Modifying Program Derivations using Program Disjunction

Lindsay Groves

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

Abstract

This paper considers how an existing program can be modified to implement a specification with a larger domain of termination. We introduce a program disjunction operator which can be used to describe this kind of modification, and show how properties of this operator can be used to combine two programs, or to adapt an existing program derivation, to obtain an implementation for the modified specification.

1 Introduction

Suppose we have a program U which implements a specification S. Using the notation of the refinement calculus [Mor94], we might express S as a specification statement, $w\!:\!\left[P \;\middle/\; R\right]$, and write $S \sqsubseteq U$ to mean that U implements S. This means that for inputs satisfying the precondition P, U must establish the postcondition R, modifying only variables in the frame w. For inputs not satisfying P, U can do anything, including not terminate.

Now, suppose we wish to modify our program so that it behaves in a specified way for a wider range of inputs. This kind of modification occurs frequently in the evolutionary development of software. For example, it might arise as part of a maintenance task, where a program is required to handle a wider range of inputs; as part of a reuse process, where a program is reused in a different context, where a wider range of inputs must be handled; or as part of an incremental development process, where we begin by implementing a program that handles some common input cases and then extend it to handle other cases, or perhaps develop programs for different cases separately and then combine them. In constructing the new program, we would like to modify only those parts of U that need to be changed, and retain the parts that are still useful — unless we choose to make more major changes in the interests of, say, efficiency or simplicity.

If the additional inputs are those satisfying precondition P' and the required behaviour for these inputs is described by the postcondition R', we can write the new specification as $w\!:\!\left[P \vee P' \;\middle/\; (P \Rightarrow R) \wedge (P' \Rightarrow R')\right]$, assuming for now that P and P' are disjoint, that R' can be established by modifying only variables in w, and that no variable in w occurs in P or P'. Alternatively, we might describe the new requirements using a separate specification statement, $w\!:\!\left[P' \;\middle/\; R'\right]$, and combine this with the original specification using a program disjunction operator, similar to the schema disjunction operator in Z [Spi92].

In this paper, we show how such a program disjunction operator can be used within the refinement calculus (see [Mor94, BvW98]) to make the kind of modification described above. We first discuss how to define a suitable operator (Section 2), and outline some possible approaches to implementing a specification described as a disjunction (Section 3). We then present some properties of program disjunction (Section 4), and show how these properties can be used to combine two programs or modify an existing derivation to implement a specification expressed as a disjunction (Sections 5 and 6). Finally, we draw our conclusions and discuss some related work (Section 7).

2 Defining Program Disjunction

We want to define a program disjunction operator, $\curlyvee$, that can be applied to arbitrary programs, not just specifications. The disjunction $S \curlyvee T$ should specify a program that behaves like S for inputs in the domain (satisfying the precondition) of S, and like T for inputs in the domain of T. As usual, we don't care what happens for inputs that are not in the domain of either S or T.

Rather than requiring the domains of S and T to be disjoint, we obtain a simpler operator by allowing $S \curlyvee T$ to behave nondeterministically like either S or T for inputs in the domains of both S and T. This is also consistent with schema disjunction in Z.

Finally, we will require that when S is chosen, $S \curlyvee T$ should only modify variables that S can modify, and when T is chosen, $S \curlyvee T$ should only modify variables that T can modify. This is unlike schema disjunction in Z, where the disjunction of two schemas can modify any variables that either of the schemas can modify, but is more consistent with the usual treatment of frames in the refinement calculus and is more convenient for program development.

These considerations suggest that $S \curlyvee T$ can be defined in terms of an **if** statement which chooses between S and T, based on their preconditions. In order to express this for arbitrary programs, we introduce an operator which gives the precondition for any program (i.e. the weakest precondition under which it is guaranteed to terminate).

Definition 1 $\mathsf{pre}\ S \mathrel{\widehat=} wp\,.\,S\,.\,true$

We can now define program disjunction as follows.

