Adapting Program Derivations using Program Conjunction

Lindsay Groves

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
lindsay@mcs.vuw.ac.nz

Abstract. Program refinement is usually seen as an idealised form of program development in which a formal specification is transformed into a correct implementation. In order to handle program modification within the refinement calculus, we need to be able to take a modified specification and adapt the existing derivation to obtain a derivation for the new specification. We show that many typical specification changes can be expressed using a form of program conjunction. We also show how suitable properties of program conjunction can be used to propagate a specification change through an existing derivation and identify places where changes are required.

1 Introduction

Most discussions of program refinement (e.g. [14]) assume that program development begins with a complete formal specification, which is turned into an executable program via a sequence of correctness preserving transformations. This idealistic view of programming ignores many aspects of the way in which software is actually developed. To obtain a more realistic theory, we need to consider other aspects of program evolution, such as incremental development, maintenance and reuse.

In this paper, we consider a particular aspect of program evolution, where an existing program is modified to implement an augmented specification imposing some additional requirement. This situation arises, for example: in incremental development, where we implement a program to fulfil some of our requirements then extended it to address the others; in software maintenance, where modifications often arise because of new requirements; and in software reuse, where we use an existing program or program component as a basis for constructing another program.

If the new specification is constructed by composing the existing specification with another specification fragment, so the new specification has the old one as a component, we can obtain a derivation for the new specification by propagating the new requirement through the existing derivation, identifying places where changes are required. How much of the existing derivation we are able to reuse, and how it is modified, will depend partly on how "big" a change the new specification requires, and partly on on other desired properties of the implementation (and derivation). We want the programmer to be able to preserve as much (or as little) of the existing derivation as is deemed desirable, and to insert, delete or modify steps as required, to obtain any desired implementation. In particular, we want to be able to experiment with different ways of satisfying the combined specification, and to be able to produce familiar algorithms as implementations. At one extreme, the existing program may already satisfy the new requirement (e.g. because of the way it resolved nondeterminism), in which case no change is required; at the other extreme, we may need to abandon the existing derivation and start again from scratch. Indeed, the new requirements may be inconsistent with the existing ones, in which case the new specification cannot be implemented.

We begin by considering some of the ways in which a specification might be augmented, showing that a form of conjunction is often required (Section 2). We then introduce our version of program conjunction and discuss its definition (Section 3), and present some of its properties (Section 4). Next, we show how these properties can be used to propagate new requirements through an existing derivation as suggested above, illustrating the approach with a simple example (Section 5). We conclude by discussing some related work (Section 6), and summarising what we have achieved and what remains to be done (Section 7).

2 Composing Specifications using Conjunction

Specifications can be combined to form new specifications in a variety of ways. The Z schema calculus [16] provides various mechanisms for composing specifications, and has provided much of the inspiration for adding specification composition operators to other formal specification and development notations (e.g. [18], [13]). In this section, we present some examples to illustrate the kinds of specification compositions that may arise in program evolution, and show that many of them can be expressed using a form of conjunction.

Suppose we have a program to find the smallest number, lo, in a sequence, A, of numbers. We can express this as a specification statement [14], using $\#A$ for the length of sequence A and $\sqcap A$ for its minimum:

$$lo : [\, \#A \geq 1 \;/\; lo = \sqcap A \,] \qquad (i)$$

This program might evolve into a larger, more complex one, in a number of ways. For instance, we might modify the specification in the following ways:

- Add a new output. E.g. as well as finding the minimum, we might want to find the maximum of A, the position of an occurrence of the minimum, or the number of occurrences of the minimum. This has the effect of increasing the dimensionality of the state space, and will always require changes in the implementation.
- Constrain the output(s). E.g. having extended the above specification to find an occurrence of the minimum, we might constrain this so as to find the first

such occurrence. This has the effect of reducing demonic non-determinism in the specification, and thus reducing implementation options available — the existing program might already be implemented in a way that satisfies the additional constraint.

- Constrain the input(s). E.g. we might add an assumption that A is in ascending order, or that all elements of A are within a specified range. This has the effect of reducing the size of the input domain for which the program is required to terminate — this will not require that the program be changed, since the existing implementation will still be correct, but might offer opportunities to simplify the program.

In each of these cases, we can form the new specification by combining the old specification with a specification expressing the new requirement. Considering how we would do this in Z suggests that in each case the combination required is some kind of conjunction.

Writing $\owedge$ for the conjunction of two specifications, we might write a selection of the specifications described above as follows:

Example 1: Modify (i) to find the maximum.

$$lo\,\mathbf{:}[\,\#A \geq 1 \;/\; lo = \sqcap A\,] \owedge hi\,\mathbf{:}[\,\#A \geq 1 \;/\; hi = \sqcup A\,]$$

Example 2: Extend (i) to locate an occurrence of the minimum.

$$lo\,\mathbf{:}[\,\#A \geq 1 \;/\; lo = \sqcap A\,] \owedge p\,\mathbf{:}[\,\#A \geq 1 \;/\; A[p] = lo\,] \qquad (ii)$$

Example 3: Extend (i) to count occurrences of the minimum.

$$lo\,\mathbf{:}[\,\#A \geq 1 \;/\; lo = \sqcap A\,] \owedge c\,\mathbf{:}[\,\#A \geq 0 \;/\; c = lo\#A\,]$$

where the notation $x\#A$ is used to mean "the number of occurrences of x in A". Note that counting is defined when $\#A \geq 0$, while finding the minimum requires $\#A \geq 1$. We can only do both when $\#A \geq 1$.

Example 4: Constrain the output of (ii) so as to locate the first occurrence of the minimum.

$$lo\,\mathbf{:}[\,\#A \geq 1 \;/\; lo = \sqcap A\,] \owedge p\,\mathbf{:}[\,\#A \geq 1 \;/\; A[p] = lo\,] \owedge [\,lo \notin A[1\,..\,p-1]\,]$$

Example 5: Constrain the input of (i) so that A is in ascending order.

$$lo\,\mathbf{:}[\,\#A \geq 1 \;/\; lo = \sqcap A\,] \owedge \{\,asc(A)\,\}$$

In the above examples, we could have incorporated the new requirement into the existing specification, to obtain a single specification statement expressing the combined requirements. For examples 1 and 2, this would give:

$$lo, hi : [\, \#A \geq 1 \;/\; lo = \sqcap A \;\wedge\; hi = \sqcup A \,]$$
$$lo, p : [\, \#A \geq 1 \;/\; lo = \sqcap A \;\wedge\; A[p] = lo \,]$$

However, this blurs the distinction between the old specification and the new requirements, and would become cumbersome in more complex examples.

