Software Evolution with Refinement and Retrenchment

Michael Poppleton[1] and Lindsay Groves[2]

[1] Department of Electronics and Computer Science,
University of Southampton, Highfield,
Southampton SO17 1BJ, UK,
mrp@ecs.soton.ac.uk,
WWW home page: http://www.ecs.soton.ac.uk/~mrp/

[2] School of Mathematical and Computing Sciences,
Victoria University of Wellington, New Zealand,
lindsay@mcs.vuw.ac.nz,
WWW home page: http://www.mcs.vuw.ac.nz/~lindsay

May 28, 2003

1 Introduction

The problem of how to construct a program correct with respect to its specification, a longstanding and central concern of the Formal Methods community, is now well understood. However, the orthogonal problem of software evolution has not until recently received serious examination by Formal Methods researchers. Most software, once constructed (and even during construction), is subject to the changing demands of its surrounding environments: physical, regulatory, market, and customer. Customer requirements change, be it in adding requirements, deleting or modifying existing requirements, or merging existing requirements/code with (not necessarily consistent) requirements/code from elsewhere. Refactoring changes may be required for reasons of optimisation against equipment or market changes, or may be driven by software reuse strategies.

These evolutionary pressures also apply to critical systems, where a documented formal development history suggests the possibility of formally-based support for evolution. Given a software component, properly developed from its specification through some sequence of refinements to code, the aspiration is to preserve and deploy that investment through controlled evolution of the component.

Given a record of the steps by which a component is constructed from its specification and a new specification, similar to the old one, we can compare the two specifications and use the differences between them as a basis for revising the derivation of the component and attempt to discharge the resulting proof obligations. This is essentially the way that program refinements are modified by hand, and could be supported by a refinement tool as outlined in [7]. Alternatively, we might describe the new specification by combining the original specification and a description of the new behaviour required using combinators similar to schema conjunction and disjunction in Z. This approach has been explored in the context of the refinement calculus [12, 13] by Groves [8, 9] and in a relational setting by Ben Ayed *et al* [5].

Independently of this revision-based and constructive work, another potential formal approach to the evolution of specifications has recently emerged. *Retrenchment* [2, 3] is a generalisation of classical refinement which was proposed to extend the reach of formal verification to applications too demanding, e.g. [4, 14], to be described, computationally, in terms of refinement. Recent work [16] has proposed the integration of these constructive and retrenchment approaches.

Drawing on the authors' previous work, this paper will report on initial work investigating the construction of evolutions using these constructive and retrenchment approaches. In special cases the evolution transformation may be a refinement. More widely applicable will be certain specification combinators and constructors. Finally, as the most general relation between specifications, retrenchment will be applicable to any evolution; the work will be in establishing how to use it to preserve structure in a useful way.

Section 2 motivates and provides background for these approaches. Retrenchment is defined in terms of classical refinement, and a theorem for calculating the transitive composition of two retrenchments is given. Section 3 examines the potential for modelling software evolution in refinement and retrenchment frameworks. The main analytical reference point is the "evolution square" of Fig. 1. The formal view of the evolution problem can be described in terms of finding the unknown elements of the square. A number of approaches are suggested, followed by informal definitions in section 3.2 of concepts of commutativity of the square. For a small illustrative example of a commutativity analysis, section 3.3 calculates the two square traversals as composed retrenchments. Section 4 then develops the constructive approach, suggesting classifications of vertical refinements and horizontal retrenchments. Certain simple commutativity results are given. In conclusion connections between the two approaches are identified.

2 Background

This section will provide a brief outline of the two approaches to describing modifications to programs being combined here, namely the constructive description of modifications in the refinement calculus, and retrenchment.

We consider specifications to be modules consisting of local state variables and operations that can modify those variables, and concentrate primarily on modifications that alter the specification of an individual operation. A more complete analysis will have to consider modifications at the module level, which will typically involve adding and deleting operations, moving operations between modules, and changing the number and/or types of parameters, as well as modifications to the functionality of individual operations.

2.1 Constructive description of modifications

The basic idea of this approach is to describe modifications to specifications (expressed in a wide spectrum language which includes both executable programs and various non-executable specification constructs, including specification statements) by composing them with specifications describing the required modifications. Modifications are described using program combinators similar to Z's schema operators. For example, consider a specification S which describes a program that finds an integer k in the range 1 to n (for some positive constant n) satisfying some condition p, assuming that at least one such value exists. We might extend S to a program that returns an integer outside the range $1 \mathinner{\ldotp\ldotp} n$ if there is no value in $1 \mathinner{\ldotp\ldotp} n$ satisfying p, by writing $S \vee T$, where T sets k to a value outside the range $1 \mathinner{\ldotp\ldotp} n$ provided there is no value in $1 \mathinner{\ldotp\ldotp} n$ satisfying p. Alternatively, we could extend S to a program that also finds the number of values in $1 \mathinner{\ldotp\ldotp} n$ satisfying p, by writing $S \wedge U$, where U sets k to the number of values in $1 \mathinner{\ldotp\ldotp} n$ satisfying p.