Definition 2 $S \curlyvee T \mathrel{\widehat=} \textbf{if}\ \mathsf{pre}\ S \rightarrow S \mathbin{[\!]} \mathsf{pre}\ T \rightarrow T\ \textbf{fi}$

Program disjunction can be used for structuring specifications, in essentially the same way that schema disjunction is used in Z. More importantly, for our purposes, it can be used for describing the kinds of modification outlined in Section 1. For example, we might begin with a program that finds a k between 1 and n (for some constant n) satisfying some condition p, assuming that at least one such value exists, with the specification:

$$k\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; k \in 1\,..\,n \wedge p(k)\right]$$

We might then want to extend this program so that it will also work when there is no such value, in which case it should return a value of k that is not within $1\,..\,n$. The additional requirement can be specified as:

$$k\colon\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n\right]$$

Using program disjunction, we can combine these to describe the desired program:

$$k\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; \begin{array}{c} k \in 1\,..\,n \\ p(k) \end{array}\right] \curlyvee k\colon\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n\right]$$

Using the definitions of specification and **if** statements [Mor94], we can see that this is equivalent to:

$$k \mathbin{:} \left[true \;\Big/\; \begin{array}{c} (\exists\, i \in 1 \mathinner{..} n \bullet p(i)) \Rightarrow k \in 1 \mathinner{..} n \wedge p(k) \\ (\forall\, i \in 1 \mathinner{..} n \bullet \neg\, p(i)) \Rightarrow k \notin 1 \mathinner{..} n \end{array} \right]$$

which is the way we would otherwise write this specification. The form using disjunction is easier to understand because it more clearly separates the two cases being considered. It is also better as a basis for modifying an existing implementation for the first disjunct, because it makes it easier to see what comes from the original program and what is the required change.

3 Implementing Program Disjunction I

The question we must now address is: How we can obtain an implementation for a specification expressed as a disjunction, in the case where we already have an implementation for one of the component specifications?

One option is simply to expand the disjunction, as illustrated above, and derive a new program from the resulting specification. This will always be an option we have to consider, since the existing implementation may not be useful, but it will normally be a last resort, as it does not allow us to take advantage of the existing implementation.

Another option is to implement the disjunction as an **if** statement, as suggested by Definition 2. As a very simple example, suppose we have a program that finds the absolute difference between x and y, assuming that x is less than or equal to y:

$$z \mathbin{:} \left[x \leq y \;\Big/\; z = \mid x - y \mid \right] \sqsubseteq z := y - x$$

We can extend this so that it also works when x is greater than y:

$$z \mathbin{:} \left[x \leq y \;\Big/\; z = \mid x - y \mid \right] \curlyvee z \mathbin{:} \left[x > y \;\Big/\; z = \mid x - y \mid \right]$$

$\sqsubseteq$ $\langle$ Defn $\curlyvee$. Reuse above derivation. $\rangle$

if $x \leq y \rightarrow z := y - x$

$[\!]$ $x > 0 \rightarrow z \mathbin{:} \left[x > y \;\Big/\; z = \mid x - y \mid \right]$

fi

$\sqsubseteq$ $\langle\, z \mathbin{:} \left[x > y \;\Big/\; z = \mid x - y \mid \right] \sqsubseteq z := x - y \,\rangle$

if $x \leq y \rightarrow z := y - x$

$[\!]$ $x > 0 \rightarrow z := x - y$

fi

In this case, it the derivation of $z := y - x$ from $z \mathbin{:} \left[x \leq y \;\Big/\; z = \mid x - y \mid \right]$ is simple enough that it is no burden to repeat it. But in a larger program, reusing the results of a previous derivation could bring a significant saving.

This approach is not always desirable. For example, there may be considerable overlap between the computations required for the two branches. In this case, it might be possible, and more desirable, to adapt the effect of the existing program to the new inputs by performing some form of pre- or post-processing. Thus, in the above example (although it is not really complex enough to warrant it), we could obtain either of the following implementations:

$$
\begin{array}{l}
\textbf{if} \quad x \leq y \rightarrow \textbf{skip} \\
[\!] \quad x > 0 \rightarrow x, y := y, x \\
\textbf{fi} \ ; \\
z := y - x
\end{array}
\qquad\qquad
\begin{array}{l}
z := y - x \ ; \\
\textbf{if} \quad z \geq 0 \rightarrow \textbf{skip} \\
[\!] \quad z < 0 \rightarrow z := -z \\
\textbf{fi}
\end{array}
$$