There are many similar situations where conjunction might be used to express an additional requirement corresponding to an evolutionary step. For example, having derived a sort program, we might impose an additional constraint that the sort be stable [7]; having derived a language recognition program, we might extend this to also construct a parse tree or translation for the input recognised.

We may, of course, wish to combine specifications in various other ways, and again the Z schema calculus provides good examples of the kinds of facilities we might consider. For example, we might compose specifications using sequential composition, or some kind of disjunction. We already have sequential composition (;) in the refinement calculus, and several forms of disjunction (demonic and angelic choice, with or without an enclosing **if** ... **fi**). We also have features that can be used to obtain the same effect as hiding (local variables and constants) and renaming (substitution).

Since conjunction appears to be especially useful in the kinds of situations we are interested in, and is not already supported in the refinement calculus, we will concentrate on it in this paper. However, similar techniques can be used with other forms of composition.

3 Defining Conjunction

Several authors have described conjunction-like operators for combining programs (e.g. [1], [2], [4], [5], [9], [10], [11], [13], [15], [17], [18]). In this section, we motivate the definition we use, by formalising the intuition underlying the examples in Section 2. Some relationships with other operators are discussed in Section 6.

In the examples given in Section 2, our intuition was that $S \owedge T$ specifies a program that would somehow "Do what S says and what T says". We might, for example, think of S and T proceeding in "parallel", on separate state spaces, and their effects then being combined. We now attempt to make this intuition more precise.

Consider first the case where S and T are both specification statements, say, $w_1 : [\, P_1 \;/\; R_1 \,]$ and $w_2 : [\, P_2 \;/\; R_2 \,]$.

We only want to insist that $S \owedge T$ terminate when both preconditions hold (consider the example of finding the minimum of a sequence and the number of times it occurs), so we will take the precondition of $S \owedge T$ to be $P_1 \wedge P_2$.

We want to allow $S \owedge T$ to modify any variables that either S or T can modify, so we take the frame of the $S \owedge T$ to be $w_1 \cup w_2$ (which we write as w_1, w_2 below).

Finally, we want $S \owedge T$ to establish both postconditions, constraining the behaviour of each specification to be consistent with the other, so the postcondition of $S \owedge T$ should be $R_1 \wedge R_2$.

Thus, we want the following relationship to hold:

$$w_1 \mathbin{:}[P_1 \;/\; R_1] \circledA w_2 \mathbin{:}[P_2 \;/\; R_2] \quad \equiv \quad w_1, w_2 \mathbin{:}[P_1 \wedge P_2 \;/\; R_1 \wedge R_2]$$

This has the desirable consequence that conjunction is monotonic (see Section 4), unlike Z's schema conjunction — if consistent behaviour is not possible (i.e. if R_1 and R_2 conflict), the conjunction is miraculous.

Restricted versions of this operator were defined by Mahony [12], who required the frames of the conjoined specifications to be disjoint, and Utting [17], who required the frames to be identical.

In defining a conjunction operator that works for programs with different frames, we have to be careful to ensure that variables occurring in only one frame are handled correctly. For example, according to the above equation, we have:

$$x \mathbin{:}[x = 0] \circledA y \mathbin{:}[\mathit{true} \;/\; y = 1] \equiv x, y \mathbin{:}[x = 0 \wedge y = 1]$$

But, by Morgan's *Expand Frame* law ([14], p303), we also have:

$$\begin{aligned} & x \mathbin{:}[x = 0] \circledA y \mathbin{:}[y = 1] \\ \equiv\ & x, y \mathbin{:}[x = 0 \wedge y = y_0] \circledA x, y \mathbin{:}[y = 1 \wedge x = x_0] \\ \equiv\ & x, y \mathbin{:}[x = 0 \wedge y = y_0 \wedge y = 1 \wedge x = x_0] \\ \equiv\ & [x = 0 \wedge y = 1] \end{aligned}$$

which is miraculous, unless $x = 0 \wedge y = 1$ holds already.

Mahony [13] avoids this problem by defining a conjunction operator that specifies which variables each program needs to constrain. Mahony's definition extends a similar operator defined by Ward [18] (who called the operator *miraculous conjunction*, and introduced the symbol $\circledA$).

The definition is based on a typing system, which identifies the *signature* of a program (the set of all variables referenced within it), denoted by $sig(S)$, and the set of variables a program can change, denoted $frame(S)$. We write $S\colon u \overset{\oplus}{\longmapsto} z$ to mean that S is a program (more precisely, a monotonic predicate transformer) with signature u and frame z, where $z \subseteq u$. Mahony also defines two new operators.

The *generalised effect* of a program S gives "the strongest effect guaranteed by all conjunctive refinements of S" [13], and arises from a syntactic formulation of the least conjunctive refinement operator (*cf* [15]):

$$\mathbf{E}\ S \;\widehat{=}\; \neg wp(S, z \neq z')[z, z' \setminus z_0, z], \text{ where } S\colon u \overset{\oplus}{\longmapsto} z$$

The "opening" operator provides an "open world" view of the specified variables, so $S \overset{v}{\oplus} w$ specifies a program that implements S, and can change variables in $w - v$:

$$S \overset{v}{\oplus} w \;\widehat{=}\; S; w - v \mathbin{:}[\,]$$

Mahony writes $S\ {}_u\!\owedge_v\ T$ to mean that S and T are conjoined in a way that allows S to influence the variables in u and T to influence the variables in v:

$$S\ {}_v\!\owedge_w\ T\ \ \widehat{=}\ \ v, w :\left[pre(S)\ \wedge\ pre(T)\ \middle/\ \mathbf{E}\ (S \stackrel{v}{\oplus} w)\ \wedge\ \mathbf{E}\ (S \stackrel{w}{\oplus} v) \right],$$
where $S\colon u \overset{\oplus}{\longmapsto} v$ and $T\colon u \overset{\oplus}{\longmapsto} w$

This definition is identical to Ward's, except that with Ward's operator, each program assumes that any variable in its signature which it doesn't explicitly change must remain unchanged.

With Mahony's definition, we have, for example:

$$\begin{aligned}
& x :[\, x = y_0 \,]\ {}_x\!\owedge_y\ y :[\, y = x_0 \,] \\
&\equiv\ x :[\, x = y_0 \,]\,; y :[\ ]\ {}_x\!\owedge_y\ y :[\, y = x_0 \,]\,; x :[\ ] \\
&\equiv\ x, y :[\, x = y_0\ \wedge\ y = x_0 \,]
\end{aligned}$$

which conjoins the specifications in the way we expect, and is quite different from:

$$\begin{aligned}
& x :[\, x = y_0 \,]\ {}_{x,y}\!\owedge_{x,y}\ y :[\, y = x_0 \,] \\
&\equiv\ x, y :[\, x = y_0\ \wedge\ y = y_0 \,]\ {}_{x,y}\!\owedge_{x,y}\ x, y :[\, y = x_0\ \wedge\ x = x_0 \,] \\
&\equiv\ [\, x = y \,]
\end{aligned}$$

4 Properties of Conjunction

The conjunction operator has a number of properties which are useful in deriving programs. We state and explain some of these properties here, concentrating on properties that allow us to eliminate conjunctions or to distribute them over other program constructors.