Having described the required modification in this form, we can now use distributive properties of these operators to push the descriptions of the modification into the structure of the existing program (or, rather, through its derivation) to the place(s) where changes are required. In some case we need to modify the structure of the existing program in order to allow the modification to be made in a satisfactory way.

2.2 Retrenchment

We briefly recall the motivation for and definition of retrenchment. The intuition that drove such a generalisation of refinement came from its limitations in certain applications. The strong central assumption of refinement, that the refining model must exactly simulate the refined one, can unacceptably limit the degree of abstraction possible. Real numbers cannot be refined to floating or fixed point numbers: a fundamental problem for the wide range of applications described in terms of continuous mathematics or classical physics. The software engineering principle of separation of concerns is breached when refinement requires an ideal functional model to be cluttered with details of exception processing in the code. Requirements incompleteness and inconsistency give further motivations for such a generalisation of the classical method.

We give a general formulation of the "forward" refinement rule, in a simple transition systems framework which glosses over the partial/total correctness distinction by stating that only defined transitions are of interest, and these all terminate. We say that abstract operation OpA, with step relation $\mathsf{stp}_A(u, i, u', o)$

over state variable u, input i and output o, is refined by concrete operation OpC with step relation $\mathsf{stp}_C(v, j, v', p)$ subject to abstraction relation $G(u, v)$ if

$$G(u, v) \wedge \mathsf{stp}_C(v, j, v', p) \Rightarrow \exists\, u', o \bullet (\mathsf{stp}_A(u, i, u', o) \wedge G(u', v')) \tag{1}$$

Retrenchment was devised for the "vertical" (in the sense of Fig. 1) problem of correct software construction from ideal specifications that are in some sense non-refinable. Retrenchment deals with the situation where the simulation property (reestablishment of G in (1)) cannot be achieved. Within a defined subdomain of the joint before-state/input frame (called the *within* clause $W_{AC}(i, j, u, v)$), we say that OpA is *retrenched* by OpC with respect to the abstraction relation $G(u, v)$ and the *concession* clause $C_{AC}(u, v, u', v', o, p)$ on the joint before- and after-state and output frame if

$$\begin{aligned} W_{AC}(i, j, u, v) \wedge G(u, v) \wedge \mathsf{stp}_C(v, j, v', p) \Rightarrow& \\ \exists\, u', o \bullet (\mathsf{stp}_A(u, i, u', o) \wedge (G(u', v') \vee C_{AC}(u, v, u', v', o, p)))& \end{aligned} \tag{2}$$

We also use a shorthand form by saying $OpA \precsim OpC$ w.r.t. (W_{AC}, G, C_{AC}).

The concession represents a formal statement of what can be achieved if simulation fails. The generality of this POB (proof obligation) is given by the simple disjunctive form of the postcondition as well as the ability of the concession to make a statement about before- and after-state as well as outputs.

Retrenchment has been formalised and used in a number of applications [Op. Cit.]. It generalises refinement (set $W \mathrel{\widehat=} \mathsf{true}$, $C \mathrel{\widehat=} \mathsf{false}$) and allows, in the extreme, *any* two operations to be related (set $C \mathrel{\widehat=} \mathsf{true}$). Intuitively, therefore, the retrenchment notion should be applicable "horizontally" (in the sense of Fig. 1) to the evolution of specifications.

2.3 Composing retrenchments

Retrenchment is transitive [3]; we give the result which will be used later on in composing retrenchments.

Theorem Assume that $Op_1(v, j, v', p)$ retrenches $Op_0(u, i, u', o)$ w.r.t. (W_1, G_1, C_1), and that further operation $Op_2(w, k, w', q)$ retrenches Op_1 w.r.t. (W_2, G_2, C_2). Then Op_2 retrenches Op_0 w.r.t. (W_{12}, G_{12}, C_{12}) where

$$\begin{aligned} W_{12} &\mathrel{\widehat=} \exists\, v, j \bullet (G_1(u, v) \wedge G_2(v, w) \wedge W_1(i, j, u, v) \wedge W_2(j, k, v, w)) \\ G_{12} &\mathrel{\widehat=} \exists\, v \bullet (G_1(u, v) \wedge G_2(v, w)) \\ C_{12} &\mathrel{\widehat=} \exists\, v, p \bullet (G_1(u, v) \wedge C_2(v, w, p, q)) \\ &\quad \vee \exists\, v, p \bullet (C_1(u, v, o, p) \wedge G_2(v, w)) \\ &\quad \vee \exists\, v, p \bullet (C_1(u, v, o, p) \wedge C_2(v, w, p, q)) \end{aligned} \tag{3}$$

The result is intuitively satisfying. The composite abstraction relation G_{12} combines component abstraction relations. The WITHIN clause W_{12} combines all component before-state abstraction relations and WITHIN constraints to ensure common v, j witnesses can be found for all the constituent terms. The concession comes from a distribution of the disjunctions over the conjunction of the two postconditions $(G_1 \lor C_1) \land (G_2 \lor C_2)$, with the term corresponding to the combined abstraction relations removed. It can be shown that the above definition of composition of retrenchments is associative [17].

