$$

It is easy to see that retrenchment generalizes refinement: choose $P \mathrel{\widehat{=}} \mathsf{trm}(Op_A)$ and $C \mathrel{\widehat{=}} \mathsf{false}$ in (7).

The retrenchment PO (7) should be seen as a relation between models that methodologically extends and generalizes the refinement relation. Given an abstract model, and a more concrete, implementation-oriented one, a refinement can usually be posited over some region of the joint before-state-input frame (which region may be so small as to be practically useless). It is always possible to posit a retrenchment - a true-concession retrenchment always holds over within clause $\mathsf{trm}(Op_A)$ - and we may call this the *vacuous* retrenchment. Hence (unless the two models are completely unrelated) it will be possible to posit informative retrenchments over suitable selected within-concession pairs. The PO states that, starting in the within region, *either* the retrieve relation will be satisfied, i.e. simulation reestablished, *or* the weaker representation of the concession relation over the joint after-state-output frame will be established. This is a more qualified and approximate statement than a refinement, but certainly more informative than no refinement at all! Finally, there may be a within region where we can guarantee establishment of some concession *without* considering the retrieve relation: this possibility goes beyond simple retrenchment to the proposal of *general* retrenchment [7], which we do not discuss further here.

Thus the retrenchment approach gives a richer and more graduated picture of the relation between two models than pure refinement: a small, refining region, is surrounded (perhaps, included in) a number of larger regions where defined retrenchments hold. All are included in the universal (true-within) region of the vacuous retrenchment.

We demonstrate the syntax of retrenchment with a simple example, rather than giving a full syntactic definition, which is available in e.g. [10]. Fig. 2 specifies the *exception* retrenchment of some ideal arithmetic, addition, of the natural numbers D-NAT to some finite natural type M-NAT. For brevity, we make some simplifications and mix standard abstract and concrete syntaxes, of B GSL (1), and machines and refinement (Fig. 1) respectively. The retrenchment of abstract Op_A by concrete Op_C is defined syntactically by the addition to the text of Op_C of a *ramification* comprising the WITHIN and CONCEDES (concession) clauses.

IO mutability is demonstrated by representing abstract state summand b by concrete input parameter bb (WITHIN), and the addition of an exception status output *resp* (CONCEDES). Ideal addition is simulated in the concrete model provided the sum of the arguments does not overflow, otherwise it provides exception processing with the relationship of such processing to the abstract model recorded in the concession clause.

```
MACHINE         Divine
VARIABLES       a, b
INVARIANT
    a, b ∈ D-NAT
INITIALISATION ···
OPERATIONS
    DAdd ≙
        a := a + b
    ···
END

MACHINE         Mundane
RETRENCHES      Divine
VARIABLES       aa
INVARIANT       aa ∈ M-NAT
RETRIEVES       aa = a
INITIALISATION ···
OPERATIONS
    resp ⟵ MAdd(bb) ≙
    bb ∈ M-NAT |
        aa + bb ≤ Max
            ⟹ aa := aa + bb || resp := Ok
     [] aa + bb > Max
            ⟹ resp := Fail
    WITHIN bb = b
    CONCEDES C(a, b, aa, resp;  bb, a₀, b₀, aa₀)
    ···
END
```

Figure 2. Retrenchment of natural addition

The within and concession relations, like the retrieve relation, are matters of design choice. Here, representation of abstract state b by concrete input bb requires their identification in WITHIN. One possible concession design for clause C in Fig. 2 is

$$\begin{aligned} &C_{Add_1}(a,b,aa,resp,bb,a_0,b_0,aa_0) \mathrel{\widehat{=}} \\ &\qquad aa_0 + bb \le Max \wedge aa = a \wedge resp = Ok \\ &\qquad \vee\ aa_0 + bb > Max \wedge aa = a - b \wedge resp = Fail \end{aligned} \tag{8}$$

which is a typical *exception* concession pattern, total on the before-state-input frame with a partition of two regions. The effect of concrete behaviour on the abstraction relation between aa and a, b is described exactly, *a priori* in terms of this partition. The first clause of each disjunct is effectively a guard. The second clause gives the effect of the step on the abstraction relation: either the concrete addition does not overflow, and simulation (and the retrieve relation) is maintained, or it overflows, and simulation fails. C_{Add_1} specifies that in the latter case aa stays unchanged ($aa = a - b$). The third clause in each disjunct specifies the output status information to be returned.

Concession C_{Add_1} exploits history information in order to be applicable over a wide ($bb = b$) within region. To give a more graduated picture of the relationship between the models in Fig. 2 as discussed above, it is in principle possible to define both a simpler, less incisive retrenchment and a refinement between *DAdd* and *MAdd*. A retrenchment that does not distinguish between refining and conceding behaviour in the manner of C_{Add_1}, is:

$$\begin{aligned} &G \mathrel{\widehat{=}} aa = a \qquad P \mathrel{\widehat{=}} bb = b \\ &C \mathrel{\widehat{=}} aa = a - b \wedge resp = Fail \end{aligned} \tag{9}$$

Imagine for a moment that concrete model is changed - concrete input *bb* must become concrete state - in order to define a refinement as per (6):

$$\begin{aligned} &R \mathrel{\widehat{=}} aa + bb \leq Max \qquad K \mathrel{\widehat{=}} aa = a \wedge bb = b \\ &V \mathrel{\widehat{=}} resp = Ok \end{aligned} \tag{10}$$

4. Pre-postcondition specification in B

We present some new B syntax for manipulating stp-defined operations and their expression in a pre-postcondition style in B.

