

The Specification Statement

CARROLL MORGAN

Oxford University

Dijkstra's programming language is extended by *specification statements*, which specify parts of a program "yet to be developed." A weakest precondition semantics is given for these statements so that the extended language has a meaning as precise as the original.

The goal is to improve the *development* of programs, making it closer to manipulations within a single calculus. The extension does this by providing one semantic framework for specifications and programs alike: Developments begin with a program (a single specification statement) and end with a program (in the executable language). And the notion of *refinement* or *satisfaction*, which normally relates a specification to its possible implementations, is automatically generalized to act between specifications and between programs as well.

A surprising consequence of the extension is the appearance of *miracles*: program fragments that do not satisfy Dijkstra's *Law of the Excluded Miracle*. Uses for them are suggested.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.2 [**Software Engineering**]: Tools and Techniques—*top-down programming*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Development calculus, guarded commands, miracles, procedural abstraction, program refinement, weakest preconditions

1. INTRODUCTION

Dijkstra in [4] introduces the *weakest precondition* of a program P with respect to a postcondition $post$; following [8] we will write this $P\langle post \rangle$. In this style, a *specification* of a program P is written

$$pre \Rightarrow P\langle post \rangle.$$

This means "if activated in a state for which pre holds, the program P must terminate in a state for which $post$ holds."

In traditional top-down developments, we build algorithmic structure around a collection of ever-decreasing program fragments "yet to be implemented," and

Author's address: Programming Research Group, Oxford University, 8-11 Keble Road, Oxford OX1 3QD, United Kingdom.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0164-0925/88/0700-0403 \$01.50

at any stage we have specifications for those fragments. Thus one finds the dictions

$$\begin{array}{c} \vdots \\ P; \\ \vdots \end{array}$$

where $pre \Rightarrow P \langle post \rangle$.

The letter P stands for the missing fragment, and the **where** clause gives its specification. But in our approach, we write instead

$$\begin{array}{c} \vdots \\ [pre, post]; \\ \vdots \end{array} \quad (1)$$

We write the specification itself at the point to be occupied by its implementation. More significantly, by giving a weakest precondition semantics to $[pre, post]$, we make this intermediate stage (1) into a *program*—albeit an abstract one.

We see program development as analogous to solving equations: One transforms an abstract program into a concrete one, just as one transforms a complex equation (e.g., $x^2 - x - 1 = 0$) into a simple equality (e.g., $x = (1 + 5^{1/2})/2$). For such formulas, the manipulations are mediated by the relation of implication: The simple equality *implies* the complex equation.

The abstract-to-concrete transformation of programs is mediated by a relation \sqsubseteq of *refinement*, which is defined so that $P \sqsubseteq Q$ means “any specification satisfied by P is satisfied by Q also.” This relation can appear between abstract programs (specifications), between concrete programs, or between one and the other. As we write

$$x^2 - x - 1 = 0 \Leftarrow x = \frac{1 + \sqrt{5}}{2},$$

so we will write with complete rigor

$$x : [x^2 - x - 1 = 0] \sqsubseteq x := \frac{1 + \sqrt{5}}{2}.$$

An unexpected consequence of our extension is the introduction of abstract programs that do not obey Dijkstra’s *Law of the Excluded Miracle*. These correspond to specifications that have no concrete solution, just as negative numbers stand for insoluble equations in elementary arithmetic (“3 from 2 won’t go”). An example is the statement $[true, false]$; we will see that the following holds:

$$true \Rightarrow [true, false] \langle false \rangle.$$

But just as negative numbers simplify arithmetic, miracles simplify program derivation.

Our overall contribution is *uniformity*: We place program development within reach, in principle, of a single calculus. We expect this to be useful not only at the level of small intricacies, but on the larger scale also. Modules, for example, can be written using specification statements instead of concrete constructions: Thus we have *specifications* of modules. Because of the generality of our approach, any structuring facility offered by the target programming language is also offered to specifications.

2. SPECIFICATION STATEMENTS

We introduce the syntax and weakest precondition semantics of specification statements, moving from simple to more general forms.

2.1 The Simple Form

The simple specification statement $[pre, post]\langle R \rangle$ comprises two predicates over the program variables \vec{v} . Informally, it means “assuming an initial state satisfying pre , establish a final state satisfying $post$.” Its precise definition is (using \triangleq for “is defined to be”)

Definition 1. $[pre, post]\langle R \rangle \triangleq pre \wedge (\forall \vec{v}. post \Rightarrow R)$.

For example, assuming \vec{v} is just the single variable x , we have

$$\begin{aligned} [true, x = 1]\langle R \rangle &= true \wedge (\forall x. x = 1 \Rightarrow R) \\ &= R[x \setminus 1]. \end{aligned}$$

The substitution $[x \setminus 1]$ denotes syntactic replacement of x by 1 in the usual way.

2.2 Confining Change

We allow the changing of variables to be confined to those of interest. For any subvector \vec{w} of \vec{v} , the statement $\vec{w}:[pre, post]$ has the following informal meaning:

Assuming an initial state satisfying pre , establish a final state satisfying $post$ while changing only variables in \vec{w} .