2.4 Choosing a formalism

There are some initial constraints on this work imposed by the two distinct formal frameworks (the B language of J.-R. Abrial [1] vs. the refinement calculus [Op.Cit.]) favoured by the authors. B is a "posit-and-prove" refinement method, where the refining syntax is designed from experience and the refinement relation between the two specifications is expressed as a proof obligation. The Refinement Calculus (as its name suggests) embodies a calculational style of stepwise refinement through pattern-matching application of refinement rules to the original specification.

For algorithm refinement B is restrictive in the sense that operation sequence and loop (i.e. algorithmic structure) may only be introduced as a "big bang" in one final refinement step to implementation. Ways to circumvent this have been reported, e.g. [6, 11], but these are complicated to use.

Various approaches to this dilemma are being considered, including expressing retrenchment in the context of the refinement calculus, expressing specification constructors in B along with the capability to do algorithmic transformations, and working directly in terms of relational or weakest precondition semantics.

3 Formalising evolution

Consider a refinement perspective on the evolution problem. The refinement-based development (Fig. 1) of a component is depicted as the vertical transformation **rf** of its specification, whereas its evolution is depicted as horizontal transformation **ev** of the specification. Assume a legacy development, where $ASpec \sqsubseteq CSpec$ (is refined by) is the top of some sequence of refinements down to code, and an evolution of specifications $ASpec \to ASpec^*$. In the figure, the given elements are the three vertices $ASpec$, $ASpec^*$, $CSpec$ and the two solid arrows. Unknown elements to be established (indicated by "?") are the dotted arrows and the bottom-right hand vertex $CSpec^*$.

The major question in this "square completion" problem is how to generate a concrete counterpart $CSpec^*$ to the evolved specification $ASpec^*$, the corresponding evolved refinement $ASpec^* \sqsubseteq CSpec^*$, and the concrete model evolution $CSpec \to CSpec^*$. The latter transformation may appear to be a byproduct

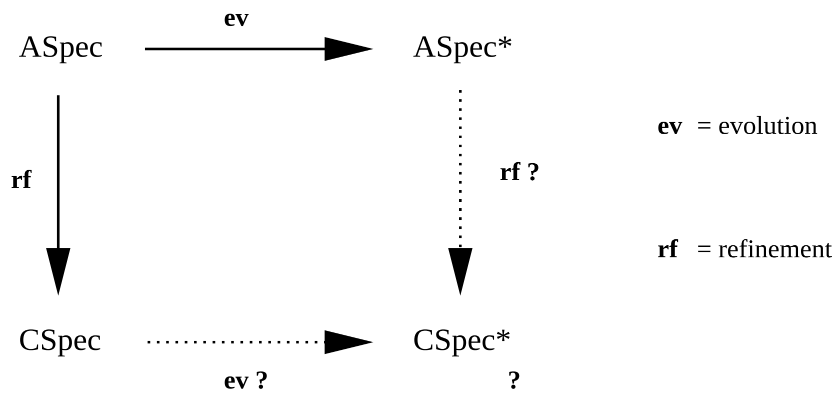

Fig. 1. The evolution square

of $ASpec \rightarrow CSpec$ and $ASpec \rightarrow ASpec^* \rightarrow CSpec^*$, but this is not the case in general. If the right-hand side of the square is completed first it is not necessarily obvious how to define the evolution $CSpec \rightarrow CSpec^*$: a whole class of retrenchments may be applicable. In fact the degrees of freedom in the square completion problem given by the fact that there are three basic unknowns (two dotted arrows and $CSpec^*$) to be found, afford a number of approaches to its solution. This paper will outline proposals for such approaches, and examine two of them more closely. A secondary question is then whether the square commutes in any meaningful and useful sense.

Any procedure answering these questions should reuse the design investment in the legacy refinement step $ASpec \sqsubseteq CSpec$. This step may take many forms, e.g. a data refinement between abstract models introducing more concrete structure, or an algorithm refinement. The step incorporates design effort, and the aspiration of an evolutionary method is to propagate the changes through the existing sequence of refinements in a manner which saves effort over refining the evolved specification from scratch. Such a procedure should be transitive in terms of further evolutions horizontally to $ASpec^{**}$ etc., and transitive vertically through refinement to code.

A retrenchment approach to completion and commutativity of the square is considered in sections 3.2 - 3.3. A more fundamental constructive approach is taken in section 4: it may be possible to construct the evolution $CSpec \rightarrow CSpec^*$ directly, as some derivation of the abstract evolution $ASpec \rightarrow ASpec^*$. This will be particularly useful if the legacy refinement $ASPec \sqsubseteq Cspec$ is a long sequence of small refinement steps.

3.1 Completing the square

In Fig. 1 each arrow must denote a well-defined transformation of a specification, or specification element[1]. These transformations must compose. Both refinement and retrenchment compose transitively, and are thus suitable models for the horizontal evolution transformation `ev` in the figure. Refinements can be included in a retrenchment framework since every refinement is expressible as a retrenchment - see e.g. [3].