In other cases, however, neither of these approaches will give a satisfactory result, because it is not possible to test the preconditions directly. For example, suppose we have a program that finds a k between 1 and n (for some constant n) satisfying some condition p, assuming that at least one such value exists:

$$
k\colon\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\Big/\; \begin{array}{c} k \in 1\,..\,n \\ p(k) \end{array}\right] \;\sqsubseteq\; \begin{array}{l} k := 1 \ ; \\ \textbf{do} \ \neg\, p(k) \rightarrow \\ \quad k := k + 1 \\ \textbf{od} \end{array}
$$

And suppose we want to extend the program so it will also work when there is no such values, in which case it should return a value of k that is not within $1\,..\,n$. In this case, we would end up writing an **if** statement with $(\exists\, i \in 1\,..\,n \bullet p(i))$ and $(\forall\, i \in 1\,..\,n \bullet \neg\, p(i))$ as guards.

In this case, a more "invasive" change is required. In order to effect this kind of change, we need some algebraic properties of program disjunction.

4 Properties of Program Disjunction

Program disjunction has a number of properties that can be used in implementing specifications expressed using it. Here we only give a few such properties, focusing on the most essential ones and those needed for our examples; a more comprehensive collection, along with proofs, can be found in [Gro00].

4.1 Basic properties

A disjunction of two specification statements can be replaced by a single specification statement. We first consider the case for specification statements with the same frame; this is easily obtained from the definition of $\curlyvee$ and the definitions of specification and **if** statements [Mor94].

Law 1

$$
w\colon\left[P \;\big/\; R\right] \curlyvee w\colon\left[P' \;\big/\; R'\right] \;=\; w\colon\left[P \vee P' \;\Big/\; \begin{array}{c} P[w \setminus w_0] \wedge R \\ \vee \\ P'[w \setminus w_0] \wedge R' \end{array}\right]
$$

where w_0 denotes the initial values of the variables in w, and $P[x \setminus y]$ is the predicate obtained by replacing each occurrence of a variable in x by the corresponding variable in y.

It is easy to see that this coincides with the form given in Section 1 under the assumptions given there, and agrees with the example in Section 2.

The version for specification statements with different frames is a bit more complicated — we need to first apply Morgan's *expand frame* law to unify the frames of the two components.

Law 2

$$u\!:\!\left[P \;\middle/\; R\right] \curlyvee v\!:\!\left[P' \;\middle/\; R'\right] \;=\; u,v\!:\!\left[P \vee P' \;\middle/\; \begin{array}{c} P[u \setminus u_0] \wedge R \wedge \Xi(v-u) \\ \vee \\ P'[v \setminus v_0] \wedge R' \wedge \Xi(u-v) \end{array}\right]$$

where $v-u$ denotes the set of variables that are in v but not in u, and Ξs stands for $s = s_0$, which means that each variable in s is equal to the corresponding variable in s_0, i.e. to its initial value.

Disjunction is idempotent, commutative and associative.

Law 3

(i) $S \curlyvee S = S$

(ii) $S \curlyvee T \;=\; T \curlyvee S$

(iii) $S \curlyvee (T \curlyvee U) \;=\; (S \curlyvee T) \curlyvee U$

4.2 Monotonicity

In order to implement a specification expressed as a disjunction, say $S \curlyvee T$, in the way proposed above, we need to be able to refine S and T independently, to obtain, say $U \curlyvee V$, where $S \sqsubseteq U$ and $T \sqsubseteq V$. Unfortunately this is not always the case; i.e. we can find programs S, T, U and V such that $S \sqsubseteq U$ and $T \sqsubseteq V$, but $S \curlyvee T \not\sqsubseteq U \curlyvee V$.

For example, take $S \;\widehat{=}\; x\!:\!\left[a = 0 \;\middle/\; x = 0\right]$, $T \;\widehat{=}\; x\!:\!\left[a = 1 \;\middle/\; x = 1\right]$, $U \;\widehat{=}\; x := 0$ and $V \;\widehat{=}\; x := 1$.