The precise definition of $\vec{w}:[pre, post]$ is

Definition 2. $\vec{w}:[pre, post]\langle R \rangle \triangleq pre \wedge (\forall \vec{w}. post \Rightarrow R)$.

The only change from Definition 1 is that the vector of quantified variables is now \vec{w} rather than \vec{v} . Taking \vec{v} to be “ x, y ”, we have

$$\begin{aligned} x:[true, x = y]\langle R \rangle &= true \wedge (\forall x. x = y \Rightarrow R) \\ &= R[x \setminus y]. \end{aligned}$$

Since $(x := y)\langle R \rangle$ equals $R[x \setminus y]$ also, we have shown that $x:[true, x = y]$ and $x := y$ have the same meaning. If we allow *both* x and y to change, this is no

longer true:

$$\begin{aligned} x, y: [true, x = y] \langle R \rangle & \\ &= true \wedge (\forall x, y. x = y \Rightarrow R) \\ &= (\forall y. R[x \setminus y]). \end{aligned}$$

The statement $x, y: [true, x = y]$ can set y to x , x to y , or both x and y to some third value.

2.3 Referring To The Initial State

Occurrences of 0-subscripted variables \vec{v}_0 in *post* refer to the values held by those variables *initially*. We reserve 0 subscripts for this purpose and assume that they do not occur as ordinary variables in programs. We now have the following informal meaning for $\vec{w}: [pre, post]$:

Assuming an initial state satisfying *pre*, change only variables in \vec{w} to establish *post*, in which 0-subscripted variables refer to the values those variables held initially.

The precise definition appears below. In practice, however, we usually apply the simpler version given in Lemma 1, which follows.

Definition 3. $\vec{w}: [pre, post] \langle R \rangle \triangleq pre \wedge (\forall \vec{w}. post[\vec{v}_0 \setminus \vec{f}] \Rightarrow R)[\vec{f} \setminus \vec{v}]$, where \vec{f} is some fresh vector of variables.

The use of fresh variables \vec{f} in Definition 3 only avoids interference with possible occurrences of \vec{v}_0 in R , which are rare in practice. Usually we can apply the simpler construction below:

LEMMA 1. *If R contains no 0-subscripted variables,*

$$\vec{w}: [pre, post] \langle R \rangle = pre \wedge (\forall \vec{w}. post \Rightarrow R)[\vec{v}_0 \setminus \vec{v}].$$

PROOF: Immediate from Definition 3. \square

Notice that if *post* contains no \vec{v}_0 , then both Definition 3 and Lemma 1 reduce to Definition 2.

For example, taking \vec{v} to be “ x, y ” as before, we have from Lemma 1

$$\begin{aligned} x: [true, x = x_0 + y_0] \langle R \rangle & \\ &= true \wedge (\forall x. x = x_0 + y_0 \Rightarrow R)[x_0, y_0 \setminus x, y] \\ &= R[x \setminus x_0 + y_0][x_0, y_0 \setminus x, y] \\ &= R[x \setminus x + y]. \end{aligned}$$

2.4 The Implicit Precondition

We allow the omission of the precondition in a specification statement. The informal meaning of $\vec{w}: [post]$ is:

Assuming it is possible to do so, change only variables in \vec{w} to establish *post*, in which 0-subscripted variables refer to the values those variables held initially.

The meaning is given syntactically—we make the missing precondition explicit:

Definition 4. $\vec{w}: [post] \triangleq \vec{w}: [(\exists \vec{w}. post)[\vec{v}_0 \setminus \vec{v}], post]$.

For example, we can write

$$m:[l \leq m \leq h] \quad \text{for} \quad m:[l \leq h, l \leq m \leq h]$$

$$\text{and } i:[a[i] = v] \quad \text{for} \quad i:[(\exists i. a[i] = v), a[i] = v]$$

The first statement places m between l and h ; the second locates an index i of value v in array a . If in either case the result is not achievable (e.g., if l exceeds h , or v does not occur in a), the statement can abort.

2.5 Generalized Assignment

We generalize assignment by giving the following meaning to the statement $x:\odot e$ for any binary relation \odot :

Assuming it is possible to do so, assign to x a value bearing the relation \odot to the expression e , where occurrences of x in e refer to its initial value.

Ordinary assignment statements are now the special case in which \odot is “=”. But we can also write, for example,

$$x:\in s \quad \text{for} \quad \text{if possible, choose } x \text{ from } s$$

$$\text{and } n:< n \quad \text{for} \quad \text{decrease } n.$$

The definition is given syntactically:

Definition 5. $x:\odot e \triangleq x:[x \odot e[x \setminus x_0]]$.

With this definition, our abbreviations above become, respectively,

$$x:[x \in s] \quad (\text{that is, } x:[s \neq \{\}, x \in s])$$

$$\text{and } n:[n < n_0].$$

The syntax for generalized assignment was suggested (long ago) by Jean-Raymond Abrial.

3. THE IMPLEMENTATION ORDERING

For programs P and Q , we give $P \sqsubseteq Q$ the informal meaning: “Every specification satisfied by P is satisfied by Q also.” This means that Q is an acceptable replacement for P . Our precise definition is

Definition 6. $P \sqsubseteq Q$ iff for all predicates R ,

$$P \langle R \rangle \Rightarrow Q \langle R \rangle.$$

The following theorem shows Definition 6 to have the property we require.