We are primarily concerned here with illustrating the use of conjunction in program modification, and with identifying the properties required for this. We therefore give these properties without proof, and rely on operational intuitions and their intended use.

We also restrict our attention to properties involving conjunctive programs, since we are only concerned with the conjunction of two specifications, or of programs constructed using conjunctive operators.

In the following, unless stated otherwise, S, T and U are conjunctive programs with frames u, v and w, respectively, x and y are distinct program variables, P and R are predicates, and e and f are expressions.

Some of these laws can be strengthened, at the cost of more complicated conditions; for example, by distinguishing between the variables in the initial and final states (i.e. variables that are read but not changed).

4.1 Monotonicity

Conjunction is monotonic with respect to refinement:

$\owedge$*-mono:* $\quad S \;{}_u\!\owedge_v\; T \;\sqsubseteq\; S' \;{}_u\!\owedge_v\; T', \quad \text{if } S \;\sqsubseteq\; S' \text{ and } T \;\sqsubseteq\; T'$

This can be strengthened to allow termination of one program to be assumed in refining the other [5].

4.2 Refinement

The conjunction of two programs with the same frame (i.e. $u = v$) refines each of them, provided the other terminates:

$\owedge$*-ref-1:* $\quad \{\, pre(T) \,\}\, S \;\sqsubseteq\; S \;{}_u\!\owedge_u\; T$

$\owedge$*-ref-2:* $\quad \{\, pre(S) \,\}\, T \;\sqsubseteq\; S \;{}_u\!\owedge_u\; T$

More generally, each program can be seen as a kind of "slice" [19] of the conjunction:

$\owedge$*-ref-3:* $\quad \{\, pre(T) \,\}\, S \;\sqsubseteq\; [\![\; \mathbf{var}\ w \;\bullet\; S \;{}_u\!\owedge_v\; T \;]\!], \quad \text{where } w = v - u$

$\owedge$*-ref-4:* $\quad \{\, pre(S) \,\}\, T \;\sqsubseteq\; [\![\; \mathbf{var}\ w \;\bullet\; S \;{}_u\!\owedge_v\; T \;]\!], \quad \text{where } w = u - v$

4.3 Frame expansion

$\owedge$*-frame:* $\quad S \;{}_u\!\owedge_v\; T \;\sqsubseteq\; S \;{}_{u'}\!\owedge_{v'}\; T, \quad \text{if } u \subseteq u' \text{ and } v \subseteq v'$

4.4 Basic algebraic properties

Conjunction is idempotent, commutative and associative:

$\owedge$*-idem:* $\quad S \;{}_u\!\owedge_u\; S \;\equiv\; S$

$\owedge$*-comm:* $\quad S \;{}_u\!\owedge_v\; T \;\equiv\; T \;{}_v\!\owedge_u\; S$

$\owedge$*-assoc:* $\quad (S \;{}_u\!\owedge_v\; T) \;{}_{u,v}\!\owedge_w\; U \;\equiv\; S \;{}_u\!\owedge_{v,w}\; (T \;{}_v\!\owedge_w\; U)$

4.5 Atomic statements

Conjunctions involving **abort**, **magic**, **skip** and unconstrained assignment can be eliminated:

$\owedge$-**abort***:* $\quad \mathbf{abort} \;{}_w\!\owedge_u\; S \;\equiv\; \mathbf{abort}, \quad \text{for any } w$

$\owedge$-**magic***:* $\quad \mathbf{magic} \;{}_w\!\owedge_u\; S \;\equiv\; \{\, pre(S) \,\}\,;\mathbf{magic}, \quad \text{for any } w$

$\owedge$-**skip***:* $\quad \mathbf{skip} \;{}_\emptyset\!\owedge_u\; S \;\equiv\; S$

$\owedge$-**choose***:* $\quad u\,\mathbf{:}[\;] \;{}_u\!\owedge_u\; S \;\equiv\; S$

4.6 Simple Specifications

Any specification, $w\!:\![\,P \;/\; R\,]$, can be written as an assumption ($\{P\}$), followed by an unconstrained assignment ($w\!:\![\;]$), followed by a coercion ($[R]$). In the case of conjunction, this gives:

$$w_1 \!:\![\,P_1 \;/\; R_1\,] \owedge w_2 \!:\![\,P_2 \;/\; R_2\,] \quad \equiv \quad \{\,P_1 \wedge P_2\,\}\,; w_1, w_2 \!:\![\;]\,; [\,R_1 \wedge R_2\,]\,,$$

This leads to various special cases, for example, allowing conjunctions of assertions, coercions or unconstrained assignments to be eliminated:

$\owedge$*-assert:* $\quad \{\,P_1\,\}\;{}_{\emptyset}\!\owedge_{\emptyset}\;\{\,P_2\,\} \quad \equiv \quad \{\,P_1 \;\wedge\; P_2\,\}$

$\owedge$*-coerce:* $\quad [R_1]\;{}_{\emptyset}\!\owedge_{\emptyset}\;[R_2] \quad \equiv \quad [R_1 \;\wedge\; R_2]$

$\owedge$*-choice:* $\quad u\!:\![\;]\;{}_{u}\!\owedge_{v}\;v\!:\![\;] \quad \equiv \quad u, v\!:\![\;]$

4.7 Assignment

Conjunctions of assignments can be eliminated; inconsistent assignments to the same variable lead to **magic**:

$\owedge$*-asn-1:* $\quad x := e\;{}_{x}\!\owedge_{y}\;y := f \;\equiv\; x, y := e, f$

$\owedge$*-asn-2:* $\quad x := e\,{}_{x}\!\owedge_{x} x := f \;\equiv\; [e = f]\,x := e \;\equiv\; \begin{cases} x := e, & \text{when } e = f \\ \textbf{magic}, & \text{when } e \neq f \end{cases}$

4.8 Serialisation

A conjunction can be turned into a sequential composition provided this introduces no interference between S and T:

$\owedge$*-ser-1:* $\quad S\;{}_{u}\!\owedge_{v}\;T \;\sqsubseteq\; S\;;\;T \quad \text{if } u \cap sig(T) = \emptyset$

$\owedge$*-ser-2:* $\quad S\;{}_{u}\!\owedge_{v}\;T \;\sqsubseteq\; T\;;\;S \quad \text{if } v \cap sig(S) = \emptyset$

In particular, a conjunction of assignments $x := e\;{}_{x}\!\owedge_{y}\;y := f$ can be turned into, $x := e; y := f$, if x does not occur in f, and into $y := f; x := e$, if y does not occur in e.