The phrase "completing the evolution square" is deceptively simple: in general there will be many possible square completions, and in pathological cases there will be none. Apart from the usual design constraints which drive the selection of a refinement, there may be other requirements driving the choice of square completion. A categorical investigation is under way [10], seeking a very abstract form of square completion (maximal within a certain class of completions) under a retrenchment model.

The problem is simplified where one or more of the unknown elements is in fact known; such prespecified elements may be requirements of the evolution. For example, the syntactic change from $ASpec$ to $ASpec^*$ may be modest, and the evolved refinement relation may be a known requirement of the evolution. The requirement may be to strengthen the refinement. In this case the class of candidates for $CSpec^*$ is precisely those specifications in the concrete space that refine $ASpec^*$ subject to the new refinement relation. The most nondeterministic of these candidates is of course Morgan and Gardiner's least refinement construction [13] (called, say, $CSpec^+$), with the distinguishing characteristic that any $CSpec^*$ that refines $ASpec^*$, also refines $CSpec^+$.

The least refinement scenario suggests the possibility of relating two distinct square completions: for example, two constructions may refine $ASpec^*$ to $CSpec_1^*$ and $CSpec_2^*$ respectively, such that $CSpec_1^* \sqsubseteq CSpec_2^*$. Then the square $(ASpec, CSpec_2^*)$ is in some sense a "subsquare" of $(ASpec, CSpec_1^*)$. If the concrete specifications have the same signature then the least refinement $CSpec^+$ gives the "greatest" such subsquare in the concrete space. Such square relationships may be helpful in choosing a square completion to meet design requirements.

A number of structural approaches may be taken to the general square completion problem (with three unknowns). An approach seeking a monotonic, or "almost" monotonic transformation of the legacy refinement $ASpec \sqsubseteq CSpec$ is discussed in [9]. This approach depends of course on the existence of monotonic transformers to describe the evolution required; it may be possible to decompose a large (in terms of the amount of syntactic change involved) evolution into a sequence of smaller ones more tractable to such an approach. On the other hand, this approach may require a syntactic similarity between the two specifications that may not exist.

[1] As indicated before, we will work at the "element", i.e. statement/operation level.

A conceptually similar approach would be to transform the retrenchment describing the horizontal evolution. Here, the retrenchment $ASpec \lesssim ASpec^*$ must be transformed to $CSpec \lesssim CSpec^*$ whilst preserving the legacy refinement $ASpec \sqsubseteq CSpec$. Initial work [17] has shown not only that monotonic retrenchment transformers exist, but that they are easier to design than refinement transformers, because of the flexibility afforded by the defining parameters of a retrenchment. The downside is that easily devised transformers may produce weak retrenchments of little design utility.

Another way to view this approach is to say that there is a class (yet to be established) of retrenchment transformers that preserve the legacy refinement. The design task is to choose one (if any) that meets requirements. This approach remains to be studied.

3.2 Commuting the square

We will consider a little more closely the question of the commutativity of the square in Fig. 1. The use of the term "commutes" depends on any requirements we have of the relationship between the square traversals $ASpec \rightarrow ASpec^* \rightarrow CSpec^*$ and $ASpec \rightarrow CSpec \rightarrow CSpec^*$. The usual interpretations are that the two traversals are logically equivalent (strong interpretation), or that the lower $ASpec \rightarrow CSpec \rightarrow CSpec^*$ traversal is logically weaker (weaker interpretation).

Where the evolutions can be described as refinements, then the square can be said to commute strongly when the two traversals are equivalent as refinements, i.e. share the composite abstraction relation. It is less obvious how to define weak square commutativity.

Describing evolutions as retrenchments (which is in principle always possible), strong commutativity can be defined similarly, in terms of logical equivalence of the defining predicate parameters for each composite retrenchment. That is, for the upper and lower composite retrenchments parameterised by (G_{cu}, W_{cu}, C_{cu}) and (G_{cl}, W_{cl}, C_{cl}) respectively, strong commutativity is defined

$$G_{cu} \Leftrightarrow G_{cl} \wedge W_{cu} \Leftrightarrow W_{cl} \wedge C_{cu} \Leftrightarrow C_{cl} \tag{4}$$

Weak commutativity can be defined where the lower traversal is a logical consequence of the upper traversal. Weak commutativity requires that $G_{cu} \Rightarrow G_{cl}$, that $W_{cl} \Rightarrow W_{cu}$, and that $C_{cu} \Rightarrow C_{cl}$. These conditions for the commutativity notions are strong; investigation is required to see how often they arise in practice, and whether weaker, more practical commutativity notions will be required.

When the two horizontal evolutions $ASpec \rightarrow ASpec^*$, $CSpec \rightarrow CSpec^*$ can be considered together, as a construction on the legacy refinement $ASpec \sqsubseteq CSpec$, this completes the square[2]. In this case, as indicated above, each square

[2] Aside: it also gives the square a different meaning; the horizontal evolution then represents a homomorphism over the refinement lattice containing $ASpec$ and $CSpec$. We do not pursue this view in this work.

traversal may be modelled as a composition of retrenchments, or even refinements (for refining evolutions). Commutativity can then be considered.