Recalling the axioms (1) of B GSL, it is obvious that each axiom has a corresponding pair of trm, prd statements for these relational components as defined by (3); e.g. $\mathsf{trm}(S \,[]\, T) \equiv \mathsf{trm}(S) \vee \mathsf{trm}(T)$. The following, corresponding, equivalences for total-correctness stp relations will be useful:

$$\begin{aligned} \mathsf{stp}(R \Longrightarrow S) \equiv \mathsf{stp}(R \mid S) & \qquad \equiv R \wedge \mathsf{stp}(S) \\ \mathsf{stp}(S \,[]\, T) \equiv \mathsf{stp}(S) \vee \mathsf{stp}(T) & \qquad \text{if } \mathsf{trm}(S) \equiv \mathsf{trm}(T) \equiv \mathsf{true} \\ \mathsf{stp}(@z \bullet S) \equiv \exists z \bullet \mathsf{stp}(S) & \qquad \text{if } \forall z \bullet \mathsf{trm}(S) \equiv \mathsf{true} \end{aligned} \tag{11}$$

We recollect from [1] the definition of the "nondeterministic pre-postcondition assignment" operation, so named because it is nondeterministic and refers to variable values in both pre- and postcondition. By $x : Q$ we mean "assign to x any value satisfying predicate Q":

$$\begin{aligned} x : Q \mathrel{\widehat{=}} @x' \bullet ([x_0, x := x, x']Q \Longrightarrow x := x') \\ \text{where } x' \setminus Q \end{aligned} \tag{12}$$

Note the syntactic difference between operation form $x : Q$, with x_0, x free in Q, and relational forms prd and stp, with x, x' free. It follows that

$$\begin{aligned} \mathsf{trm}(P \mid x : Q) &\equiv P \\ \mathsf{prd}(P \mid x : Q) &\equiv P \Rightarrow [x_0, x := x, x']Q \\ \mathsf{stp}(P \mid x : Q) &\equiv P \wedge [x_0, x := x, x']Q \end{aligned} \tag{13}$$

and it is easy to show the following GSL operation equalities[2] (we may read the following as a definition of parallel composition $||$):

$$\begin{aligned} P \mid x, y : Q \wedge R &= P \mid (x : Q \,||\, y : R) \qquad \text{where } x \setminus R, y \setminus Q \\ P \mid (x : Q \,[]\, x : R) &= P \mid x : Q \vee R \\ P \mid x : Q \Rightarrow R &= P \mid (x : \neg\, Q \,[]\, x : R) \end{aligned} \tag{14}$$

It is also easy to show the following results. Any precondition clause factors through as above[3]:

$$\begin{aligned} (x : x = x_0) &= \mathsf{skip} \\ &\quad \text{because } \mathsf{prd}(x : x = x_0) \equiv x' = x \\ (x : x = E(x_0)) &= x := E(x) \\ &\quad \text{because } \mathsf{prd}(x : x = E(x_0)) \equiv x' = E(x) \\ (x : F(x_0, z) \wedge E(x_0, x)) &= (F(x, z) \Longrightarrow x : E(x_0, x)) \\ &\quad \text{where } z \setminus x \end{aligned} \tag{15}$$

5. Lifting a retrenchment to an operation specification

We give the factorization of a B retrenchment into a retrenchment followed by a refinement. The result is analogous to that for simple transition systems [6]. Fig. 3 shows the construction relating the three models *Abs*, *Conc*, *Univ*. We assume a retrenchment (7), i.e. $Op_A \lesssim_{G,P,C} Op_C$, is given between *Abs* and *Conc*, which will be factored into the sequential composition of a retrenchment $Op_A \lesssim Op_U$ and a refinement $Op_U \sqsubseteq Op_C$, via constructed operation Op_U in *Univ*.

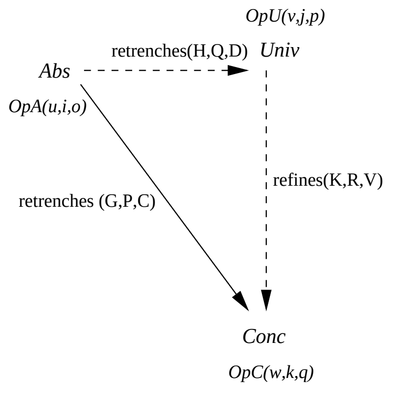

Figure 3. Factorization of a retrenchment

Abs and *Conc* have state, input, and output spaces U, I, O and W, K, Q respectively. The state, input, and output space of *Univ* is V, J, P. The *Univ* spaces are constructed from

2 The nonfreeness sidecondition $x \setminus Q, y \setminus P$ does not preclude x_0 from being free in Q, or y_0 from being free in P.