THEOREM 1. *If $pre \Rightarrow P \langle post \rangle$ and $P \sqsubseteq Q$, then also $pre \Rightarrow Q \langle post \rangle$.*

PROOF. Since $P \sqsubseteq Q$, we have $P \langle post \rangle \Rightarrow Q \langle post \rangle$. The result follows immediately. \square

As an example of refinement between programs, let P be

```

if 2 |  $x \rightarrow x := x \div 2$ 
  || 3 |  $x \rightarrow x := x \div 3$ 
fi,

```

and let Q be

```

if 2 |  $x \rightarrow x := x \div 2$ 
  ||  $\neg(2 | x) \rightarrow x := x \div 3$ 
fi,

```

where $2 | x$ means “2 divides x exactly”, and \div denotes integer division. We have $P \sqsubseteq Q$ because

$$\begin{aligned}
 P\langle R \rangle &= (2 | x \vee 3 | x) \\
 &\quad \wedge (2 | x \Rightarrow R[x \setminus x \div 2]) \\
 &\quad \wedge (3 | x \Rightarrow R[x \setminus x \div 3])
 \end{aligned}$$

and

$$\begin{aligned}
 Q\langle R \rangle &= (2 | x \Rightarrow R[x \setminus x \div 2]) \\
 &\quad \wedge (\neg(2 | x) \Rightarrow R[x \setminus x \div 3]).
 \end{aligned}$$

Thus $P\langle R \rangle \Rightarrow Q\langle R \rangle$ for any R . But Q differs from P in that Q will always terminate, even when $x = 7$. And Q is deterministic: If $x = 6$, Q will establish $x = 3$. In spite of these differences, Q is an acceptable substitute for P , and that is why we can implement P as **IF** $2 | x$ **THEN** $x := x \div 2$ **ELSE** $x := x \div 3$ **END**.

We now state the well-known but crucial fact that the program constructors are monotonic with respect to \sqsubseteq ; only this ensures that refining a fragment (say P above) “in place” in some larger program refines that larger program overall.

THEOREM 2. *If $F(P)$ is a program containing the program fragment P and for another program fragment Q we have $P \sqsubseteq Q$, then*

$$F(P) \sqsubseteq F(Q).$$

PROOF. By structural induction, over the program constructors “;”, “**if**”, and “**do**”. \square

4. SUITABILITY OF THE DEFINITIONS

We now show the suitability of our definitions by proving that

$$pre \Rightarrow P\langle post \rangle \quad \text{iff} \quad [pre, post] \sqsubseteq P.$$

In fact, we prove a stronger result, dealing with the general form of Section 2.3.

In long formulas, we sometimes “stack” conjunctions for clarity by writing

$$\left(\begin{array}{l} \textit{this} \\ \textit{that} \end{array} \right) \quad \text{for} \quad (\textit{this} \wedge \textit{that}).$$

Our theorem is a consequence of the following two lemmas.

LEMMA 2. If \tilde{u} and \tilde{w} partition the vector \tilde{v} of program variables, then

$$(pre \wedge \tilde{v} = \tilde{v}_0) \Rightarrow \tilde{w}:[pre, post]\langle post \wedge \tilde{u} = \tilde{u}_0 \rangle.$$

PROOF. Here we must use Definition 3 rather than Lemma 1, since the postcondition contains \tilde{v}_0 . We have

$$(pre \wedge \tilde{v} = \tilde{v}_0) \Rightarrow \tilde{w}:[pre, post]\langle post \wedge \tilde{u} = \tilde{u}_0 \rangle$$

if, by Definition 3,

$$(pre \wedge \tilde{v} = \tilde{v}_0) \Rightarrow pre \wedge \left(\forall \tilde{w}. post[\tilde{v}_0 \setminus \tilde{f}] \Rightarrow \frac{post}{\tilde{u} = \tilde{u}_0} \right) [\tilde{f} \setminus \tilde{v}]$$

$$\text{if } \tilde{v} = \tilde{v}_0 \Rightarrow \left(\forall \tilde{w}. post[\tilde{v}_0 \setminus \tilde{f}] \Rightarrow \frac{post}{\tilde{u} = \tilde{u}_0} \right) [\tilde{f} \setminus \tilde{v}],$$

$$\text{if } \tilde{v} = \tilde{v}_0 \Rightarrow \left(\forall \tilde{w}. post[\tilde{v}_0 \setminus \tilde{f}] \Rightarrow \frac{post}{\tilde{u} = \tilde{u}_0} \right) [\tilde{f} \setminus \tilde{v}_0],$$

$$\text{if } \tilde{v} = \tilde{v}_0 \Rightarrow \left(\forall \tilde{w}. post \Rightarrow \frac{post}{\tilde{u} = \tilde{u}_0} \right),$$

$$\text{if true, since } \tilde{u}, \tilde{w} \text{ partition } \tilde{v}. \quad \square$$

LEMMA 3. If $(pre \wedge \tilde{v} = \tilde{v}_0) \Rightarrow P \langle post \wedge \tilde{u} = \tilde{u}_0 \rangle$, then

$$\tilde{w}:[pre, post] \sqsubseteq P$$

where \tilde{w} and \tilde{u} partition the program variables \tilde{v} .