These laws can be also be used to turn a conjunction of assertions into a sequence, which can then be collapsed into a single assertions. Similarly for a conjunction of coercions, of unconstrained assignment, or of any combination of these three.

To obtain a more general law, we can think of $S\;{}_{u}\!\owedge_{v}\;T$ as meaning that S and T are executed in parallel, each with its own local state (so that there is no interference), with the resulting states being merged in a way that ensures compatibility of results. Since there is no interference, these versions of S and T can be executed serially.

$\circledwedge$-*ser:* $$S \;{}_u\!\circledwedge_v\; T \;\equiv\; [\![\;\mathbf{var}\ \alpha', \beta' \bullet\; \alpha', \beta' := \alpha, \beta;\; S' \,;\, T' \,;\; (u := u' \;{}_u\!\circledwedge_v\; v := v') \;]\!] \qquad (\textit{or}\ \ T' \,;\, S')$$

where $\alpha = sig(S)$, $\beta = sig(T)$, α' and β' are (sets of) distinct fresh variables corresponding to α and β, $S' = S[\alpha \setminus \alpha']$, $T' = T[\beta \setminus \beta']$, and u' and v' are the subsets of α' and β' corresponding to u and v, respectively.

The final assignment, $u := u' \;{}_u\!\circledwedge_v\; v := v'$, ensures that any demonic nondeterminism in S' and T' is resolved in a consistent manner. We can turn this into a multiple assignment, provided that it is not miraculous; i.e., if every variable occurring in both u and v is assigned the same value by the two statements.

For example, writing $x :\in s$ for $x : [\, x \in s \,]$:

$$x :\in \{0,1\} \;{}_x\!\circledwedge_x\; x :\in \{1,2\} \;\equiv\; [\![\;\mathbf{var}\ x_1, x_2 \bullet\; x_1, x_2 := x, x;\; x_1 :\in \{0,1\}; x_2 :\in \{1,2\};\; (x := x_1 \;{}_x\!\circledwedge_x\; x := x_2) \;]\!]$$

The conjunction $x := x_1 \;{}_x\!\circledwedge_x\; x := x_2$ is only feasible when $x_1 = x_2$, which can only be achieved when $x_1 :\in \{0,1\}$ and $x_2 :\in \{1,2\}$ both select 1 as the value to assign. Thus, the right hand side reduces to $x := 1$.

This law allows any multiple assignment to be turned into a sequence of assignments, possibly involving new variables. The final conjunction of assignments can be turned into a sequence, since it assigns one set of variables to another, where the variables being assigned are all distinct (since they were the left hand side of a multiple assignment) and are disjoint from the variables on the right hand sides (which were fresh variables). For example, we can turn $x := y \;{}_x\!\circledwedge_y\; y := x$ into a sequence of assignments using two additional variables, which can be simplified to obtain the familiar version with one additional variable.

4.9 Distribution

The most important properties for our purposes are ones allowing conjunction to be distributed over other program constructors. This allows the logic which addresses the new requirement to be integrated with that of the original implementation.

We consider how to refine a specification of the form $S \;{}_u\!\circledwedge_v\; T$, where S is formed using various program constructors, and T is an arbitrary (conjunctive) program or is formed using the same program constructor as S.

Sequence We can distribute T over a sequence, $S_1 ; S_2$, by conjoining it with either S_1 or S_2, provided this introduces no interference.

$\owedge$*-dist-seq-1:* $(S_1 \;;\; S_2)\ {}_u{\owedge}_v\ T \;\sqsubseteq\; (S_1\ {}_u{\owedge}_v\ T) \;;\; S_2, \quad \text{if } u \cap v = \emptyset$

$\owedge$*-dist-seq-2:* $(S_1 \;;\; S_2)\ {}_u{\owedge}_v\ T \;\sqsubseteq\; S_1 \;;\; (S_2\ {}_u{\owedge}_v\ T), \quad \text{if } u \cap v = \emptyset$

Note that S_1 and S_2 have the same frame (u), which may require using the *Expand Frame* law. This ensures that any variable assigned by S_1, but not S_2 (or vice versa), must be preserved by the conjunction.

When T is also a sequence, we can distribute the sequence over both sides, synchronising at their mid-points, provided this does not introduce interference. Of course, this requires an intelligent choice in splitting T; a poor choice will lead to infeasibility.

$\owedge$*-dist-seq-3:* $(S_1 \;;\; S_2)\ {}_u{\owedge}_v\ (T_1 \;;\; T_2) \;\sqsubseteq\; (S_1\ {}_u{\owedge}_v\ T_1) \;;\; (S_2\ {}_u{\owedge}_v\ T_2)$
if $u \cap v = \emptyset$

Alternation We can distribute T over an alternation by conjoining T with each branch of the alternation:

$\owedge$*-dist-alt-1:* $\textbf{if } \big[\!\big]_i\ G_i \rightarrow S_i \textbf{ fi}\ {}_u{\owedge}_v\ T \;\sqsubseteq\; \textbf{if } \big[\!\big]_i\ G_i \rightarrow S_i\ {}_u{\owedge}_v\ T \textbf{ fi}$

Note that all the S_i's have the same frame (u), so any variable assigned in one branch, but not in another, must be preserved by the conjunction.

We get a special case when T is also an alternation:

$\owedge$*-dist-alt-2:* $\textbf{if } \big[\!\big]_i\ G_i \rightarrow S_i \textbf{ fi}\ \ {}_u{\owedge}_v\ \ \textbf{if } \big[\!\big]_j\ H_j \rightarrow T_j \textbf{ fi}$
$\sqsubseteq\; \textbf{if } \big[\!\big]_{i,j}\ G_i \wedge H_j \rightarrow S_i\ {}_u{\owedge}_v\ T_j \textbf{ fi}$

which can be further specialised when the sets of guards are identical:

$\owedge$*-dist-alt-3:* $\textbf{if } \big[\!\big]_i\ G_i \rightarrow S_i \textbf{ fi}\ \ {}_u{\owedge}_v\ \ \textbf{if } \big[\!\big]_i\ G_i \rightarrow T_i \textbf{ fi}$
$\sqsubseteq\; \textbf{if } \big[\!\big]_i\ G_i \rightarrow S_i\ {}_u{\owedge}_v\ T_i \textbf{ fi}$

Local variables We can distribute T over a variable declaration:

$\owedge$*-dist-var-1:* $[\![\ \textbf{var } x \bullet S'\ ]\!]\ {}_u{\owedge}_v\ T \;\sqsubseteq\; [\![\ \textbf{var } x \bullet S'\ {}_{u \cup x}{\owedge}_{v \cup x}\ T'\ ]\!]$

where $x \notin v$ and $T' = x \mathbin{:}[\ ] ; T$.