In this case, we have $x\!:\!\left[a = 0 \;\middle/\; x = 0\right] \sqsubseteq x := 0$, i.e. $S \sqsubseteq U$, and $x\!:\!\left[a = 1 \;\middle/\; x = 1\right] \sqsubseteq x := 1$, i.e. $T \sqsubseteq V$. But $S \curlyvee T \not\sqsubseteq U \curlyvee V$, since we have:

$$\begin{array}{lcl} S \curlyvee T & = & x\!:\!\left[a = 0 \;\middle/\; x = 0\right] \curlyvee x\!:\!\left[a = 1 \;\middle/\; x = 1\right] \\ & = & x\!:\!\left[a = 0 \vee a = 1 \;\middle/\; (a = 0 \Rightarrow x = 0) \wedge (a = 1 \Rightarrow x = 1)\right] \end{array}$$

which is not refined by:

$$\begin{array}{lcl} U \curlyvee V & = & x := 0 \curlyvee x := 1 \\ & = & x\!:\!\left[\mathit{true} \;\middle/\; x = 0 \vee x = 1\right] \end{array}$$

The problem here is that in refining $x\!:\!\left[a = 0 \;\middle/\; x = 0\right]$ to $x := 0$, we have introduced behaviours for inputs in the domain of $x\!:\!\left[a = 1 \;\middle/\; x = 1\right]$ which are inconsistent with those specified by $x\!:\!\left[a = 1 \;\middle/\; x = 1\right]$, and similarly in the refinement of $x\!:\!\left[a = 1 \;\middle/\; x = 1\right]$ to $x := 1$. Thus, $x := 0 \curlyvee x := 1$ can set x to 1 when a is 0, whereas $x\!:\!\left[a = 0 \;\middle/\; x = 0\right] \curlyvee x\!:\!\left[a = 1 \;\middle/\; x = 1\right]$ does not allow this behaviour.

Technically, this means that $\curlyvee$ is not monotonic with respect to refinement. Such monotonicity may be taken as suggesting that $\curlyvee$ can not (or should not) be used in software development, and that specifications using it should be expanded before development begins (e.g. [Mah99]). However, this need not be the case; and it isn't really an option for our purposes in program modification. We can obtain a weak monotonicity result showing that refinements can be performed on the components of a disjunction, provided that this does not introduce conflicting behaviours for inputs in the domain of the other component. The required conditions in fact provide useful guidance in implementing the disjuction.

The following is a simple version of this law, which just ensures that the refinement does not add any behaviours for inputs in the domain of the other components.

Law 4 (*Weak monotonicity 1*) $S \curlyvee T \sqsubseteq U \curlyvee V$, if $S \sqsubseteq U$, $T \sqsubseteq V$, $\mathsf{pre}\ S \wedge \mathsf{pre}\ V \Rrightarrow \mathsf{pre}\ T$ and $\mathsf{pre}\ T \wedge \mathsf{pre}\ U \Rrightarrow \mathsf{pre}\ S$.

Note that the example above does not satisfy the provisos for this law, as will any refinement that does not increase the domain of termination (i.e. doesn't weaken the precondition).

The following version allows refinements that add behaviours for inputs in the domain of the other component, so long as such behaviour is consistent with that component.

Law 5 (*Weak monotonicity 2*) $S \curlyvee T \sqsubseteq U \curlyvee V$, if $S \sqsubseteq U$, $T \sqsubseteq V$, $\left\{\mathsf{pre}\ U \wedge \neg\, \mathsf{pre}\ S\right\}$; $T \sqsubseteq U$ and $\left\{\mathsf{pre}\ V \wedge \neg\, \mathsf{pre}\ T\right\}$; $S \sqsubseteq V$.

The statement $\left\{A\right\}$ is an *assertion*, which has no effect (i.e. is equivalent to **skip**) when A holds, and is unpredictable (i.e. is equivalent to **abort**) otherwise. The condition $\left\{\mathsf{pre}\ U \wedge \neg\, \mathsf{pre}\ S\right\}$; $T \sqsubseteq U$ thus says that U is consistent with T for initial states where U terminates but S doesn't.

4.3 Distribution laws

Disjunctions can often be distributed over programs with similar structure. The following laws show some of the simplest cases when this can be done — more general laws can be obtained, but the applicability conditions are often too complex to be useful (see [Gro00]). In most cases the provisos just say that the transformation does not affect termination of either component.

We can combine two variable declaration blocks declaring the same variable, provided that whether their bodies terminate does not depend on the initial value of the declared variable.

Law 6 $[\![\, \mathbf{var}\ x \bullet S \,]\!] \curlyvee [\![\, \mathbf{var}\ x \bullet T \,]\!] = [\![\, \mathbf{var}\ x \bullet S \curlyvee T \,]\!]$, if $(\forall x \bullet \mathsf{pre}\ S) \equiv \mathsf{pre}\ S$ and $(\forall x \bullet \mathsf{pre}\ T) \equiv \mathsf{pre}\ T$.