PROOF.

$$(pre \wedge \tilde{v} = \tilde{v}_0) \Rightarrow P \langle post \wedge \tilde{u} = \tilde{u}_0 \rangle,$$

hence by distributivity of \Rightarrow over weakest preconditions,

$$\begin{aligned} pre \wedge \tilde{v} = \tilde{v}_0 \wedge \left(\forall \tilde{v}. \frac{post}{\tilde{u} = \tilde{u}_0} \Rightarrow R \right) \\ \Rightarrow P \langle R \rangle, \end{aligned}$$

hence

$$pre \wedge \tilde{v} = \tilde{v}_0 \wedge (\forall \tilde{w}. post \Rightarrow R) [\tilde{u} \setminus \tilde{u}_0] \Rightarrow P \langle R \rangle,$$

hence

$$pre \wedge \tilde{v} = \tilde{v}_0 \wedge (\forall \tilde{w}. post \Rightarrow R) \Rightarrow P \langle R \rangle,$$

hence since pre and $P \langle R \rangle$ do not contain \tilde{v}_0 ,

$$pre \wedge (\forall \tilde{w}. post \Rightarrow R) [\tilde{v}_0 \setminus \tilde{v}] \Rightarrow P \langle R \rangle,$$

hence by Lemma 1,

$$\tilde{w}:[pre, post] \langle R \rangle \Rightarrow P \langle R \rangle.$$

Since R was arbitrary, we conclude from Definition 6 that $\tilde{w}:[pre, post] \sqsubseteq P$ as required. \square

Those two lemmas give us our theorem immediately.

THEOREM 3. *If \tilde{w}, \tilde{u} partition, the program variables \tilde{v} , then*

$$(\text{pre} \wedge \tilde{v} = \tilde{v}_0) \Rightarrow P \langle \text{post} \wedge \tilde{u} = \tilde{u}_0 \rangle$$

if and only if

$$\tilde{w} : [\text{pre}, \text{post}] \sqsubseteq P.$$

PROOF. “If” follows from Lemma 2 and Theorem 1; “only if” is Lemma 3 exactly. \square

5. USING SPECIFICATION STATEMENTS

For illustration we take the simplest of examples: We are given an array $a[0 \dots N - 1]$ and must find an index i at which the value v occurs. And we may assume there is such an i . The program is

$$i : \left[\begin{array}{l} 0 \leq i < N \\ a[i] = v \end{array} \right] \quad (2)$$

This is a program, though abstract, and perhaps we can execute it directly (see further below). But for now, we assume not, and so we “solve” it by refining it to statements we can execute.

First we use Definition 4, rewriting

$$i : \left[\begin{array}{l} (\exists i. 0 \leq i < N \wedge a[i] = v), \\ a[i] = v \end{array} \right] \quad (3)$$

We take as invariant

$$\text{Inv} \triangleq \begin{array}{l} 0 \leq i < N \\ (\exists j. i \leq j < N \wedge a[j] = v) \end{array}$$

The variant is $N - i$. With these and Theorem 3, we can prove that (3)

$$\sqsubseteq i : [(\exists i. 0 \leq i < N \wedge a[i] = v), \text{Inv}];$$

do $a[i] \neq v \rightarrow$

$$i : \left[\begin{array}{ll} \text{Inv} & i_0 < i \\ a[i] \neq v, & \text{Inv} \end{array} \right]$$

od

Notice that the fragments “to be developed” are written in line and that the above mixture of abstract and concrete is still a program. The first component we can refine to $i := 0$, and the second we can refine to $i := i + 1$. For illustration, we show the second refinement in more detail: By Theorem 3 we need

$$\left[\begin{array}{l} 0 \leq i < N \\ (\exists j. i \leq j < N \wedge a[j] = v) \\ a[i] \neq v \\ i = i_0 \end{array} \right] \Rightarrow i := i + 1 \left\langle \begin{array}{l} i_0 < i < N \\ (\exists j. i \leq j < N \wedge a[j] = v) \end{array} \right\rangle$$

By the semantics of assignment [4], the consequent is

$$i_0 < i + 1 < N \\ (\exists j. i + 1 \leq j < N \wedge a[j] = v).$$

That follows easily from the antecedent.

Having our development, we may wish to collect it and others into a small “database module” based on arrays. As is typical in modern programming languages, the implementation

```
i := 0;
do a[i] ≠ v → i := i + 1 od
```

would be hidden within the “implementation part” of the module. What should appear in the definition part? We suggest (using the syntax of Modula-2 [21])

```
module Database;
export Find, N;
const N = ?;
var a: array[0 .. N - 1] of ?;
procedure Find(v: ?; var i: [0 .. N - 1]);
begin
  i: [ 0 ≤ i < N
      a[i] = v ]
end Find
:
end Database
```

This is not informal. Except for the “?”, the module contains only constructions whose semantics are known precisely. Now, a programmer wishing to implement (2) can do so directly, using the copy rule of Algol-60 (suitably extended for modules). The programmer just writes $Find(v, i)$, whose meaning is given by substituting the procedure body from the *definition* module. This is discussed further in [15].

Thus we show that our approach applies not only to small constructions, and that in particular it supports the view that the “definition module” specifies the “implementation module.”