When T is also a variable declaration, we can combine the declarations:

$\owedge$*-dist-var-2:* $[\![\ \textbf{var } x \bullet S'\ ]\!]\,{}_u{\owedge}_v\,[\![\ \textbf{var } y \bullet T'\ ]\!] \sqsubseteq [\![\ \textbf{var } x, y \bullet S'\,{}_u{\owedge}_v T'\ ]\!]$,
if $x \notin sig(T)$ and $y \notin sig(S)$

Repetition Various laws can be found for distributing conjunction over repetitions. Here we just give a special case, where S and T are both one-branch loops with the same guard:

$\owedge$*-dist-rep:* $\quad \mathbf{do}\ G \rightarrow S'\ \mathbf{od}\ {}_u\!\owedge_v\ \mathbf{do}\ G \rightarrow T'\ \mathbf{od}\ \sqsubseteq\ \mathbf{do}\ G \rightarrow S'\ {}_u\!\owedge_v\ T'\ \mathbf{od}$

This can easily be extended, using $\owedge$*-dist-alt-3*, to obtain a version where the two loops have multiple branches with identical sets of guards.

5 Propagating Modifications through a Derivation

We will now show how the properties presented in Section 4 can be used to propagate a specification change through an existing derivation and identify places where changes are required. We begin by outlining the general idea, then illustrate the approach using a simple example. The example is chosen to be simple enough to be able to present within a few pages, whilst also illustrating enough of the techniques for propagating new requirements to be interesting.

5.1 General Approach

Suppose we have a derivation of a program P from a specification S, and we wish to construct a derivation for an augmented specification $S \owedge T$, where T expresses some new requirements. Let the given derivation be $S_0 \sqsubseteq_{r_1} S_1 \sqsubseteq_{r_2} \cdots \sqsubseteq_{r_n} S_n$, where $S_0 = S$, $S_n = P$, and for $i = 1 \mathinner{..} n$, r_i is a rule which justifies $S_{i-1} \sqsubseteq S_i$. We will attempt to construct a derivation for an augmented specification $S \owedge T$, by adapting steps in the given derivation.

At each step in the new derivation, we already have a rule r_i which allows us to refine S_i. What we need to do is determine whether this rule can be used (perhaps in modified form) to also refine the corresponding specification in the new derivation, and if so how.

In some cases, P may already implement T, so no change will be required. Or we may be able to implement T separately from S, and combine their implementations; e.g. using sequential composition. In this case, the derivation of S will be unchanged, and we will end up with P as a component of the new program. This approach, however, may not lead to a satisfactory implementation. For example, we know that we can compute both the minimum and the maximum of a sequence using a single loop, whereas a compositional approach would result in two loops. That could be far less efficient if the elements of A are computed on the fly, and completely unsatisfactory if elements of A are read from an non-repeatable source (e.g. user input).

In some situations, we will not be able to (or, for some other reason, not wish to) use some or all of the steps of the original derivation. In the worst case, we may end up abandoning the original derivation entirely, and deriving a new program for $S \owedge T$ from scratch. We are primarily concerned with finding ways of reusing the existing derivation when it is both possible and desirable to do so, so we will assume that there is something in the existing derivation worth saving. Recognising when it is possible and/or desirable to reuse the existing derivation remains an important strategic decision.

In the following example, we begin with a derivation of an algorithm to find the minimum in a sequence of numbers, then extend it to also find the maximum. Since the initial derivation should be quite familiar, and is not of much interest in its own right, we present it with little comment. Proof obligations are also omitted.

5.2 Initial derivation

The following is a derivation of an algorithm to find the minimum element (lo) in a sequence A of length n (where $n \geq 1$). We write $A[\,..j]$ for the segment of A beginning at index 1 and ending at index j.

$$lo\,\mathbf{:}\big[\, n \geq 1 \;/\; lo = \sqcap A[\,..n] \,\big] \qquad (S_1)$$

(S_1) $\sqsubseteq$ (*Introduce Local Variable*)
var k $\bullet$
$lo, k\,\mathbf{:}\big[\, n \geq 1 \;/\; lo = \sqcap A[\,..n] \,\big]$ $\qquad (S_2)$

(S_2) $\sqsubseteq$ (*Split Specification*)
$lo, k\,\mathbf{:}\big[\, n \geq 1 \;/\; 1 \leq k \leq n \wedge lo = \sqcap A[\,..k] \,\big]\,;$ $\qquad (S_3)$
$lo, k\,\mathbf{:}\big[\, 1 \leq k \leq n \wedge lo = \sqcap A[\,..k] \;/\; lo = \sqcap A[\,..n] \,\big]$ $\qquad (S_4)$

(S_3) $\sqsubseteq$ (*Introduce Assignment*)
$lo, k := A[1], 1$

(S_4) $\sqsubseteq$ (*Introduce DO*, inv: $1 \leq k \leq n \wedge lo = \sqcap A[\,..k]$, var: $n - k$)
do $k \neq n \rightarrow$

$$lo, k\,\mathbf{:}\left[\begin{array}{c} 1 \leq k < n \\ lo = \sqcap A[\,..k] \end{array} \middle/ \begin{array}{c} 1 \leq k \leq n \\ lo = \sqcap A[\,..k] \\ k_0 < k \end{array}\right] \qquad (S_5)$$

od

(S_5) $\sqsubseteq$ (*Following Assignment*, *Contract Frame*)
$lo\,\mathbf{:}\big[\, 1 \leq k < n \wedge lo = \sqcap A[\,..k] \;/\; lo = \sqcap A[\,..k{+}1] \,\big]\,;$ $\qquad (S_6)$
$k := k{+}1$

$$
\begin{array}{lll}
(S_6) & \sqsubseteq & (\mathit{Introduce\ IF}) \\
& & \mathbf{if}\ \ lo \le A[k+1] \rightarrow \\
& & \qquad lo \mathbin{:} \left[\begin{array}{l} 1 \le k < n \\ lo = \sqcap A[\,..\,k] \\ lo \le A[k+1] \end{array} \Big/ \ lo = \sqcap A[\,..\,k+1] \right] \qquad (S_7) \\
& & [\!]\ \ lo \ge A[k+1] \rightarrow \\
& & \qquad lo \mathbin{:} \left[\begin{array}{l} 1 \le k < n \\ lo = \sqcap A[\,..\,k] \\ lo \ge A[k+1] \end{array} \Big/ \ lo = \sqcap A[\,..\,k+1] \right] \qquad (S_8) \\
& & \mathbf{fi}
\end{array}
$$

$$
\begin{array}{lll}
(S_7) & \sqsubseteq & (\mathit{Introduce\ Skip}) \\
& & \mathbf{skip}
\end{array}
$$

$$
\begin{array}{lll}
(S_8) & \sqsubseteq & (\mathit{Introduce\ Assignment}) \\
& & lo := A[k+1]
\end{array}
$$