3 When we write $E(x_0)$ and $E(x)$ in the same context, we mean by the former that $x \setminus E(x_0)$ and by the latter $[x_0 := x]E(x_0)$

Abs and *Conc* as follows: $\mathrm{V} = \mathrm{U} \times \mathrm{W}$, $\mathrm{J} = \mathrm{I} \times \mathrm{K}$ and $\mathrm{P} = \mathrm{O} \times \mathrm{Q}$. The invariant on the state space V is given by:

$$Inv_U(v) = (v = (u, w) \wedge Inv_A(u) \wedge Inv_C(w)) \tag{16}$$

Next, *Univ* has a set of operations $Op_U \in \mathrm{OpsU}$ that correspond to operations Op_C in *Conc*. By the nature of retrenchment there are in general more Op_C in *Conc* than Op_A in *Abs*. To give the transitions of *Univ* we first note that OpsU comprises operations Op_U in either OpsA or in $\mathrm{OpsU} - \mathrm{OpsA}$. For an operation $Op_U \in (\mathrm{OpsU} - \mathrm{OpsA})$, the trm and stp predicates are defined:

$$\begin{aligned}
&\mathsf{trm}(Op_U)(v, j) = (v = (u, w) \wedge j = (i, k) \wedge \mathsf{trm}(Op_C)(w, k))\\
&\mathsf{stp}(Op_U)(v, j, v', p) = \mathsf{trm}(Op_U) \wedge \mathsf{stp}(Op_C)(w, k, w', q)\\
&\qquad \wedge (v = (u, w) \wedge j = (i, k) \wedge v' = (u', w') \wedge p = (o, q)\\
&\qquad \wedge u = u' \wedge o = \epsilon)
\end{aligned} \tag{17}$$

Thus the non-Op_A transitions of Op_U form a copy of the Op_C stp relation with identity behaviour on the abstract component, asserted to terminate on states/inputs that project to $\mathsf{trm}(Op_C)$. For an $Op_U \in \mathrm{OpsA}$ we have:

$$\begin{aligned}
&\mathsf{trm}(Op_U)(v, j) =\\
&\qquad (v = (u, w) \wedge j = (i, k) \wedge \mathsf{trm}(Op_A)(u, i) \wedge \mathsf{trm}(Op_C)(w, k))\\
&\mathsf{stp}(Op_U)(v, j, v', p) = \mathsf{trm}(Op_U) \wedge Inv_A(u') \wedge Inv_C(w')\\
&\qquad \wedge (v = (u, w) \wedge j = (i, k) \wedge v' = (u', w') \wedge p = (o, q))\\
&\qquad \wedge (G(u, w) \wedge P(i, k, u, w) \Rightarrow\\
&\qquad\qquad \mathsf{stp}(Op_A)(u, i, u', o) \wedge\\
&\qquad\qquad (G(u', w') \vee C(u', w', o, q;\ i, k, u, w)))
\end{aligned} \tag{18}$$

Note that the implicational form admits many junk steps, namely $\neg\ (G \wedge P) \wedge Inv_A \wedge Inv_C \wedge o \in O \wedge q \in Q$. The last element of the universal machine is its initialization:

$$\begin{aligned}
&Init_U(v) \mathrel{\widehat{=}}\\
&\qquad v : v = (u, w) \wedge \mathsf{stp}(Init_A)(u) \wedge \mathsf{stp}(Init_C)(w) \wedge G(u, w)
\end{aligned} \tag{19}$$

The consistency obligations for *Univ* are satisfied for both $Op_U \in \mathrm{OpsU} - \mathrm{OpsA}$ and $Op_U \in \mathrm{OpsA}$:

$$\begin{aligned}
&[Init_U] Inv_U\\
&Inv_U \wedge \mathsf{trm}(Op_U) \Rightarrow [Op_U] Inv_U
\end{aligned} \tag{20}$$

We now give the retrenchment $Op_A \lesssim_{H,Q,D} Op_U$ and the refinement $Op_U \sqsubseteq Op_C$. The data for the retrenchment consists of the retrieve relation, and the within and concedes relations for this operation pair:

$$\begin{aligned}
H(u, v) &\mathrel{\widehat{=}} (v = (u, w) \wedge G(u, w)) \qquad (21)\\
Q(i, j, u, v) &\mathrel{\widehat{=}} (j = (i, k) \wedge v = (u, w) \wedge P(i, k, u, w))\\
D(u', v', o, p;\ i, j, u, v) &\mathrel{\widehat{=}} (v' = (u', w') \wedge p = (o, q) \wedge j = (i, k)\\
&\qquad \wedge v = (u, w) \wedge C(u', w', o, q;\ i, k, u, w))
\end{aligned}$$

The retrenchment POs are satisfied for $Op_U \in \mathrm{OpsA}$:

$$\begin{aligned}
&[Init_U] \neg [Init_A] \neg H\\
&Inv_A \wedge Inv_U \wedge H(u, v) \wedge trm(Op_U) \wedge Q(i, j, u, v)\\
&\ \Rightarrow trm(Op_A) \qquad (22)\\
&\qquad \wedge [Op_U] \neg [Op_A] \neg (H(u', v') \vee D(u', v', o, p;\ i, j, u, v))
\end{aligned}$$