6. MIRACLES

In [4] it is stated that for all programs P ,

$$P \langle false \rangle = false. \quad (4)$$

This is no longer true: We have, for example,

$$\begin{aligned} [true, false] \langle false \rangle \\ &= true \wedge (\forall \vec{v}. false \Rightarrow false) \\ &= true. \end{aligned}$$

The statement $[true, false]$ is called a *miracle* because it implements anything: We have for all R that $P\langle R \rangle \Rightarrow [true, false]\langle R \rangle$, and so for any P whatsoever,

$$P \sqsubseteq [true, false].$$

Although $[true, false]$ implements anything, it cannot itself be implemented by anything free of miracles. This is because “ P is free of miracles” implies by (4) that $P\langle false \rangle = false$; and so taking $R = false$ in Definition 6, we have $[true, false] \not\sqsubseteq P$.

A program that cannot be rid of miracles is *infeasible* in the following precise way:

Definition 7. We say that a program P is feasible iff

$$P\langle false \rangle = false.$$

Otherwise it is *infeasible*, or *miraculous*.

Clearly, all programs free of specification statements are by (4) feasible: Indeed, they are “implementations” already. For specifications, however, we have the following:

THEOREM 4. $\tilde{w} : [pre, post]$ is feasible iff

$$pre \Rightarrow (\exists \tilde{w}. post)[\tilde{v}_0 \setminus \tilde{v}].$$

PROOF. By Definitions 3, 7, and predicate calculus. \square

Miracles can arise “accidentally” in program development if we make an incorrect design step; this is discussed in more detail in [10] and [16]. For the present, we take a trivial example: We (mistakenly) want to implement $x : [x = 0]$ as a sequential composition whose second component is $x := y$. That is, we want to solve the following formula for P :

$$x : [x = 0] \sqsubseteq P; x := y. \quad (5)$$

By Theorem 3, we have (5)

iff

$$(x = x_0 \wedge y = y_0) \Rightarrow (P; x := y)\langle x = 0 \wedge y = y_0 \rangle,$$

iff by sequential composition

$$(x = x_0 \wedge y = y_0) \Rightarrow P\langle x := y \langle x = 0 \wedge y = y_0 \rangle \rangle,$$

iff

$$(x = x_0 \wedge y = y_0) \Rightarrow P\langle y = y_0 = 0 \rangle,$$

iff by Theorem 3 again,

$$x : [true, y = 0] \sqsubseteq P.$$

We have found our solution P by showing unconditionally that

$$x : [x = 0] \sqsubseteq x : [true, y = 0]; x := y.$$

In fact, the above shows that $x: [true, y = 0]$ is the most general solution of (5), and so we take it as representative of them all, calling it “the” solution. This development technique, in which formulas like (5) are so solved, is the subject of [10].

But after all, the statement $x: [true, y = 0]$ is infeasible; and the importance of the example is its illustration of that consequence of mistaken design steps. The formula (5) is *not* insoluble, but we cannot develop executable code from its solution.

7. GUARDED COMMANDS ARE MIRACLES

Miracles are clearly a strict extension of our programming capabilities since they cannot be executed. We now show how close miracles are, nevertheless, to being in the original language.

A guarded command has the syntactic form

$$B \rightarrow P,$$

where B is a Boolean expression and P is the command guarded. Originally, these occurred only within *if* and *do* constructions. Here we give meaning to guarded commands standing alone.

Informally, we say that a “naked” guarded command *cannot* be executed unless its guard is true. More precisely, we have

$$\textit{Definition 8. } (B \rightarrow P)\langle R \rangle \triangleq B \Rightarrow P\langle R \rangle.$$

If B is true, then $B \rightarrow P$ behaves like P . But if B is false, we consider $B \rightarrow P$ to be miraculous: We may as well, since in this case we *cannot* execute it to check.

Thus we have a compact notation for miracles: They are naked guarded commands whose guards are not identically true. For example, our first miracle $[true, false]$ can be written, for *any* program P ,

$$false \rightarrow P.$$

The following theorem shows that, in fact, every miracle can be written this way. We have

THEOREM 5. *For any program P , feasible or not, there is a guard B and a feasible program Q such that*

$$P = B \rightarrow Q.$$

PROOF. We take

$$B = \neg P\langle false \rangle$$

$$Q = \text{if } B \rightarrow P \text{ fi.}$$

Definition 7 shows that Q is feasible, and Definition 8 shows that the equality holds. \square

We can also define a nondeterministic composition \parallel and a “guardless **if**,” achieving correspondence with the original meaning of these constructs. We have

Definition 9. For any programs P and Q , the program $P \parallel Q$ is defined

$$(P \parallel Q) \langle R \rangle \triangleq P \langle R \rangle \wedge Q \langle R \rangle.$$

Definition 10. For any program P , the program **if** P **fi** is defined

$$\mathbf{if} P \mathbf{fi} \langle R \rangle \triangleq \neg P \langle \mathit{false} \rangle \wedge P \langle R \rangle.$$

Definition 9 is simple nondeterministic choice; in fact,

$$P \parallel Q = \mathbf{if} \mathit{true} \rightarrow P \parallel \mathit{true} \rightarrow Q \mathbf{fi}.$$