The resulting program is:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ k \bullet \\
\quad lo, k := A[1], 1; \\
\quad \mathbf{do}\ k \ne n \rightarrow \\
\qquad \mathbf{if}\ \ lo \le A[k+1] \rightarrow \mathbf{skip} \\
\qquad [\!]\ \ lo \ge A[k+1] \rightarrow lo := A[k+1] \\
\qquad \mathbf{fi}; \\
\qquad k := k+1 \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

5.3 Extending the derivation

We will now extend this derivation to also find the maximum (hi) of $A[\,..\,n]$. We express the new requirement as a specification statement, and conjoin it with the original specification.

$$lo \mathbin{:} [\, n \ge 1 \ / \ lo = \sqcap A[\,..\,n] \,] \ {}_{lo}\circledast_{hi}\ hi \mathbin{:} [\, n \ge 1 \ / \ hi = \sqcup A[\,..\,n] \,] \qquad (T_1)$$

We could immediately refine this to a sequential composition, which would lead to an implementation with two loops. For the purposes of the example, we consider the alternative of integrating the new requirement with the existing program, by performing similar refinements on the new requirement and then distributing the conjunction. In presenting the derivation, we omit rule names and sometimes combine refinement steps, focusing on the decisions made in effecting the required modification.

The first step of the given derivation introduces a local variable to use as a loop counter. We could could now refine the new specification in the same way, introducing another variable, and distribute the conjunction over the two declarations using *Ⓐ-dist-var-2*. Alternatively, we could use *Ⓐ-dist-var-1* to distribute the conjunction over the declaration in the original derivation, making its variable available to the new specification. Anticipating a similar form of loop for computing hi, we take the latter option:[1]

$$
\begin{array}{lll}
(T_1) & \sqsubseteq & \mathbf{var}\ k \bullet \\
& & lo, k\,\mathbf{:}\big[\,n \geq 1 \ \big/\ lo = \sqcap A[\,..\,n]\,\big]\ {}_{lo,k}Ⓐ_{hi,k} \\
& & hi, k\,\mathbf{:}\big[\,n \geq 1 \ \big/\ hi = \sqcup A[\,..\,n]\,\big] \qquad\qquad (T_2)
\end{array}
$$

The next step in the given derivation was to split S_2, introducing the loop invariant as the intermediate assertion. We need to refine the specification for hi in a similar way, so we apply the same step to the right-hand side of T_2, using $1 \leq k \leq n \wedge hi = \sqcup A[\,..\,k]$ as the intermediate assertion (and invariant), then distribute the conjunction over the pair of sequential compositions (using *Ⓐ-dist-seq-3*):

$$
\begin{array}{lll}
(T_2) & \sqsubseteq & \left(lo, k\,\mathbf{:}\left[n \geq 1 \ \middle/\ \begin{array}{c} 1 \leq k \leq n \\ lo = \sqcap A[\,..\,k] \end{array} \right] {}_{lo,k}Ⓐ_{hi,k} \right. \\
& & \quad \left. hi, k\,\mathbf{:}\left[n \geq 1 \ \middle/\ \begin{array}{c} 1 \leq k \leq n \\ hi = \sqcup A[\,..\,k] \end{array} \right] \right) ; \qquad (T_3) \\
& & \left(lo, k\,\mathbf{:}\left[\begin{array}{c} 1 \leq k \leq n \\ lo = \sqcap A[\,..\,k] \end{array} \ \middle/\ lo = \sqcap A[\,..\,n] \right] {}_{lo,k}Ⓐ_{hi,k} \right. \\
& & \quad \left. hi, k\,\mathbf{:}\left[\begin{array}{c} 1 \leq k \leq n \\ hi = \sqcup A[\,..\,k] \end{array} \ \middle/\ hi = \sqcup A[\,..\,n] \right] \right) \qquad (T_4)
\end{array}
$$

In the given derivation, the left hand side of T_3 was refined to an assignment. We can refine the right hand side of T_3 in the same way, and combine the two assignments to get one multiple assignment:

$$
\begin{array}{lll}
(T_3) & \sqsubseteq & lo, k := A[1], 1\ {}_{lo,k}Ⓐ_{hi,k}\ hi, k := A[1], 1 \\
& \sqsubseteq & lo, hi, k := A[1], A[1], 1
\end{array}
$$

The last step has $1 = 1$ as a proof obligation, because k is assigned by both sides of the conjunction.

We now introduce a loop on the right-hand side of T_4, just as we did on the left-hand side in the original derivation. We have already chosen the loop invariant to be similar to that used on the left-hand side; we also choose the

[1] It takes some foresight to realise that we should use the same local variable in computing hi. To lessen the dependence on foresight, we could instead use a different variable and unite the two variables later as a simplification.

same variant function $(n - k)$. We then distribute the conjunction over the two loops, giving:

$$
\begin{array}{lll}
(T_4) & \sqsubseteq & \mathbf{do}\ k \neq n \rightarrow \\
& & \qquad lo, k \mathbin{\boldsymbol{:}} \left[\begin{array}{c} 1 \leq k < n \\ lo = \sqcap A[\,..\,k] \end{array} \middle/ \begin{array}{c} 1 \leq k \leq n \\ lo = \sqcap A[\,..\,k] \\ k_0 < k \end{array} \right] \\
& & \qquad {}_{lo,k}\owedge_{hi,k} \\
& & \qquad hi, k \mathbin{\boldsymbol{:}} \left[\begin{array}{c} 1 \leq k < n \\ hi = \sqcup A[\,..\,k] \end{array} \middle/ \begin{array}{c} 1 \leq k \leq n \\ hi = \sqcup A[\,..\,k] \\ k_0 < k \end{array} \right] \\
& & \mathbf{od}
\end{array}
\qquad (T_5)
$$

We can now perform *Following Assignment* and *Contract Frame* on both sides of the loop body, and distribute the conjunction over the two sequences:

$$
\begin{array}{lll}
(T_5) & \sqsubseteq & (\ lo \mathbin{\boldsymbol{:}} [\, 1 \leq k < n \wedge lo = \sqcap A[\,..\,k] \ /\ lo = \sqcap A[\,..\,k{+}1] \,]\ {}_{lo,k}\owedge_{hi,k} \\
& & \quad hi \mathbin{\boldsymbol{:}} [\, 1 \leq k < n \wedge hi = \sqcup A[\,..\,k] \ /\ hi = \sqcup A[\,..\,k{+}1] \,]\ )\ ; \\
& & k := k{+}1
\end{array}
\qquad (T_6)
$$