For the refinement $Op_U \sqsubseteq_{K,R,V} Op_C$, the data consists of the retrieve, within, and output relations. These are simply the projections:

$$\begin{aligned}
K(v, w) &\mathrel{\widehat{=}} (v = (u, w))\\
R(j, k) &\mathrel{\widehat{=}} (j = (i, k))\\
V(p, q) &\mathrel{\widehat{=}} (p = (o, q))
\end{aligned} \tag{23}$$

The refinement POs are satisfied for both Op_U versions:

$$\begin{aligned}
&[Init_C] \neg [Init_U] \neg K\\
&Inv_U \wedge Inv_C \wedge K(v, w) \wedge trm(Op_U) \wedge R(j, k)\\
&\ \Rightarrow trm(Op_C) \wedge [Op_C] \neg [Op_U] \neg (K(v', w') \wedge V(p, q))
\end{aligned} \tag{24}$$

It remains to define the composition of the retrenchment and IO-filtered refinement just constructed. This requires care since not only are retrenchments and refinements different concepts, but also the collections of variables of the intermediate system occurring in abutting relations are not the same. We define the composition to be a retrenchment for which the component relations are given as follows. The retrieve relation is the (usual) composition of the component retrieve relations:

$$(H; K)(u, w) \mathrel{\widehat{=}} \exists\, v \bullet (H(u, v) \wedge K(v, w)) \tag{25}$$

The within relation is the composition of the component within and retrieve relations in the following sense:

$$\begin{aligned}
&(Q; (R \wedge K))(i, k, u, w)\\
&\mathrel{\widehat{=}} (\exists\, v, j \bullet Q(i, j, u, v) \wedge R(j, k) \wedge K(v, w))
\end{aligned} \tag{26}$$

The concedes relation is a combination of the component concedes, retrieve, output, and within relations in the following manner:

$$\begin{aligned}
&(D; (K' \wedge V \wedge R \wedge K))(u', w', o, q;\ i, k, u, w)\\
&\mathrel{\widehat{=}} (\exists\, v, j, v', p \bullet D(u', v', o, p;\ i, j, u, v) \wedge K(v', w') \wedge V(p, q)\\
&\qquad \wedge R(j, k) \wedge K(v, w))
\end{aligned} \tag{27}$$

The reader may check that this notion of composition recovers the defining relations of the original retrenchment (7).

In this reinterpretation the work of [6] in B, we have reconstructed the factorization of the original retrenchment through a certain retrenchment and a refinement. In fact the original construction of *Univ* [6] is universal, or minimal, in a category-theoretic sense, within a class of models with similar properties: any other candidate construction $Univ' \neq Univ$ for the factorization, itself factors through *Univ* in a particular way. The reinterpretation of this universal property in B, not itself required in this paper, is work in progress.

6. The factorized retrenchment: example

The construction Op_U of section 5 represents the "lifting" of the given retrenchment, via the factorization, to an operation specification at the level of abstraction of the original ideal operation Op_A. Using the equivalences of section 4, we express the construction in B GSL. We then discuss its utility in two example retrenchment specifications.

Since only the case $Op_U \in \mathrm{OpsA}$ is of interest, we express its step relation stp_{Op_U} (18) over input-state-output space $\mathrm{I} \times \mathrm{K} \times \mathrm{U} \times \mathrm{W} \times \mathrm{O} \times \mathrm{Q}$ in B GSL:

$$\begin{aligned} & u, w, o, q : Inv_A(u) \wedge Inv_C(w) \wedge \mathsf{trm}_{Op_A}(u_0, i) \wedge \mathsf{trm}_{Op_C}(w_0, k) \\ & \quad \wedge (G(u_0, w_0) \wedge P(i, k, u_0, w_0) \\ & \qquad \Rightarrow \mathsf{stp}_{Op_A}(u_0, i, u, o) \\ & \qquad\quad \wedge (G(u, w) \vee C(u, w, o, q;\ i, k, u_0, w_0))) \end{aligned} \tag{28}$$

We expand the lifted $\mathsf{trm}_{Op_U}, \mathsf{stp}_{Op_U}$ definition (18) and apply (14,15) to define $Op_U(i, k, u, w, o, q)$:

$$\begin{aligned} & \mathsf{trm}_{Op_A}(u, i) \wedge \mathsf{trm}_{Op_C}(w, k) \\ & \mid \big(\neg\, G(u, w) \vee \neg\, P(i, k, u, w) \Longrightarrow u, w, o, q : Inv_A(u) \wedge Inv_C(w) \\ & \quad [\,]\ u, w, o, q : Inv_A(u) \wedge Inv_C(w) \wedge \mathsf{stp}_{Op_A}(u_0, i, u, o) \wedge G(u, w) \\ & \quad [\,]\ u, w, o, q : Inv_A(u) \wedge Inv_C(w) \\ & \qquad \wedge \mathsf{stp}_{Op_A}(u_0, i, u, o) \wedge C(u, w, o, q;\ i, k, u_0, w_0)\big) \end{aligned} \tag{29}$$