Definition 10 is an extension of Dijkstra’s language (necessarily, since it is not monotonic with respect to \sqsubseteq ; it is in fact the “+” operator of [10]). Nevertheless, the meaning that Definitions 8, 9, and 10 give to the **if** construction **if** $(\text{li. } B_i \rightarrow P_i)$ **fi** is exactly as before. We have

THEOREM 6. *If P_i are feasible programs, then*

$$\mathbf{if} (\text{li. } B_i \rightarrow P_i) \mathbf{fi} \langle R \rangle = (\bigvee i. B_i) \wedge (\bigwedge i. B_i \Rightarrow P_i \langle R \rangle).$$

PROOF. Let P be $(\text{li. } B_i \rightarrow P_i)$. By Definitions 9 and 10,

$$P \langle R \rangle = (\bigwedge i. B_i \Rightarrow P_i \langle R \rangle). \quad (6)$$

Hence because the P_i are feasible,

$$\neg P \langle \mathit{false} \rangle = \neg (\bigwedge i. B_i \Rightarrow \mathit{false}) = (\bigvee i. B_i). \quad (7)$$

The result now follows from (6), (7), and Definition 10. \square

Unfortunately, we must note in conclusion that because the construction **if** \dots **fi** is not monotonic, we have in general

$$P \sqsubseteq Q \quad \text{does not imply} \quad \mathbf{if} P \mathbf{fi} \sqsubseteq \mathbf{if} Q \mathbf{fi}.$$

This limits its use in program development.

8. POSITIVE APPLICATIONS OF MIRACLES

By Definitions 6 and 7, miracles refine only to other miracles, and hence by Dijkstra’s law never to programs. Thus if a specification *overall* is miraculous (we can check using Theorem 4), the development is doomed.

In VDM, where specifications are written like our predicate pairs, the check for miracles is the “implementability test” [11, p. 134]. In Z [7, 17, 20], where specifications are single predicates corresponding to our implicit form of Section 2.4, miracles cannot be written: Definition 4 and Theorem 4 show that single predicate specification statements are always feasible.

From a feasible beginning, miracles can arise through mistaken refinement tactics. As shown in Section 6, the “improper division” of $x := 0$ by $x := y$ gives the miraculous $x : [\mathit{true}, y = 0]$. If we recognize the miracle then, we could stop there and try some other tactic; if we do not, we are stuck later. But the *rules* for such division (the weakest prespecification of [10]) are simpler now that

soundness has been delegated to the unimplementable miracles: There is less need for “applicability conditions.”

There is other potential for the deliberate use of miracles. Consider the following assignment in which f is some function hard to calculate but easy to invert:

$$x := f(c). \quad (8)$$

And suppose in a variable y we might have the desired answer already. We can make the following refinements in which both right-hand sides are miracles:

$$x := f(c) \sqsubseteq c = f^{-1}(y) \rightarrow x := y \quad (9)$$

$$x := f(c) \sqsubseteq c \neq f^{-1}(y) \rightarrow x := f(c). \quad (10)$$

Neither (9) nor (10) can be implemented on its own. Case (9) can be executed only when y does contain the desired answer already; case (10) can be executed only when it does not. But their \parallel combination is *not* miraculous, and can always be executed:

$$(c = f^{-1}(y) \rightarrow x := y) \parallel (c \neq f^{-1}(y) \rightarrow x := f(c)). \quad (11)$$

Since $P \sqsubseteq Q$ and $P \sqsubseteq R$ implies $P \sqsubseteq Q \parallel R$ (easily shown from Definitions 6 and 9), we have refined (8) to (11). Such developments are also treated in [1] and [19].

Another application is as follows. Ordinarily we limit the syntax of our concrete programming language so that miracles cannot be written in it: No specifications can appear, nor can naked guarded commands. If we relax this restriction, allowing naked guarded commands, then operational reasoning suggests a *backtracking* implementation. For example, consider the following backtracking strategy for finding the position i of a value v in an array $a[0 \dots N - 1]$:

Choose i at random from the range $0 \dots N - 1$, and evaluate $a[i] = v$. If equality holds, then terminate; otherwise, backtrack and try again.

We have this refinement:

```

i:[a[i] = v]
 $\sqsubseteq$  if
    i := 0  $\parallel$  ...  $\parallel$  i := N - 1;
    a[i] = v  $\rightarrow$  skip
fi

```

We are using the generalized **if** ... **fi** of Section 6, which here allows abortion if its body is miraculous; and the body is miraculous *only* when no branch of the alternation can avoid the miraculous behavior to follow. In this context **if** ... **fi** resembles the “cut” of Prolog, allowing failure (preventing backtracking) if no solution is found within (beyond). If there is a successful branch, however, the implementation is obliged to find it: Only then can it execute the second statement, which we could syntactically sugar, writing **force** $a[i] = v$. Note that the first statement can be written $i: [0 \leq i < N]$.