At this point in the original derivation, the left hand side of T_6 was refined to an alternation, and the branches refined to a **skip** and an assignment. We now refine the right hand side in a similar way:

$$
\begin{array}{lll}
(T_6) & \sqsubseteq & \mathbf{if}\ \ lo \leq A[k{+}1] \rightarrow \mathbf{skip} \\
& & [\!]\ \ lo \geq A[k{+}1] \rightarrow lo := A[k{+}1] \\
& & \mathbf{fi} \\
& & \qquad {}_{lo,k}\owedge_{hi,k} \\
& & \mathbf{if}\ \ hi \geq A[k{+}1] \rightarrow \mathbf{skip} \\
& & [\!]\ \ hi \leq A[k{+}1] \rightarrow hi := A[k{+}1] \\
& & \mathbf{fi}
\end{array}
$$

We could now turn the $\owedge$ into a sequence, giving a loop body containing two **if** statements. We choose instead to distributing the conjunction to obtain a single alternation. Because the two **if** statements have different guards, we use $\owedge$*-dist-alt-2*, giving:

$$
\begin{array}{l}
\mathbf{if}\ \ lo \leq A[k{+}1]\ \wedge\ hi \geq A[k{+}1] \rightarrow \mathbf{skip}\ {}_{lo,k}\owedge_{hi,k}\ \mathbf{skip} \\
[\!]\ \ lo \leq A[k{+}1]\ \wedge\ hi \leq A[k{+}1] \rightarrow \mathbf{skip}\ {}_{lo,k}\owedge_{hi,k}\ hi := A[k{+}1] \\
[\!]\ \ lo \geq A[k{+}1]\ \wedge\ hi \geq A[k{+}1] \rightarrow lo := A[k{+}1]\ {}_{lo,k}\owedge_{hi,k}\ \mathbf{skip} \\
[\!]\ \ lo \geq A[k{+}1]\ \wedge\ hi \leq A[k{+}1] \rightarrow lo := A[k{+}1]\ {}_{lo,k}\owedge_{hi,k}\ hi := A[k{+}1] \\
\mathbf{fi}
\end{array}
$$

Note that the branches had to be conjoined using the same frames as the **if** statements (i.e. lo, k and hi, k). Thus, for example, the implementation of $\textbf{skip}\ _{lo,k}\owedge_{hi,k}\ \textbf{skip}$ is required to keep lo, hi and k unchanged.

We can now eliminate the $\owedge$'s from the first three branches (using $\owedge$-**skip**), and turn the fourth branch into a multiple assignment (using $\owedge$-*asn-1*).

5.4 Further simplification

We might now observe that the last branch in the **if** statement is redundant, because $lo \le hi$ is invariant throughout the loop, so its guard is then equivalent to $lo = A[k{+}1] \ \wedge\ hi = A[k{+}1]$, which is covered by the other branches.

To show that $lo \le hi$ is indeed invariant throughout the loop, we could add it as a conjunct in both loop invariants and redo the derivation of the loop. By using conjunction again, we can avoid redoing steps we have already done, and concentrate on just the steps required to show that $lo \le hi$ is invariant.

We add the coercion $[\, lo \le hi \,]$ as an additional conjunct in T_2 (this is a form of *Strengthen Postcondition*).

We split this coercion into $[\, lo \le hi \,]\,;\ \mathbf{:}[\, lo \le hi \ / \ lo \le hi \,]$, and distribute over the sequence introduced by refining T_2. This means that T_3 has to establish $lo \le hi$, and T_4 has to preserve $lo \le hi$.

The assignment introduced in refining T_3 sets lo and hi to 1, so the $[\, lo \le hi \,]$ conjunct on T_3 can be eliminated.

The conjunct $\mathbf{:}[\, lo \le hi \ / \ lo \le hi \,]$ on T_4 is propagated through the refinement of T_4 and T_5 in a similar way to the specification for hi.

The refinement of T_6 can proceed as before, but we defer refining the branches of the **if** statements. We thus get the four branch **if** statement conjoined with $\mathbf{:}[\, lo \le hi \ / \ lo \le hi \,]$. We can now use the precondition of $\mathbf{:}[\, lo \le hi \ / \ lo \le hi \,]$ to simplify the fourth guard to $lo = hi$, showing that this branch can be eliminated.

We now distribute this conjunction, conjoining it to each branch of the **if** statement. It is then easy to show that the **skip** and assignment statements introduced in refining the branches of the **if** statement all preserve the $lo \le hi$, and these conjuncts can be eliminated.

We have now succeeded in showing that the loop preserves $lo \le hi$, and simplifying the **if** statement to eliminate the redundant branch.

5.5 Summary of modifications

This completes the modification required to implement the augmented specification. The resulting program is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ k \bullet \\
\quad lo, hi, k := A[1], A[1], 1; \\
\quad \textbf{do}\ k \neq n \rightarrow \\
\qquad \textbf{if}\ \ lo \leq A[k{+}1]\ \wedge\ hi \geq A[k{+}1] \rightarrow \textbf{skip} \\
\qquad [\!]\ \ lo \leq A[k{+}1]\ \wedge\ hi \leq A[k{+}1] \rightarrow hi := A[k{+}1] \\
\qquad [\!]\ \ lo \geq A[k{+}1]\ \wedge\ hi \geq A[k{+}1] \rightarrow lo := A[k{+}1] \\
\qquad \textbf{fi}; \\
\qquad k := k{+}1 \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

Our first modification (Section 5.3) involved adding a new output, which was the first kind of change discussed in Section 2. The subsequent simplification (Section 5.4) involved constraining both the precondition and postcondition of the loop body, which were the second and third kinds of change discussed in Section 2. In this case, we were able to show that the additional constraints required no change to the program, but did allow a small simplification.

We were able to reuse most of the original derivation, and did not need to reconsider any of the original proof obligations. Considering how to propagate the new requirement at each step suggested several possible implementation choices, so we can see exactly what choices lead to the final version of the program.

6 Related work

Several authors have investigated conjunction-like program operators, using different semantic frameworks and with different aims. These operators often differ in their treatment of non-termination and/or infeasibility; for example, (i) requiring the conjoined postcondition to be established only when both preconditions hold *and* when both postconditions are feasible (e.g. [18]), or (ii) requiring each postcondition to be established whenever the corresponding precondition holds (e.g. [17], [2], [11]). Others do not identify the variables being updated (e.g. Back and Butler's *fusion* operator [4], [5], which is otherwise identical to our conjunction operator).