We can similarly combine a block declaring a variable x and any other program, say T, not contining x, by first surrounding T by a redundant declaration of x. This relies on the fact that $T = [\![\, \mathbf{var}\ x \bullet T \,]\!]$ if x does not occur free in T.

We can move a common leading statement out of a disjunction, provided that it is terminating, deterministic and feasible.

Law 7 $S\ ;\ T \curlyvee S\ ;\ U = S\ ;\ (T \curlyvee U)$, if S is terminating, deterministic and feasible.

We can move a common trailing statement out of a disjunction, provided that it always terminates.

Law 8 $S\ ;\ U \curlyvee T\ ;\ U = (S \curlyvee T)\ ;\ U$, if U is terminating.

This proviso can be weakened to $\mathsf{pre}\ S \Rrightarrow \mathsf{pre}\ (S\ ;\ U)$ and $\mathsf{pre}\ T \Rrightarrow \mathsf{pre}\ (T\ ;\ U)$, which says that U terminates when executed following termination of either S or T.

We can combine two **if** statements, pushing the disjunction into the branches of the resulting program. The following law gives the simple case where both **if** statements have the same pair of complementary guards.

Law 9 $\textbf{if } G \to S \,[\!]\, \neg G \to T \textbf{ fi} \curlyvee \textbf{if } G \to U \,[\!]\, \neg G \to V \textbf{ fi} \;=\; \textbf{if } G \to S \curlyvee U \,[\!]\, \neg G \to T \curlyvee V \textbf{ fi}$

We can combine two loops with the same loop body, provided that the common body if S is terminating, deterministic and feasible.

Law 10 $\textbf{do } G \to S \textbf{ od} \curlyvee \textbf{do } H \to S \textbf{ od} \;\sqsubseteq\; \textbf{do } G \wedge H \to S \textbf{ od}$, if S is terminating, deterministic and feasible.

Note that this law does not give an equality, since the right hand side always takes result of the first loop to terminate.

We can obtain a stronger version of this law by observing that when combined in this way, the bodies of the two loops only have to display the same behaviour for initial states where $G \wedge H$ holds. This means we can use the guard of one loop to simplify the body of the other.

Law 11 $\textbf{do } G \to S \textbf{ od} \curlyvee \textbf{do } H \to T \textbf{ od} \;\sqsubseteq\; \textbf{do } G \wedge H \to U \textbf{ od}$, if U is terminating, deterministic and feasible, and $\big\{G \wedge H\big\} \,;\, S \;=\; \big\{G \wedge H\big\} \,;\, U \;=\; \big\{G \wedge H\big\} \,;\, T$.

5 Implementing Program Disjunction II

We now consider how the above properties of program disjunction can be used to modify an existing program to incorporate the kind of modification described in Section 1. In particular, we show how we can incorporate a modification in a way that alters the structure of the existing program, as suggested at the end of Section 3.

In this section, we present a basic form of our approach, in which we construct a complete implementation for the modification and then use the distribution laws to merge this with the existing program. Section 6 will present a more flexible version.

Consider again the search example discussed in Section 3. We assume we have an implementation for the initial specification:

$$k \mathbin{:} \left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; \begin{matrix} k \in 1\,..\,n \\ p(k) \end{matrix} \right] \;\sqsubseteq\; \begin{array}{l} k := 1\,; \\ \textbf{do } \neg\, p(k) \to \\ \qquad k := k + 1 \\ \textbf{od} \end{array} \tag{1}$$

And we wish to extend this to obtain an implementation for the extended specification:

$$k \mathbin{:} \left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; \begin{matrix} k \in 1\,..\,n \\ p(k) \end{matrix} \right] \curlyvee\; k \mathbin{:} \left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n \right] \tag{2}$$

First, we must check that the refinement in (1) can be substituted for the left-hand side of (2). This will certainly be the case if $p(k)$ does not hold for any k, since in this case the loop will not terminate, so the domain of termination is not increased.[1] If $p(k)$ does hold for some value of k greater than n, the loop will terminate with a value satisfying the right-hand side of (2). Thus, although the domain of termination is increased, the added behaviour is consistent with the required modification, so this satisfies the conditions for the second weak monotonicity law (law 5).