3.3 Commutativity analysis: an example

We now consider a simple example evolution to demonstrate a simple commutativity approach. The legacy abstract specification fragment *ASpec* is a simple resource allocation operation which, given an input resource *rs* of appropriate type, adds that resource to a set *arss* of allocated resources. Its refinement *CSpec* is more concrete, representing the set as a sequence *crss*, and modelling allocation as sequence append. The specifications are given in the abstract sytax of B, whose meaning is given by the relational stp models for these specifications. The abstraction relation G defining the refinement is also given:

$$\begin{aligned} Alloc_A &\mathrel{\widehat=} rs \in RSS \mid arss := arss \cup \{rs\} \\ Alloc_C &\mathrel{\widehat=} rs \in RSS \mid crss := crss \frown [rs] \\ G(arss, crss) &\mathrel{\widehat=} arss = \operatorname{ran} crss \wedge arss \in \mathbb{P}\, RSS \wedge crss \in \mathsf{seq} RSS \\ \mathsf{stp}_A(arss, arss') &\mathrel{\widehat=} rs \in RSS \wedge arss' = arss \cup \{rs\} \\ \mathsf{stp}_C(crss, crss') &\mathrel{\widehat=} rs \in RSS \wedge crss' = crss \frown [rs] \end{aligned} \tag{5}$$

Next an evolutionary change is specified. It is now required to be more discriminating in allocation: any input resource *rs* not already allocated will be, but any *rs* already allocated will result in skip. The evolution here is of the refinement as well as the abstract specification: it is required to strengthen the abstraction relation to force a more efficient concrete representation: an injective sequence. This is ensured, at concrete level, by skipping where the resource is already allocated, i.e. an evolution of the concrete specification which mirrors the abstract one in a construction of the same syntactic shape:

$$\begin{aligned} Alloc_{A^*} &\mathrel{\widehat=} rs \in RSS \mid (rs \notin arss^* \Longrightarrow arss^* := arss^* \cup \{rs\} \\ &\qquad\qquad\qquad [] rs \in arss^* \Longrightarrow \mathsf{skip}) \\ Alloc_{C^*} &\mathrel{\widehat=} rs \in RSS \mid (rs \notin \operatorname{ran} crss^* \Longrightarrow crss^* := crss \frown [rs] \\ &\qquad\qquad\qquad [] rs \in \operatorname{ran} crss^* \Longrightarrow \mathsf{skip}) \\ G^*(arss^*, crss^*) &\mathrel{\widehat=} arss^* = \operatorname{ran} crss \wedge arss^* \in \mathbb{P}\, RSS \wedge crss{*} \in \mathsf{iseq} RSS \\ \mathsf{stp}_{A^*}(arss^*, arss^{*\prime}) &\mathrel{\widehat=} rs \in RSS \wedge arss^{*\prime} = arss^* \cup \{rs\} \\ \mathsf{stp}_{C^*}(crss^*, crss^{*\prime}) &\mathrel{\widehat=} rs \in RSS \wedge (rs \notin \operatorname{ran} crss^* \Rightarrow crss^{*\prime} = crss^* \frown [rs]) \\ &\qquad \wedge (rs \in \operatorname{ran} crss^* \Rightarrow crss^{*\prime} = crss^* \end{aligned} \tag{6}$$

By simple set theory, the evolution $ASPec \rightarrow ASpec^*$ is the identity refinement with the following abstraction relation including type invariants:

$$G_A(arss, arss^*) \mathrel{\widehat=} arss = arss^* \wedge arss, arss^* \in \mathbb{P}\, RSS \tag{7}$$

The evolution $CSpec \rightarrow CSpec^*$ can be described as a retrenchment. The abstraction relation is the identity. Within and concession clauses are always design choices to be made in retrenchment. In this simple example true WITHIN clauses can always be used, here offering two obvious candidate concessions:

$$\begin{aligned} G_C(crss, crss^*) &\mathrel{\widehat{=}} crss = crss^* \wedge crss \in \mathsf{seq} RSS \wedge crss^* \in \mathsf{iseq} RSS \\ C_{Cnd} &\mathrel{\widehat{=}} crss = crss^* \vee crss = crss^* \frown [rs] \\ C_C &\mathrel{\widehat{=}} (rs \notin \operatorname{ran} crss_0^* \Rightarrow crss = crss^*) \\ &\qquad \wedge (rs \in \operatorname{ran} crss_0^* \Rightarrow crss = crss^* \frown [rs]) \end{aligned} \tag{8}$$

Concession C_{Cnd} (named for Concrete, non-discriminating) does not discriminate between the two possibilities for the paired steps taken by $Alloc_C$ and $Alloc_{C^*}$: either both update the state identically, or $crss$ is appended when $crss^*$ is skipped. We will use the more discriminating C_C which describes, in terms of before-state variables (denoted by zero-subscripts), exactly when these two cases arise.