Op_U is the most nondeterministic operation at the level of abstraction of Op_A, that both satisfies the postcondition $G \vee C$ of the given retrenchment and refines to Op_C. The WITHIN, RETRIEVES and CONCEDES clauses that describe the relationship between Op_A and Op_C may include requirements information; these clauses now become explicit in the specification (29) of Op_U in the pre-postcondition assignment style. Returning to the example of Fig. 2, we note that

$$\begin{aligned} & Inv_A(a, b) \mathrel{\widehat{=}} a, b \in \text{D-NAT} \qquad Inv_C(aa) \mathrel{\widehat{=}} aa \in \text{M-NAT} \\ & \mathsf{stp}_{DAdd}(a, a', b, b') \equiv a' = a + b \wedge b' = b \end{aligned} \tag{30}$$

Thus we can express the universal Op_U for Fig. 2, with concession C_{Add_1} (8), in the expanded pre-post form of (29). Recall that in the pre-post style, before and after variables are denoted zero- and unsubscripted, respectively:

$$\begin{aligned} & Op_U(a, b, aa, bb, resp) \mathrel{\widehat{=}} bb \in \text{M-NAT} \mid \\ & \quad a \neq aa \vee b \neq bb \Longrightarrow \\ & \qquad a, b, aa, resp : a, b \in \text{D-NAT} \wedge aa \in \text{M-NAT} \\ & \quad [\,]\ a, b, aa, resp : a, b \in \text{D-NAT} \wedge aa \in \text{M-NAT} \\ & \qquad\qquad \wedge a = a_0 + b_0 \wedge b = b_0 \wedge aa = a \\ & \quad [\,]\ a, b, aa, resp : a, b \in \text{D-NAT} \wedge aa \in \text{M-NAT} \\ & \qquad\qquad \wedge a = a_0 + b_0 \wedge b = b_0 \\ & \qquad\qquad \wedge aa_0 + bb \leq Max \wedge aa = a \wedge resp = Ok \\ & \quad [\,]\ a, b, aa, resp : a, b \in \text{D-NAT} \wedge aa \in \text{M-NAT} \\ & \qquad\qquad \wedge a = a_0 + b_0 \wedge b = b_0 \\ & \qquad\qquad \wedge aa_0 + bb > Max \wedge aa = a - b \\ & \qquad\qquad \wedge resp = Fail \end{aligned} \tag{31}$$

We number the cases of (31) from 1, (31.1) being the technical "junk" transition case. In the cases (31.1, 31.2) *resp* does not appear free in the predicate and thus may take either defined value *Ok*, *Fail*.

The cases can be simplified using the results of section 4, and the assumption that in the before-state Inv_U, that is, $Inv_A(u_0) \wedge Inv_C(w_0)$ always holds - these clauses are assumptions of all POs defining any context in which Op_U is interpreted. The precondition clause in *bb* distributes implicitly to each case.

For case (31.2), the precondition is irrelevant, by equality of predicates in that context we replace $aa = a$ with $aa = a_0 + b_0$, and $Inv_A(a, b)$ follows from the equalities on a, b. As before *resp* is undetermined in the predicate. The concrete assignment cannot be simplified; $Inv_C(aa)$ prevents overflowing abstract additions appearing in this refining case:

$$\begin{aligned} (31.2) = {} & (a, b := a + b, b) \\ & \mid\mid aa : aa = a_0 + b_0 \wedge aa \in \text{M-NAT} \\ & \mid\mid resp : resp \in \{Ok, Fail\} \end{aligned} \tag{32}$$

In the above *resp* is left undetermined because the construction (28) does not define a relationship between abstract and concrete output in the refining G postcondition region; such a relationship is only defined in the concession C region. This "output-incompleteness" of simple retrenchment (7) has been dealt with by the recent proposal of *output retrenchment* [8], which conjoins an OUTPUT clause to the retrieve relation, similarly to refinement (6). Notice that case (31.3) below determines *resp* by virtue of extra information in the concession.

(31.3) is reduced similarly, and here a guard is extracted from the concrete before-state-input clause. $Inv_A(a, b)$ follows as before, and $Inv_C(aa)$ follows from the guard, so that all assignments become simple:

$$\begin{aligned} (31.3) = {} & aa + bb \leq Max \Longrightarrow \\ & \quad a, b, aa, resp := a + b, b, a + b, Ok \end{aligned} \tag{33}$$

Similarly, for (31.4) we extract a guard. We replace $aa = a - b$ with $aa = a_0$ by equality of predicates, and use the retrieve relation $a_0 = aa_0$ to infer $Inv_C(aa)$:

$$\begin{aligned} (31.4) = {} & aa + bb > Max \Longrightarrow \\ & \quad a, b, aa, resp := a + b, b, a, Fail \end{aligned} \tag{34}$$

Putting the pieces together gives

$$\begin{aligned} & Op_U(C_{Add_1})(a, b, aa, bb, resp) \mathrel{\widehat{=}} bb \in \text{M-NAT} \mid \\ & \quad a \neq aa \vee b \neq bb \Longrightarrow \\ & \qquad a, b, aa, resp : a, b \in \text{D-NAT} \\ & \qquad\qquad \wedge aa \in \text{M-NAT} \wedge resp \in \{Ok, Fail\} \\ & \quad [\,]\ (a, b := a + b, b) \mid\mid aa : aa = a_0 + b_0 \wedge aa \in \text{M-NAT} \\ & \qquad\qquad \mid\mid resp : resp \in \{Ok, Fail\} \\ & \quad [\,]\ aa + bb \leq Max \Longrightarrow a, b, aa, resp := a + b, b, a + b, Ok \\ & \quad [\,]\ aa + bb > Max \Longrightarrow a, b, aa, resp := a + b, b, a, Fail \end{aligned} \tag{35}$$