A third opportunity for exploiting miracles is in novel proof rules. We introduce for a moment the weaker relation \leq between programs, which holds, if for all

predicates R ,

$$P\langle R \rangle \wedge Q\langle true \rangle \Rightarrow Q\langle R \rangle.$$

This is simply *partial* correctness. Now in the style of VDM we can consider a loop invariant to be a *statement*, rather than an assertion: Any number of iterations of the loop body must refine the invariant statement I . The advantage is that we have easy reference to the initial state; our development law is

If

$$I \leq I; G \rightarrow S$$

and

$$X \leq I; \mathbf{force} \neg G$$

then

$$X \leq I; \mathbf{do} G \rightarrow S \mathbf{od}$$

We “explain” this rule as follows (but it is *proved* using weakest precondition semantics). The first condition requires preservation of the effect of I by one more execution of the body $G \rightarrow S$. If G holds, the body behaves like S , but if G fails (and therefore we should *not* execute S), the first condition still holds because $G \rightarrow S$ in that case is miraculous, refining anything (and **skip** in particular).

Similar reasoning applies to the second condition. For the result, we argue informally that

$$\begin{aligned} X \leq I; \mathbf{force} \neg G \\ \leq \text{by induction over the first condition} \\ I; G \rightarrow S; \dots; G \rightarrow S; \mathbf{force} \neg G \\ \leq I; \mathbf{do} G \rightarrow S \mathbf{od} \end{aligned}$$

Take for example the following program in which we calculate the sum s of an array a indexed by $0 \leq i < N$.

$$\begin{aligned} X = s, n: [s = (\sum i: 0 \leq i < N: a[i])] \\ I = s, n: [s = (\sum i: 0 \leq i < n: a[i]) \wedge 0 \leq i \leq N] \\ G = n \neq N \\ S = s, n := s + a[n], n + 1 \end{aligned}$$

Because $I \sqsubseteq s, n := 0, 0$ (this is the initialization), and because we can prove the conditions hold (using definitions and Theorem 3), we have by our rule above

$$\begin{aligned} X \leq I; \mathbf{do} n \neq N \rightarrow s, i := s + a[n], n + 1 \mathbf{od} \\ \leq s, n := 0, 0; \\ \mathbf{do} n \neq N \rightarrow s, i := s + a[n], n + 1 \mathbf{od} \end{aligned}$$

9. CONCLUSION

We have extended Dijkstra’s programming language with a construct allowing abstract programs, as predicate pairs, to be written within otherwise conventional “concrete” programs. The advantages follow:

—Program development takes on the character of solving equations, which is well established in mathematics generally. The transformation from abstract to concrete occurs within a single semantic framework.

- As lambda expressions allow us to write functions without names (rather than the labored “*f* **where** $f(x) = \dots$ ”), we can write specifications directly, avoiding “*P* **where** $\dots \Rightarrow P(\dots)$.” Instead of a lambda calculus, this leads to a refinement calculus.
- We gain *miracles* as an artefact of our extension, and there is increasing evidence that they simplify the development process. In [16] it is shown that applicability conditions for refinement can be simplified, or even removed altogether, because mistaken development steps simply lead to miracles from which progress must eventually cease. Also in [1], [10], [18], and more recently [19], it is argued that miracles simplify the theory. In [14] it is shown that miracles allow proof of certain data refinements that were not provable previously.
- The lack of distinction between abstract and concrete programs allows their treatment as procedures to be made more uniform. As in Algol-60, a procedure call, whether abstract or not, is equivalent to its text substituted in line. This uniformity and the resulting treatment of parameters is explored in [15].
- The programmer’s repertoire is increased by providing easy access to nonconstructive idioms, for example: $i: [a[i] = v]$ finds the index i of value v in array a ; $m: [l \leq m \leq h]$ chooses m between l and h .
- A ready connection is made with state-based specifications such as those of Z [7, 17, 20], allowing their systematic development into code.

A refinement calculus would be a collection of *laws*, each proved directly from weakest precondition definitions. They could be used, without further proof, in program developments, just as one uses a table of integrals in engineering. For example, one such law is

Assignment Law:

$$w: [post[w \setminus E][v_0 \setminus v], post] \sqsubseteq w := E$$

It is easily proved from Definitions 3 and 6. A comprehensive collection of such laws is given and demonstrated in practice in [16].

Such a development style would be very close to VDM [11], in which specifications are predicate pairs just as here. But Jones does not base VDM on the weakest precondition calculus, nor does he present a general refinement relation operating uniformly between all programs whether abstract or concrete (although he could do so). Another difference of our method is our use of classical logic rather than the logic of partial functions [3, 11]. Jones does not treat miracles.

In the Z specification technique, specifications are given as single predicates corresponding to our “implicit preconditions.” Thus, where we write $n: [0 \leq n < n_0]$ for “decrease n , but not below 0,” in Z one would write (omitting types)

$$\left| \frac{n, n'}{0 \leq n' < n} \right.$$

In Z there is deliberately no commitment to a *fixed* state (our \vec{v}), because that flexibility is needed to build large specifications from their smaller components. Examples of large-scale Z specifications can be found in [7]. But when algorithmic

structures are introduced, that is, once *development* begins, this lack of commitment becomes a hindrance.

Therefore, one aim of our work is to provide a development method specifically for Z by identifying the two specifications above, then using the weakest precondition calculus to reach a concrete program. Another approach to Z development, derived from ours, is given in [12].