Note that each of the three refinements identified is a retrenchment with $W \equiv \mathsf{true} \wedge C \equiv \mathsf{false}$. In order to consider commutativity we can thus apply theorem (3) to each of the square traversals. To summarise we observe that for the upper and lower traversals respectively

$$\begin{aligned} Alloc_A &\lesssim Alloc_{A^*} \quad \text{w.r.t.} \quad (\mathsf{true}, G_A, \mathsf{false}) \\ Alloc_{A^*} &\lesssim Alloc_{C^*} \quad \text{w.r.t.} \quad (\mathsf{true}, G^*, \mathsf{false}) \end{aligned} \tag{9}$$

$$\begin{aligned} Alloc_A &\lesssim Alloc_C \quad \text{w.r.t.} \quad (\mathsf{true}, G, \mathsf{false}) \\ Alloc_C &\lesssim Alloc_{C^*} \quad \text{w.r.t.} \quad (\mathsf{true}, G_C, C_C) \end{aligned} \tag{10}$$

The composite upper traversal is therefore a refinement with abstraction relation

$$\begin{aligned} G_u(arss, crss^*) &\mathrel{\widehat{=}} \exists\, arss^* \bullet (G_A \wedge G^*) \\ &\equiv arss = \operatorname{ran} crss^* \wedge arss \in \mathbb{P}\, RSS \wedge crss^* \in \mathsf{iseq} RSS \end{aligned} \tag{11}$$

The composite lower traversal is a retrenchment with

$$\begin{aligned} W_l &\mathrel{\widehat{=}} \mathsf{true} \\ G_l(arss, crss^*) &\mathrel{\widehat{=}} \exists\, crss \bullet (G \wedge G_C) \\ &\equiv arss = \operatorname{ran} crss^* \wedge arss \in \mathbb{P}\, RSS \wedge crss^* \in \mathsf{iseq} RSS \\ &\equiv G_u \\ C_l(crss_0^*, arss, crss^*, rs)& \\ &\mathrel{\widehat{=}} \exists\, crss \bullet (G \wedge C_C) \\ &\equiv \exists\, crss \bullet (G(arss, crss) \wedge rs \notin \operatorname{ran} crss_0^* \wedge crss = crss^*) \\ &\vee \exists\, crss \bullet (G(arss, crss) \wedge rs \in \operatorname{ran} crss_0^* \wedge crss = crss^* \frown [rs]) \\ &\equiv G(arss, crss^*) \wedge rs \notin \operatorname{ran} crss_0^* \\ &\quad \vee G(arss, crss^* \frown [rs]) \wedge rs \in \operatorname{ran} crss_0^* \end{aligned} \tag{12}$$

In terms of the commutativity discussion of section 3.2, simple composition of retrenchments shows that the square commutes weakly: both traversals share the abstraction relation G_u, and the lower traversal is weaker by a nontrivial concession. We bear in mind experience [4] that retrenchment is a conservative theory, and often results in weaker statements than can be made by further deployment of information available. This is because of the proliferation of disjuncts in the composition of retrenchments. Here, the concession C_l misses the fact that $CSpec^*$ is designed to keep $crss$ injective.

Reasoning with information available outside the retrenchment framework allows us to strengthen the square to strong (equivalent) commutativity. In the expanded POB (proof obligation) for the lower traversal, G_l gives $crss^* \in \mathsf{iseq} RSS$ as a hypothesis. To strengthen C_l to G_l in the postcondition consider its two before-state conditions on rs in turn. If $rs \notin \operatorname{ran} crss^*$ then $crss^{*\prime} = crss^* \frown [rs] \in \mathsf{iseq} RSS$ by the property of sequences, strengthening C_l as required. If $rs \in \operatorname{ran} crss^*$ then $crss^{*\prime} = crss^* \in \mathsf{iseq} RSS$. We have

$$\begin{aligned} G(arss, crss^* \frown [rs]) \equiv arss &= \operatorname{ran}(crss^* \frown [rs]) \\ &= (\operatorname{ran} crss^*) \cup \{rs\} = \operatorname{ran} crss^* \end{aligned}$$

which again strengthens C_l to G_l as required.

4 Describing evolutionary modifications

An evolutionary transformation may change a specification in quite arbitrary ways. It is useful, however, to consider a number of elementary modification forms that can be combined to describe more complex transformations. If we consider a specification expressed as a specification statement, $w : [P\ ,\ R]$, we can consider six simple kinds of modification (ignoring for now transformations that alter the types of variables):

- Increase/decrease the frame.
- Strengthen/weaken the precondition.
- Strengthen/weaken the postcondition.

It is easy to see that an arbitrary modification can be described using a combination of these: To turn $w_1 : [P_1/R_1]$ into $w_2 : [P_2/R_2]$, we remove all of the variables in w_1 from the frame, strengthen the precondition to *false* and weaken the postcondition to *true*, then add the variables in w_2 to the frame, weaken the precondition to P_2 and strengthen the postcondition to R_2. This is not generally a useful way to describe a modification as it discards all the information we had about the initial specification. Usually we will seek to describe a modification in a way that preserves as much of the original specification as we can and makes the smallest changes we can in order to get the required new specification.