[1] We will avoid issues of undefinedness and assume that $p(k)$ is defined for all k.

We now need to find an implementation for the right-hand side of (2) which satisfies the provisos for law 4. We also want this implementation to have a "similar" algorithmic structure to that for S, so that the two can be merged. Such an implementation is:

$$
\begin{array}{l}
k := 1\ ; \\
\mathbf{do}\ k \leq n \rightarrow \\
\quad \left\{\neg\, p(k)\right\}\ ; \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}
$$

where the assertion $\left\{P\right\}$ ensures that the domain of termination is not increased by making the program abort if a k satisfying p is found. This refinement is therefore allowed within the $\curlyvee$, by the first weak monotonicity law (law 4).

We will not attempt to explain here how this program was constructed (see Section 6), but instead concentrate to showing how it can be combined with the original program.

Now we can use the distribution and elimination laws for program disjunction to merge the two programs.

After substituting the two programs for the two parts of the extended specification (2), we have:

$$
\left(\begin{array}{l}
k := 1\ ; \\
\mathbf{do}\ \neg p(k) \rightarrow \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}\right)
\curlyvee
\left(\begin{array}{l}
k := 1\ ; \\
\mathbf{do}\ k \leq n \rightarrow \\
\quad \left\{\neg\, p(k)\right\}\ ; \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}\right)
$$

We can move the initial assignment out of the disjunction, since assignment statements are terminating, deterministic and feasible.

$$
\begin{array}{ll}
= & \langle \text{ law } 7 \ \rangle \\
 & k := 1\ ; \\
 & \left(\begin{array}{l}
\mathbf{do}\ \neg p(k) \rightarrow \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}\right)
\curlyvee
\left(\begin{array}{l}
\mathbf{do}\ k \leq n \rightarrow \\
\quad \left\{\neg\, p(k)\right\}\ ; \\
\quad k := k + 1 \\
\mathbf{od}
\end{array}\right)
\end{array}
$$

We can now combine the loops, since their bodies are identical when both guards hold, and assignment statements are terminating, deterministic and feasible. Note that this requires the stronger loop law.

$$
\begin{array}{ll}
\sqsubseteq & \langle \text{ law } 11 \ \rangle \\
 & \mathbf{do}\ \neg p(k) \wedge k \leq n \rightarrow \\
 & \quad k := k + 1 \\
 & \mathbf{od}
\end{array}
$$

6 Interleaving Implementation and Merging

The approach illustrated in Section 5 is not ideal, since it involves constructing a complete implementation for the modification before merging it with the existing program. The main reasons for presenting it are that it shows clearly just what is involves in integrating the modification with the existing program, and that it works when we have no derivation for the initial program.

If we have a formal derivation for the existing program, we can adopt a more realistic and more flexible process in which case we can interleave the design of the program to implement the modification and its integration with the existing program. In this case, the refinement steps in the original derivation can guide the steps required for the modification; we can also alter the derivation of the original program if necessary to make the integration work more easily.

We will illustrate this approach by again considering the above example.

6.1 Derivation for Search Program

First, we present a derivation for the original search program. This requires no explanation, except to note that proof obligations are preceded by "Ob".

$$k \colon \left[\exists\, i \in 1 \mathinner{..} n \bullet p(i) \;\middle/\; k \in 1 \mathinner{..} n \wedge p(k) \right] \tag{1}$$

(1) $\sqsubseteq$ $\langle$ *split specification*; $I \mathrel{\widehat{=}} k \in 1 \mathinner{..} n \wedge (\exists\, i \in k \mathinner{..} n \bullet p(i))$ $\rangle$

$$k \colon \left[\exists\, i \in 1 \mathinner{..} n \bullet p(i) \;\middle/\; I \right] ; \tag{2}$$

$$k \colon \left[I \;\middle/\; k \in 1 \mathinner{..} n \wedge p(k) \right] \tag{3}$$

(2) $\sqsubseteq$ $\langle$ *introduce assignment* $\rangle$

$k := 1$

Ob: $(\exists\, i \in 1 \mathinner{..} n \bullet p(i)) \Rrightarrow (\exists\, i \in 1 \mathinner{..} n \bullet p(i))$