10. ACKNOWLEDGMENTS

Back [2] first embedded specifications within programs using the weakest precondition calculus. His specifications, like those of Z , consist of one predicate only, and he does not take advantage of miracles. More recently, Morris [18] presents independently the same extension of Back's work as we do; we have had useful discussions since discovering each other. Our refinement relation \sqsubseteq is the same as that of Back and of Morris.

Meertens [13] also has developed these ideas, using predicate pairs, but gave them a different meaning: (In our notation) he defines

$$[pre, post]\langle R \rangle \triangleq pre \Rightarrow (\exists \vec{v}. post) \\ \wedge (\forall \vec{v}. post \Rightarrow R)$$

But Meertens' definition does not have the property of Lemma 2, which we consider to be fundamental; in general, for Meertens

$$[pre, post]\langle post \rangle \neq pre.$$

Hehner [8] uses predicate pairs *for specifications* as we use specification statements, but he does not integrate the approach by giving them a weakest precondition semantics. He also uses the refinement ordering \sqsubseteq .

The earliest example of a formulation like ours for the weakest precondition of a specification seems to be Hoare's [9], in which it is given as the axiomatic meaning of procedure calls. But Hoare did not separate abstraction from procedure calling, as we have done (and discuss further in [15]). In [5, p. 153] the definition, again coupled to procedure calls, can also be found.

The idea of using pre- and post-conditions to describe program behavior is widespread, and its use in VDM is notable. In fact our approach is very close to VDM, and I hope identical in spirit. Jones does not, however, make his specifications "first-class citizens" as we do. An advantage of Jones's natural deduction style is perhaps its appeal to the wider audience of practicing programmers, just as natural deduction in logic is so-called because it is more "natural." But we prefer the increased freedom of the axiomatic approach directly (in logic, too): It offers more scope to the experienced user, who can construct new laws (meta-theorems) to suit his taste and skill.

Hoare [10] provided the direct inspiration for treating specifications as programs; he obtains similar results in the relational calculus. Miracles appear as partial relations but are not discussed in detail.

Most recently, Nelson [19] has integrated specifications and programs, but his ordering over these objects differs from ours. In particular, it does not allow the reduction of nondeterminism, an essential idea in program development. He discusses miracles at some length.

Much of this work was done in collaboration with Ken Robinson. I thank Rick Hehner, Joe Morris, Doaitse Swierstra, members of IFIP 2.1, and the referees for their very useful comments.

REFERENCES

1. ABRIAL, J.-R. Generalised substitutions. 26 Rue des Plantes, Paris 75014, France, 1987.
2. BACK, R.-J. On the correctness of refinement steps in program development. Tech. Rep. A-1978-4, Dept. of Computer Science, Univ. of Helsinki, Helsinki, 1978.
3. BARRINGER, H., CHENG, J. H., AND JONES, C. B. A logic covering undefinedness in program proofs. *Acta Inf.* 21 (1984), 251-269.
4. DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
5. GRIES, D. *The Science of Programming*. Springer-Verlag, New York, 1981.
6. GRIES, D., AND PRINS, J. A new notion of encapsulation. In *Proceedings of the SIGPLAN Symposium on Language Issues in Programming Environments* (Seattle, Wash., June 25-28, 1985). ACM, New York, 1985, pp. 131-139.
7. HAYES, I. J., ED. *Specification Case Studies*. Prentice-Hall, Englewood Cliffs, N.J., 1987.
8. HEHNER, E. C. R. *The Logic of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1984.
9. HOARE, C. A. R. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages. Notes in Mathematics 188*, E. Engeler, Ed., Springer-Verlag, New York, 1971.
10. HOARE, C. A. R., AND HE, J. The weakest prespecification. *Fundamenta Informaticae IX* (1986), 51-84.
11. JONES, C. B. *Systematic Software Development using V.D.M.* Prentice-Hall, Englewood Cliffs, N.J., 1986.
12. JOSEPHS, M. B. Formal methods for stepwise refinement in the Z specification language. Programming Research Group, Oxford Univ., 1986.
13. MEERTENS, L. Abstracto 84: The next generation. In *Proceedings of the ACM 1979 Annual Conference* (Detroit, Mich., Oct. 29-31, 1979). ACM, New York, 1979, pp. 33-39.
14. MORGAN, C. C. Data refinement using miracles. *Inf. Process. Lett.* 26, 5 (Jan. 1988), 243-246.
15. MORGAN, C. C. Procedures, parameters, and abstraction: Separate concerns. To appear in *Sci. Comput. Program.*
16. MORGAN, C. C., AND ROBINSON, K. A. Specification statements and refinement. *IBM J. Res. Dev.* 31, 5 (Sept. 1987), 546-555.
17. MORGAN, C. C., AND SUFRIN, B. A. Specification of the UNIX filing system. *IEEE Trans. Softw. Eng.* SE-10, 2 (Mar. 1984).
18. MORRIS, J. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.* 9, 3 (Dec. 1987), 298-306.
19. NELSON, G. A generalization of Dijkstra's calculus. Tech. Rep. 16, Digital Systems Research Center, Palo Alto, Calif., Apr. 1987.
20. SPIVEY, J. M. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3, CUP, Cambridge, U.K., 1988.
21. WIRTH, N. *Programming in Modula-2*. Springer-Verlag, New York, 1982.

Received May 1986; revised June 1987; accepted December 1987