A refinement may involve decreasing the frame, weakening the precondition or strengthening the postcondition; the other modifications do not give refinements. Increasing and decreasing the frame can be considered to be special cases of weakening and strengthening postconditions (provided the signature of the specification is unchanged, otherwise they are special forms of data refinement), so do not need to be considered separately.

The remaining four primitive modifications can be described more formally, as follows:

- Weaken precondition: $w : [P\ ,\ R] \longrightarrow w : [Q\ ,\ R]$, where $P \Rightarrow Q$.
- Strengthen precondition: $w : [P\ ,\ R] \longrightarrow w : [Q\ ,\ R]$, where $Q \Rightarrow P$.
- Weaken postcondition: $w : [P\ ,\ R] \longrightarrow w : [P\ ,\ Q]$, where $R \Rightarrow Q$.
- Strengthen postcondition: $w : [P\ ,\ R] \longrightarrow w : [P\ ,\ Q]$, where $Q \Rightarrow R$.

In investigating properties of these modifications, it is more convenient to describe them as operators that make explicit the change being performed. To facilitate this, we first note that $A \Rightarrow B$ iff $B \equiv A \vee C$, for some predicate C, and $B \Rightarrow A$ iff $B \equiv A \wedge D$, for some predicate D. We may impose further restrictions on C and D, for example, that A and C are mutually exclusive ($C \Rightarrow \neg A$) and that D does not allow any states excluded by A ($D \Rightarrow A$). We can thus define the required operators, as applied to an arbitrary specification statement, as follows:

- Weaken precondition: $w : [P\ ,\ R]\ \mathrm{pre}^{-}_{P'} \mathrel{\widehat{=}} w : [P \vee P'\ ,\ R]$
- Strengthen precondition: $w : [P\ ,\ R]\ \mathrm{pre}^{+}_{P'} \mathrel{\widehat{=}} w : [P \wedge P'\ ,\ R]$
- Weaken postcondition: $w : [P\ ,\ R]\ \mathrm{post}^{-}_{R'} \mathrel{\widehat{=}} w : [P\ ,\ R \vee R']$
- Strengthen postcondition: $w : [P\ ,\ R]\ \mathrm{post}^{+}_{R'} \mathrel{\widehat{=}} w : [P\ ,\ R \wedge R']$

We can now investigate commuting properties of refinements and evolutionary modifications by comparing the two paths from *ASpec* to *CSpec** in Fig. 1, where **rf** and **rf?** are refinement transformations. This task is complicated by the number of cases that need to be considered and the variety of ways in which **rf?** and **ev?** can differ from **rf** and **ev**. As a modest start to such an investigation, and to illustrate the kinds of results we anticipate, we will consider the cases where **rf?** is identical to **rf** and **ev?** is identical to **ev**. Thus, for each pair of primitive modifications (**rf**, **ev**), where **rf** is a refinement transformation, we consider the effects of the compositions **rf** $\fatsemi$ **ev** and **ev** $\fatsemi$ **rf** on an arbitrary specification statement.

The following cases are easy to prove:

(1) $\mathrm{pre}^{-}_{P'} \fatsemi \mathrm{pre}^{-}_{P''} = \mathrm{pre}^{-}_{P''} \fatsemi \mathrm{pre}^{-}_{P'}$

(2) $\text{pre}^-_{P'} \mathbin{;} \text{post}^-_{R'} = \text{post}^-_{R'} \mathbin{;} \text{pre}^-_{P'}$

(3) $\text{pre}^-_{P'} \mathbin{;} \text{post}^+_{R'} = \text{post}^+_{R'} \mathbin{;} \text{pre}^-_{P'}$

(4) $\text{post}^+_{R'} \mathbin{;} \text{pre}^-_{P'} = \text{pre}^-_{P'} \mathbin{;} \text{post}^+_{R'}$

(5) $\text{post}^+_{R'} \mathbin{;} \text{pre}^+_{P'} = \text{pre}^+_{P'} \mathbin{;} \text{post}^+_{R'}$

(6) $\text{post}^+_{R'} \mathbin{;} \text{post}^+_{R''} = \text{post}^+_{R''} \mathbin{;} \text{post}^+_{R'}$

Cases (1) and (6) hold because they make the same change to the same component, so hold because $\vee$ and $\wedge$ are both commutative. The other cases hold because they make changes to different components so the order in which the changes are performed makes no difference.

The remaining two cases involve making both a strengthening and a weakening modification to the same component. In these cases the order of application makes a difference, and we get a refinement between the two orders.

(7) $\text{pre}^-_{P'} \mathbin{;} \text{pre}^+_{P''} \sqsubseteq \text{pre}^-_{P'} \mathbin{;} \text{pre}^+_{P''}$

$$
\begin{aligned}
& (w : [P\ ,\ R]\ \text{pre}^-_{P'})\ \text{pre}^+_{P''} \\
= {} & w : [P \vee P'\ ,\ R]\ \text{pre}^+_{P''} \\
= {} & w : [(P \vee P') \wedge P''\ ,\ R] \\
= {} & w : [P \wedge P'' \vee P' \wedge P''\ ,\ R] \\
\sqsubseteq {} & w : [P \wedge P'' \vee P'\ ,\ R] \\
= {} & w : [P \wedge P''\ ,\ R]\ \text{pre}^-_{P'} \\
= {} & (w : [P \wedge P''\ ,\ R]\ \text{pre}^+_{P''})\ \text{pre}^-_{P'}
\end{aligned}
$$

(8) $\text{post}^+_{R'} \mathbin{;} \text{post}^-_{R'} \sqsubseteq \text{post}^+_{R'} \mathbin{;} \text{post}^-_{R'}$

The proof is similar to (7).