Since we can see informally that $aa + bb \leq Max$ defines precisely the region where $aa : aa = a_0 + b_0$ can establish that $aa \in$ M-NAT, case (35.2) contains case (35.3) as a set of transitions. In particular (35.2) is more nondeterministic w.r.t. the concrete output *resp*. Thus we might choose to refine Op_U (35) without losing any relevant information, by deleting case (35.2). It is a curiosity that in this case, the first, refining C_{Add_1} disjunct makes the retrieve relation G redundant.

Although this is a simple example, the utility of the lifting construction in providing an alternative perspective on the original retrenchment, is evident. Whereas in the original retrenchment, the relationship between operations Op_A and Op_C is located in a third conceptual “place”, i.e. in the retrenchment PO (7), this relationship is now explicit in the Op_U specification in the single *Univ* frame. In each non-junk case we see all variable transitions, abstract and concrete, related by the conditions that pertain in the subregion in question (defined by G or by C-disjunct).

6.1. Discussion

A stronger concession than C_{Add_1} (8) seems desirable insofar as, intuitively, it seems redundant for the concession region to overlap with the retrieving region. An obvious choice is the non-refining disjunct of C_{Add_1}:

$$C_{Add_2} \mathrel{\widehat{=}} aa_0 + bb > Max \wedge aa = a - b \wedge resp = Fail \qquad (36)$$

This version of Op_U (which we might call $Op_U(C_{Add_2})$) has the form (35) minus transitions (35.3): notwithstanding the increased *resp* nondeterminism discussed above, this gives a more succinct universal specification.

In this example we note that $C_{Add_2} \Rightarrow C_{Add_1}$, and that $Op_U(C_{Add_2})$ may be constructed by deletion of transitions from $Op_U(C_{Add_1})$. In fact, deleting transitions amounts to refinement over the same variable type: consider (6) with identity within, retrieve and output relations. Any concrete transition reestablishes the identity postcondition, witnessed by the identity, abstract transition. Hence $Op_U(C_{Add_1}) \sqsubseteq Op_U(C_{Add_2})$ here. Moreover, concession strengthening in $Op_A \lesssim Op_C$ amounts to refinement of Op_U: imagine two candidate concessions C_1, C_2 for the given retrenchment such that $C_2 \Rightarrow C_1$. Then $C_1 \Leftrightarrow (C_2 \vee (\neg\, C_2 \wedge C_1))$, and deletion of the transition set that establishes $\neg\, C_2 \wedge C_1$ in Op_U constitutes a refinement.

This is another useful feature of the lifting construction. There is considerable concession design freedom in specifying a retrenchment of operations. It is obvious that, for the given retrenchment, a stronger concession which is closer to false, moves closer to specifying a refinement, which is more incisive than a retrenchment. Alternative retrenchment designs can be compared as Op_U specifications. If two such designs differ only by relative concession strength, then these designs lift to two operations related by refinement.

The point is emphasised by considering

$$C_{Add_0} \mathrel{\widehat{=}} aa_0 + bb > Max \Rightarrow aa = a - b \wedge resp = Fail \qquad (37)$$

Since C_{Add_1} is equivalent to the implicative guarded form $(gd \Rightarrow txn_{Ok}) \wedge (\neg\, gd \Rightarrow txn_{Fail})$ provided $txn_{Ok} \wedge txn_{Fail} \Leftrightarrow \mathsf{false}$, it follows that $C_{Add_1} \Rightarrow C_{Add_0}$ and thus that $Op_U(C_{Add_0}) \sqsubseteq Op_U(C_{Add_1}) \sqsubseteq Op_U(C_{Add_2})$. That $Op_U(C_{Add_0})$ contains $Op_U(C_{Add_1})$ as a set of transitions is clear from calculating the former explicitly from (29) as before:

$$\begin{array}{l}
Op_U(C_{Add_0})(a, b, aa, bb, resp) \mathrel{\widehat{=}} bb \in \text{M-NAT} \mid \\
\quad a \neq aa \vee b \neq bb \Longrightarrow \\
\qquad a, b, aa, resp : a, b \in \text{D-NAT} \\
\qquad\qquad\qquad \wedge\, aa \in \text{M-NAT} \wedge resp \in \{Ok, Fail\} \\
\quad [\,]\ (a, b := a + b, b) \mid\mid aa : aa = a_0 + b_0 \wedge aa \in \text{M-NAT} \\
\qquad\qquad\qquad \mid\mid resp : resp \in \{Ok, Fail\} \\
\quad [\,]\ aa + bb \leq Max \Longrightarrow (a, b := a + b, b) \\
\qquad\qquad\qquad \mid\mid aa, resp : aa \in \text{M-NAT} \\
\qquad\qquad\qquad\quad \wedge\, resp \in \{Ok, Fail\} \\
\quad [\,]\ a, b, aa, resp := a + b, b, a, Fail
\end{array} \qquad (38)$$

The junk and refining cases (38.1, 38.2) are the same as for (35). (38.3) is the case of the negation of the C_{Add_0} guard $aa + bb > Max$, which leaves $aa, resp$ undetermined. This case therefore contains transitions (35.3). Finally, the concession assignment clauses now being unguarded, (38.4) contains (35.4).