In the context of relational [10] or predicative [9] semantics, program conjunction corresponds to set intersection and logical conjunction, respectively, and is a natural dual of demonic choice. With these semantics, program conjunction is very similar to Z's schema conjunction. This is attractive, from the point of view of composing specifications, but is a disadvantage in program development, because the operator is not monotonic with respect to refinement. This arises because these semantics typically do not distinguish between non-termination and infeasibility, or between preconditions and guards. In weakest precondition semantics, the dual of demonic choice is angelic choice, which has many of the properties of conjunction, but is not conjunctive. In this setting, a conjunction-like operator can be obtained by taking the least conjunctive refinement of the angelic choice of two programs ([17], [15], [13]) — this gives the variants described under (ii) above.

Some authors have been concerned with using conjunction for structuring specifications, and thus assume that conjunctions are eliminated before program development begins; for example, Ainsworth [2] uses several conjunction-like operators to describe ways of combining partial specifications (viewpoints) into a single specification. Others have been concerned with deriving parallel programs, so retain conjunctions in the final program. There appears to have been little work on using conjunction in the development process in the way illustrated here.

Hehner [9] describes a technique, which he calls "Programming by Parts", for constructing programs from specifications containing conjunctions, but does not explore the technique in any detail. This technique is developed further, in the context of relational semantics, by Frappier *et al* [6]. While this work appears to be similar in spirit to that presented here, the details are quite different, concentrating on the algebraic form of relational specifications.

7 Conclusions

We have shown that evolutionary modifications to specifications can often be expressed using a form of program conjunction, and have identified a suitable operator (sometimes called miraculous conjunction) for expressing a range of typical modifications. The novel contribution of the paper is to show that properties of this operator can be used to propagate a modification through an existing derivation to identify places where changes are required.

Seeing program conjunction used in this way suggests other ways in which conjunction can be used in program derivation. We might use these techniques to combine complete programs or to merge different extensions of a program (*cf* [3]). We might also use these techniques earlier in the development process by decomposing a specification into a conjunction of specifications, for (some of) which we already have derivations. For example, we might decompose a sort specification to $s : [\, asc(s) \,] \circledwedge s : [\, perm(s, s_0) \,]$, then consider how an algorithm to construct an ascending sequence can be modified so that the sequence constructed is a permutation of a given one, or how to modify an algorithm to construct a permutation of a given sequence so that the resulting sequence is ascending.

Even in a simple loop, such as the one to find the minimum of a sequence, we can usefully view the loop body (S_5) as a conjunction of two specifications, one ensuring that the loop terminates and one ensuring that the required postcondition will hold on termination:

$$k : \left[1 \le k < n \;\middle/\; \begin{array}{c} 1 \le k \le n \\ k_0 < k \end{array} \right] \circledwedge lo : [\, lo = \sqcap A[\,..\,k] \;/\; lo = \sqcap A[\,..\,k] \,]$$

Again, we can consider a variety of techniques for implementing either of these specifications, and how to adapt them to also implement the other. This allows a greater separation of concerns, leading more easily to an implementation where the loop counter may be updated within the code that preserves the loop invariant (as happens, for example, in a binary search).

We are investigating further properties of program conjunction, to allow more general combinations of programs and modifications. In particular, we are investigating more general laws for combining loops. We are also investigating other operators, such as sequence and disjunction, which can be used to effect a wider range of changes. Morgan [14] gives a number of rules for distributing various programming constructs over sequences and alternations, which can be used in this way, though they do not appear to have been used like this.

Finally, we are investigating ways of supporting modification and combination of derivations in a refinement tool. In particular, we are attempting to extend the Refinement Structure Diagram techniques described in [8] to visualise derivations involving propagating operators such as conjunctions, to allow a user to understand more easily to overall state of a derivation.

Acknowledgements

Thanks to Ray Nickson, Peter Andreae and the anonymous referees for their comments and corrections on earlier drafts of this paper. Thanks also to Brendan Mahony for access to [13] and for answering many questions about the details therein.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Mike Ainsworth and Peter Wallis. "Co-Refinement". *Proc. 6th Refinement Workshop*, David Till (Ed), Springer-Verlag, 1994, pp151–166.
3. Valdis Berzins. "On Merging Software Extensions". *Acta Informatica* **23** (1986), pp607–619.
4. R.J.R Back and M.J. Butler. "Exploring Summation and Product Operators in the Refinement Calculus". *Mathematics of Program Construction*, B. Möller (Ed), LNCS 947, Springer-Verlag, 1995.
5. R.J.R Back and M.J. Butler. "Fusion and Simultaneous Execution in the Refinement Calculus". Technical Reports, DSSE-TR-96-1, University of Southampton.
6. M. Frappier, A. Mili and J. Desharnais. "A Relational Calculus for Program Construction by Parts". *Science of Computer Programming* **26** (1996), pp237–254.
7. Lindsay Groves. "Deriving Programs by Combining and Adapting Refinement Scripts". Proc. APSEC'95, IEEE Computer Society Press, 1995, pp354–363. Technical Report TR-CS-95/13, Victoria University of Wellington.
8. Lindsay Groves and Ivan Tuckwell. "An Editor for Refinement Structure Diagrams." *Formal Methods Pacific '97, Work in Progress Papers*, Lindsay Groves and Steve Reeves (Eds), Victoria University of Wellington, 1997.
9. Eric C. R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993

10. C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey and B. A. Sufrin "Laws of Programming". *Communications of the ACM*, **30** (1987), pp672–686.
11. K. Rustan M. Leino and Rajit Manohar. "Joining specification statements". Digital Systems Research Center, TCS-1, 1996.
(See http://www.research.digital.com/SRC/personal/Rustan_Leino/home.html)
12. Brendan Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, University of Queensland, 1991.
(See ftp://ftp.it.uq.edu.au/pub/Thesis/brendan_mahony.ps.Z)
13. Brendan Mahony. "The Least Conjunctive Refinement and Promotion in the Refinement Calculus". Draft paper, 1998.
14. Carroll Morgan. *Programming from Specifications.* Prentice-Hall, Second Edition, 1994.
15. Carroll Morgan. "The Cuppest Capjunctive Capping". *A Classical Mind*, A. W. Roscoe (Ed), Prentice Hall, 1994, pp317–332.
16. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Second edition, 1992.
17. Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, 1992.
18. Nigel Ward. "Adding specification constructors to the refinement calculus". *FME'93: Industrial-Strength Formal Methods*, J C P Woodcock and P G Larsen (Eds), Springer-Verlag, 1993, pp652–670.
19. Mark Weiser. "Program slicing". *IEEE Transactions on Software Engineering*, **10** (1984), pp352–357.