The evolutionary modifications performed in practice often combine two or more of these simple modifications. For example, when weakening a precondition as an evolutionary change, we usually also want to specify the results to be produced for the additional initial states for which the program is now required to terminate. In the search example outlined in Section 2.1, when adding initial states where there is no value in $1..n$ satisfying p, we also added the requirement that k should be a value outside the range $1..n$. Similarly, when strengthening a postcondition, we might also have to strengthen the precondition to ensure that the resulting postcondition is satisfiable. For example, in the search example we might add the requirement that k be an odd number to the postcondition, in which case we would also add an assumption to the precondition that there is at least one odd number in $1..n$ satisfying p. The combinators used in [9] to describe modifications are designed to allow such modifications to be described easily. The above results should be readily extended to these combinators.

In order to carry these results into algorithmic development, we also need to consider the case where some of the specifications are given in an algorithmic form. In this case, we want to be able to preserve as much of the algorithmic structure where this is possible and convenient. A wide range of such results are given in [9], using a selection of combinators to describe modifications, and we are attempting to extend these to cover the more general modifications that can be described as retrenchments.

5 Conclusion

This paper has outlined various formal approaches to the problem of software evolution. These approaches, deriving from the authors' respective interests, have been described as refinement-based/constructive, and retrenchment respectively. One of the aims of this work in progress is the combination of these approaches.

All the forms of modification discussed in section 4 are expressible as retrenchments, and it is expected that syntactic patterns will emerge in the retrenchments corresponding to these forms. Such patterns should be useful in the proposed analyses of square completion and commutativity. The compositional nature of this constructive approach, where modifications may easily be applied in sequence, fits well with the retrenchment approach. As indicated in section 3.3, further reasoning is required outside the simple composition of retrenchments in order to prove useful properties. Such reasoning support may be provided by the constructive approach. Section 3.1 speculated about other potential retrenchment approaches to completion of the square; it is expected that these would combine with the constructive approach in similar ways.

Retrenchment has to date been concerned with relationships between abstract specifications, with strong emphasis on describing change in variables and types between such specifications. Since it is based on the forward simulation proof rule, which proves data as well as algorithm refinements, retrenchment is in principle capable of relating algorithmic specifications. This has not yet been examined, and may need some work: an algorithmic operation contains a lot of information to be collapsed into one concession predicate, an effect exacerbated when two algorithmic operations are related through the concession. This may be an application for *decomposition* of retrenchments [15], where multiple retrenchment relations (using distinct WITHIN clauses) may be expressed between a given pair of operations. This approach decomposes one complex proof obligations into a small number of stronger, simpler proof obligations, enhancing understandability, and tractability to theorem provers.

References

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

[2] 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.

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

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

[5] R. Ben Ayed, J. Desharnais, M. Frappier, and A. Mili. A calculus of program adaptation and its applications. *Science of Computer Programming*, 38:73–123, 2001.

[6] M. Butler and Mairead Meagher. Performing algorithmic refinement before data refinement in B. In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *First International Conference of B and Z Users*, volume 1878 of *LNCS*, pages 324–343, York, UK, August 2000.

[7] Lindsay Groves. Deriving programs by combining and adapting refinement scripts. In *Proc. 1995 Asia Pacific Software Engineering Conference*, pages 354–363. IEEE Computer Society Press, 1995.

[8] Lindsay Groves. Adapting program derivations using program conjunction. In Jim Grundy, Martin Schwenke, and Trevor Vickers, editors, *International Refinement Workshop and Formal Methods Pacific 1998 (IRW/FMP'98)*, pages 145–164, Canberra, 1998. Springer-Verlag.

[9] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[10] C. Jeske and R. Banach. Reconciling retrenchments and refinements. *in preparation*, 2002.

[11] L. Klenov and L. Pierre. Refining iterative programs in the B method. In M. Butler and T. Muntean, editors, *RCS'02 - International Workshop on Refinement of Critical Systems*, IMAG, Grenoble, January 2002. http://www.esil.univ-mrs.fr/ spc/rcs02/rcs02.html.

[12] C.C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[13] C.C. Morgan and T. Vickers. *On The Refinement Calculus*. FACIT. Springer, 1994.

[14] 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.

[15] M. Poppleton and R. Banach. Structuring retrenchments in B by decomposition. Accepted for FM2003, Pisa, September 2003.

[16] M. Poppleton and L. Groves. Formal perspectives on software evolution: From refinement to retrenchment. Submitted for publication; invited chapter, 2003.

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