(3) $\sqsubseteq$ $\langle$ *introduce* **do** , inv: I, var: $n - k$ $\rangle$

do $\neg\, p(k) \rightarrow$

$$k \colon \left[I \wedge \neg\, p(k) \;\middle/\; I \wedge\ k_0 < k \leq n \right] \tag{4}$$

od

Ob: $I \wedge p(k) \Rrightarrow k \in 1 \mathinner{..} n \wedge p(k)$

(4) $\sqsubseteq$ $\langle$ *introduce assignment* $\rangle$

$k := k + 1$

Ob: $I \wedge \neg\, p(k) \Rrightarrow I[k \setminus k + 1] \wedge k < k + 1 \leq n$

The result is as indicated previously:

$$k \colon \left[\exists\, i \in 1 \mathinner{..} n \bullet p(i) \;\middle/\; \begin{array}{c} k \in 1 \mathinner{..} n \\ p(k) \end{array} \right] \sqsubseteq \begin{array}{l} k := 1\,; \\ \mathbf{do}\ \neg\, p(k) \rightarrow \\ \quad k := k + 1 \\ \mathbf{od} \end{array}$$

6.2 Extending the search to allow for no p

We now show how to integrate the implementation of the required modification with this derivation. We begin with the extended specification, as in Section 5.

$$k\!:\!\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; k \in 1\,..\,n \wedge p(k)\right] \;\curlyvee \tag{1}$$

$$k\!:\!\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; k \notin 1\,..\,n\right] \tag{1'}$$

In other circumstances, the obvious thing to do would be to refine ($1'$) to, either, $k := 0$ or $k := n + 1$. We cannot do that in this case, where ($1'$) is a component of a disjunction, because of the monotonicity law. Instead, we will have to try to refine ($1'$) in a similar way to (1) and merge them.

We can refine (1) as we did in the original. The implementation of ($1'$) is going to require a loop, so that it can be merged with the existing loop in the implementation of (1). So we refine ($1'$) in the same way as (1), choosing an intermediate assertion that can be used as a loop invariant. As usual, choosing a loop invariant requires some forethought, and in practice, we may only be able to determine the appropriate assertion to use when we come to merge the two programs. In this case, the loop invariant is going to record what we know as a result of the iterations performed so far (i.e. that no value satsifying p has been found). It will also prove to be convient if we can establish this invariant using the same initialisation as the invariant for the other loop.

$$\sqsubseteq \quad \langle\ \textit{split};\ I' \mathrel{\widehat=} k \in 1\,..\,n{+}1 \wedge (\forall\, i \in 1\,..\,k{-}1 \bullet \neg\, p(i))\ \rangle$$

$$k\!:\!\left[\exists\, i \in 1\,..\,n \bullet p(i) \;\middle/\; I\right];\, k\!:\!\left[I \;\middle/\; k \in 1\,..\,n \wedge p(k)\right] \tag{2 ; 3}$$

$$\curlyvee$$

$$k\!:\!\left[\forall\, i \in 1\,..\,n \bullet \neg\, p(i) \;\middle/\; I'\right];\, k\!:\!\left[I' \;\middle/\; k \notin 1\,..\,n\right] \tag{2' ; 3'}$$

We can refine (2) to an assignment as before. We can similarly refine ($2'$) to the same assignment.

$$\sqsubseteq \quad \langle\ \textit{introduce assignment},\ \text{twice}\ \rangle$$

$$k := 1\,;\, k\!:\!\left[I \;\middle/\; k \in 1\,..\,n \wedge p(k)\right]$$

$$\curlyvee$$

$$k := 1\,;\, k\!:\!\left[I' \;\middle/\; k \notin 1\,..\,n\right]$$

Our choice of intermediate assertion in the refinement of ($1'$) has allowed the same initialisation to be used on both sides of the disjunction. We can now move the common assignment out of the disjunction, since assignment statements are terminating, deterministic and feasible.

$$\sqsubseteq \quad \langle\ \text{law 7}\ \rangle$$

$$k := 1\,;$$

$$(\,k\!:\!\left[I \;\middle/\; k \in 1\,..\,n \wedge p(k)\right] \curlyvee k\!:\!\left[I' \;\middle/\; k \notin 1\,..\,n\right]\,) \tag{$3 \curlyvee 3'$}$$

We refine (3) to a loop as before, and as anticipated above, introduce a similar loop for ($3'$).