In the limit, a false concession in the given retrenchment constitutes the refinement of Op_U (29) by deletion of all conceding case (29.3) transitions.

Finally, it is illustrative to compare C_{Add_1} with

$$\begin{array}{l}
\mathsf{stp}_{MAdd}(aa, bb, aa', resp) \mathrel{\widehat{=}} \\
bb \in \text{M-NAT} \wedge \\
\quad ((aa + bb \leq Max \wedge aa' = aa + bb \wedge resp = Ok) \\
\quad\ \vee (aa + bb > Max \wedge aa' = aa \wedge resp = Fail))
\end{array} \qquad (39)$$

These two predicates differ only in the precondition term which types bb in stp_{MAdd}, and insofar as the second, abstraction clause in each disjunct of C_{Add_1} is replaced by a state transition clause in stp_{MAdd}. Observe that in the context of Fig. 2 $C_{Add_1} \Rightarrow \mathsf{stp}_{MAdd}$: in this simple deterministic example, a full concession account of the transformation of the abstraction relation, over the retrenchment step, implies the concrete step relation.

6.2. A second example

We consider, in outline only, an *approximation* retrenchment example with richer and more general syntactic structure from [20]. The example is a simple resource allocator. Abstract model *Abs* states a simple functional requirement about resource allocation, based on yes/no availability. *Abs* makes a simple binary guarded choice between allocating some available resource x in the environment, which meets an input requirement parameter i, or skipping if no resource is available. Any selected x is added to set of allocated resources, the state variable u.

Concrete model *Conc* models the constraints of providing resources that meet requirements, in a distributed environment, in a three-way choice: allocation of an available and trusted resource to state variable w, allocation of the best available but only partially trusted resource to w, or skipping. This gives quite a general pattern of abstract/concrete model specification and retrenchment. Two operations, each constituting a choice over a set of guarded commands, have decoupled guard sets, where the degree of decoupling depends on how much *Conc* differs from *Abs*. Each operation also incorporates nondeterministic choice over a set of candidate resources.

The retrenchment describes the approximate representation of abstract allocated set u by concrete w, via retrieve and concession relations which are quite different in shape and design intention from the previous example. Here we have "RETRIEVES $G_{\delta,n}$" and "CONCEDES $G_{\delta+1,n} \vee G_{\delta,n+1}$"; δ counts how much bigger u is than w, assuming that *Conc* will fail to allocate more often than *Abs* will. n counts the number of partially trusted elements allocated in w.

Schematically, Op_A has a precondition $AP(u,i)$, and a guard $Q(x,i,u)$ which determines the availability of an allocatable resource x meeting requirement i:

$$\begin{aligned} Op_A \mathrel{\widehat{=}} AP(u,i) \mid \quad & @x \bullet (Q(x,i,u) \Longrightarrow u := u \cup \{x\}) \\ & [] \neg\ \exists x \bullet Q(x,i,u) \Longrightarrow \mathsf{skip}) \end{aligned} \tag{40}$$

Since the GSL definition of Op_U (29) makes no reference to the concrete syntax of Op_C, we need not define the latter explicitly. By instantiating (29) and applying (11,15) as before we have:

$$\begin{aligned} & Op_U \mathrel{\widehat{=}} AP(u,i) \wedge \mathsf{trm}(Op_C)(w,k) \mid \\ & \quad \big(\ \ \neg\, G(u,w) \vee \neg\, P(i,k,u,w) \Longrightarrow u,w,q : u \in \mathrm{U} \wedge Inv_C(w) \\ & \quad [] \ @x \bullet (Q(x,i,u) \Longrightarrow u,w,q : u = u_0 \cup \{x\} \\ & \qquad\qquad \wedge (G_{\delta,n} \vee G_{\delta+1,n} \vee G_{\delta,n+1}) \wedge u \in \mathrm{U} \wedge Inv_C(w)) \\ & \quad [] \neg\ \exists x \bullet Q(x,i,u) \Longrightarrow u,w,q : u = u_0 \qquad (41) \\ & \qquad\qquad \wedge (G_{\delta,n} \vee G_{\delta+1,n} \vee G_{\delta,n+1}) \wedge u \in \mathrm{U} \wedge Inv_C(w)\big) \end{aligned}$$

As before, the requirements in the retrenchment parameters - here, the change in approximate representation - are lifted to the Op_U specification. The simplifications in the first example were made possible by the concession style: detailed partitioning of the before-state-input frame and specification of transitions in each case. This is not possible with the concession style here.

However, it is illustrative to consider, as before, the way alternative retrenchment designs lift and compare in *Univ*. In the first example, we saw how concession-implication lifted to refinement. It is obvious in (29.1) that relative P-weakening (i.e. $\neg\, P$-strengthening) lifts to refinement in the same way by deletion of junk transitions in Op_U.