$$
\begin{array}{l}
\sqsubseteq \;\; \left\langle \begin{array}{l} \textit{introduce}\ \mathbf{do}\ \text{, inv: } I\text{, var: } n-k \\ \textit{introduce}\ \mathbf{do}\ \text{, inv: } I'\text{, var: } n-k+1 \end{array} \right\rangle \\
\quad \mathbf{do}\ \neg\, p(k) \rightarrow \\
\qquad k\,\mathbf{:}\!\left[I \wedge \neg\, p(k) \;\middle/\; I \wedge k_0 < k \leq n \right] \qquad (4) \\
\quad \mathbf{od} \\
\quad\;\; \curlyvee \\
\quad \mathbf{do}\ k \leq n \rightarrow \\
\qquad k\,\mathbf{:}\!\left[I' \wedge k \leq n \;\middle/\; I' \wedge k_0 < k \leq n{+}1 \right] \qquad (4') \\
\quad \mathbf{od} \\
\quad \mathsf{Obs:}\;\; I \wedge p(k) \Rrightarrow k \in 1\,..\,n \wedge p(k) \;\; \text{(as before)} \\
\qquad\quad\;\, I' \wedge k > n \Rrightarrow k \notin 1\,..\,n
\end{array}
$$

We can't merge the loops at this point, because their bodies are different, so instead we proceed to refine their bodies. We can refine (4) to an assignment as before, and hope to refine (4′) in a similar way. Unfortunately, we cannot refine (4′) to $k := k + 1$, because we cannot infer $\neg\, p(k)$ from its precondition. We will side step this problem for the moment by including $\neg\, p(k)$ as an assertion.

$$
\begin{array}{lll}
(4) & \sqsubseteq & k := k + 1 \\
(4') & \sqsubseteq & \left\{\neg\, p(k)\right\}\ \mathbf{;}\ k := k + 1
\end{array}
$$

We can finally combine the loops, again using the strong loop law, because the loop bodies are different but can are the same when both loop guards hold.

$$
\begin{array}{l}
\sqsubseteq \;\; \langle \text{ law 10 } \rangle \\
\quad \mathbf{do}\ \neg\, p(k) \wedge k \leq n \rightarrow \\
\qquad k := k + 1 \\
\quad \mathbf{od}
\end{array}
$$

7 Conclusions

We have introduced a program disjunction operator and show how it can be used to specify a common form of modification, where the program is extended to handle additional inputs. We have also shown how properties of this operator can be used to incorporate such modifications when they require changes to the structure of the program. We have presented two versions of this process: one which requires a complete program implementing the modification to be constructed and then integrated with the existing program; the other involves interleaving the design of the program to implement the modification with its integration with the existing program. The greatest problem in this process arises from the fact that disjunction is not monotonic, but we have identified the cases where components of a disjunction can be refined independently and shown how the associated condition can provide guidance in implementing a modification.

Several other authors have considered adding a form of program conjunction operator to the refinement calculus. Ward [War93] and Mahony [Mah99] wanted to provide an operator that was identical to schema disjunction in Z, and so allowed the disjunction to modify any variable that either component could modify. Utting [Utt92] used a similar operator as a basis for defining type construction operators. Back and Butler [BB95, BB98] take a more theoretical approach, and define a sum operator reminiscent of our disjunction operator as a category theoretic dual of a product operator. None of these authors explore the use of these operators in program construction in great deal, and none consider their use in implementing modifications.

Acknowledgements

Part of this work was completed while I was on study leave at Oxford University. I would like to thank the Oxford University Computing Laboratory, Kellogg College, and my host, Jim Woodcock, for their hospitality and support during this period.

References

[BB95] R.J.R Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 128–158, 1995.

[BB98] R.J.R Back and M.J. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35(11):921–949, 1998.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[Gro00] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[Mah99] Brendan Mahony. The least conjunctive refinement and promotion in the refinement calculus. *Formal Aspects of Computing*, 11:75–105, 1999.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual (second edition)*. Prentice Hall International, 1992.

[Utt92] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, 1992.

[War93] Nigel Ward. Adding specification constructors to the refinement calculus. In J. C. P. Woodcock and P. G. Larsen, editors, *Formal Methods Europe*, volume 670 of *Lecture Notes in Computer Science*, pages 652–670. Springer-Verlag, April 1993.