Conversely, P-strengthening consitutes abstraction in *Univ* by addition of junk transitions in the $\neg\, P$ case (29.1). The full treatment [20] of the resource allocator proposes a method of *decomposition* of a retrenchment. It is shown how, by considering a set of strengthened P clauses (which partition the before-state-input frame in terms of enabled abstract and concrete guards) it is possible to produce a corresponding set of stronger-concession, *finer-grained* retrenchments exploiting that partition information. In this example this produces four clauses that characterize the four possible situations that can arise in $Abs, Conc$. These clauses are named P_{11} (abstract allocate, concrete allocate), P_{12} (abstract allocate, concrete partial-trusted allocate), P_{13} (abstract allocate, concrete skip), and P_{23} (abstract skip, concrete skip). These four retrenchments have in turn postconditions $G_{\delta,n}$ (refinement), $G_{\delta,n} \vee G_{\delta,n+1}$, $G_{\delta,n} \vee G_{\delta+1,n}$, and again $G_{\delta,n}$.

In the P_{13} case we see how the retrenchment definition (7), by mandating a $G \vee C$ postcondition shape, prevents a simpler (concession-only) postcondition statement $G_{\delta+1,n}$ being made. The $G \vee C$ statement is too weak where we can prove it is precisely the concession that will hold in the postcondition. In this example the concession is strictly weaker than the retrieve relation, so here this excessive postcondition-weakness is not an issue. However, precondition-strengthening (41) as per [20] to P_{13} gives, for the (41.2) case:

$$\begin{aligned} & @x \bullet (Q(x,i,u) \Longrightarrow \\ & \qquad u,w,q : u = u_0 \cup \{x\} \wedge (G_{\delta,n} \vee G_{\delta+1,n}) \\ & \qquad\qquad \wedge u \in \mathrm{U} \wedge Inv_C(w)) \end{aligned} \tag{42}$$

Thus it is intruiging that, once lifted to *Univ*, the P_{13} case can be refined by deleting the $G_{\delta,n}$ transitions to establish precisely the postcondition statement $G_{\delta+1,n}$. This indicates that the lifting construction provides in this sense, through refinement, a more finely-grained, incisive specification method than simple retrenchment. This suggests that there is an opportunity, by "unlifting" Op_U back to the level of abstraction of the original retrenchment, of making correspondingly finer-grained statements about the relationship between *Abs* and *Conc* than are possible in simple retrenchment.

7. Conclusion and further work

We have seen different ways in which requirements can be represented semantically in the retrenchment relations between models, as well as being syntactically represented within those models. The universal factorization [6] of a given retrenchment $Abs \lesssim Conc$ into a universal retrenchment $Abs \lesssim Univ$ followed by a refinement $Univ \sqsubseteq Conc$

was reformulated in B. It was then shown how this construction "lifted" the semantic retrenchment information in the within, retrieves, and concedes relations to syntactic operation specification in the universal model *Univ*. Some new pre-postcondition syntax for B facilitates this lifting process by enabling the lifted operation to be expressed using the full B GSL language in the conventional way.

The examples showed the cognitive utility of lifting retrenchment information in this way. In particular the lifted operation, as a highly nondeterministic set of transitions, is very manipulable by refinement by the deletion of transitions. This kind of refinement in *Univ* also provides a simple way to compare alternative retrenchment designs (related by relative *C*-strength, or *P*-weakness) at the specification level. The second example showed how other forms of analysis of the retrenchment, such as decomposition, can be lifted and analysed by combining abstraction and refinement in the *Univ* transition space. This gives a viewpoint, at the level of the *Univ* specification, that maps back to a more finely-grained relationship between *Abs* and *Conc* than the original retrenchment. These forms of analysis look promising and will be investigated further.

Animation and model-checking technology (e.g. *ProB* [17] for B) has been applied to the verification of refinements, and is in principle similarly applicable to retrenchments. The lifted model offers a more understandable locus (in the checking of traces, rather than pairs of traces) for model-checking.

References

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

[2] J.-R. Abrial. http://www.atelierb.societe.com/index_uk.html, 1998. Atelier-B.

[3] K. Araki, S. Gnesi, and D. Mandrioli, editors. *International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, Pisa, Italy, September 2003. Springer.

[4] R. J. R. Back and J. von Wright. Refinement calculus part I: Sequential nondeterministic programs. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop, Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 42–66. Springer, 1989.

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

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

[7] R. Banach. Aspects of general retrenchment: Symmetric propositional theory. work in progress, 2004.

[8] 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.

[9] 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.

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

[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] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*, chapter 10, pages 227–252. Prentice-Hall, 1995.

[15] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. FACIT. Springer, 2001.

[16] P. Henderson. Reasoning about asynchronous behaviour in distributed systems. In *Proc. Eighth IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS02)*, pages 17–24, Greenbelt, Maryland, December 2002. IEEE Computer Society Press.

[17] M. Leuschel and M. Butler. ProB: a model checker for B. In Araki et al. [3], pages 855–874.

[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. Bowen, M. 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. Poppleton and R. Banach. Structuring retrenchments in B by decomposition. In Araki et al. [3], pages 814–833.

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