Specification statements and refinement

by Carroll Morgan
Ken Robinson

We discuss the development of executable programs from state-based specifications written in the language of first-order predicate calculus. Notable examples of such specifications are those written using the techniques Z and VDM; but our interest is in the rigorous derivation of the algorithms from which they deliberately abstract. This is, of course, the role of a *development method*. Here we propose a development method based on *specification statements* with which specifications are embedded in programs—standing in for developments "yet to be done." We show that specification statements allow description, development, and execution to be carried out within a *single* language: programs/specifications become hybrid constructions in which both predicates and directly executable operations can appear. The use of a single language—embracing both high- and low-level constructs—has a very considerable influence on the development style, and it is that influence we discuss: the specification statement is described, its associated calculus of refinement is given, and the use of that calculus is illustrated.

1. Introduction

In the Z [1] and VDM [2] specification techniques, descriptions of external behavior are given by relating the "before" and "after" values of variables in a hypothetical program state. It is conventional to assume that the *external* aspects are treated by designating certain variables as containing initially the *input* values, and certain others as containing finally the *output* values. As development proceeds, structure is created in the program—and the specifications, at that stage more "abstract algorithms," come increasingly to refer to internal program variables as well. For example, we may at some stage wish to describe the operation of taking the square root of some integer variable n; by adopting the convention that n refers to the value of that variable *after* the operation, and n_0 to its value *before*, this description could be written

$$n^2 = n_0 \qquad (1)$$

Ordinarily, we would call the above a *specification*, because "conventional" computers do not execute (i.e., find a valuation making true) arbitrary formulas of predicate logic (logic programming languages deal only with a restricted language of predicates).

Two notable features of our specification (1) above are its *nondeterminism* and that it is *partial*. It is nondeterministic in the sense that for some initial values n_0 (e.g., 4) there may be several appropriate final values n (± 2 in this case). It is partial in the sense that for some initial values (e.g., 3) there are *no* appropriate final values. We see below that our proposed development method makes this precise in the usual way (e.g., that of [3]): the nondeterminism allows an implementation to return either result (whether consistently or varying from one execution to the next); and the implementor can *assume* that the initial value is a perfect square, providing a program whose behavior is wholly arbitrary otherwise.

In presenting a development technique, we are not ignorant of the fact that VDM already has (or even *is*) one; rather we are concentrating our attention on Z, where development has been less well worked out. In this our aim is most definitely to propose a *lightweight* technique—as Z is itself—in which existing material is used as much as possible. Dijkstra's language [3] therefore was chosen as the

target, because it has a mathematically attractive and above all simple semantic basis, and because it includes nondeterminism naturally.

The *key* to a smooth development process—the subject of this paper—is we believe the integration of description and execution in one language. This is not achieved, as is so often proposed, by restricting our language to those specifications which are executable, and thus treating specifications as programs; instead we extend the language to allow ourselves to write programs which we cannot see at first how to execute: in effect we treat programs as specifications. It is precisely the lack of semantic distinction between the two that allows finally our smooth transition from abstract description to executable algorithm.

We assume some familiarity with Dijkstra's weakest pre-condition concept and its associated guarded command programming language [3].

• *Weakest pre-conditions and specifications*

In [3], Dijkstra introduces for program P and predicate R over the program variables, the *weakest pre-condition* of R with respect to P; he writes it

$$wp(P, R)$$

This weakest pre-condition is intended to describe exactly those states from which execution of P is guaranteed to establish R, and Dijkstra goes on to develop a small language by defining for its every construct precise syntactic rules for writing $wp(P, R)$ as a predicate itself. For example, the meaning of *assignment* in this language is defined as follows for variable x, expression E, and post-condition R:

$$wp(\text{“}x := E\text{”}, R) = R[x\backslash E]$$

The notation $[x\backslash E]$ here denotes syntactic replacement in R of x by E in the usual way (avoiding variable capture, etc.). Thus

$$\begin{aligned} & wp(\text{“}x := x - 1\text{”}, x \geq 0) \\ = \ & (x \geq 0)[x\backslash x - 1] \\ = \ & (x - 1) \geq 0 \\ = \ & x > 0 \end{aligned} \tag{2}$$

We can *specify* a program P by giving *both* a pre-condition (not necessarily weakest) and a post-condition; our pre-condition and post-condition predicates we usually call *pre* and *post*:

$$pre \Rightarrow wp(P, post) \tag{3}$$

Informally, this is read "if *pre* is true, then execution of P must establish *post*"; formally, we regard the above as admitting only program texts P for which it is valid. Either way, it is a specification in the sense that it directs the implementor to develop a program with the required property.

Our point of divergence from the established style (3) is to write instead

$$[pre, post] \sqsubseteq P \tag{4}$$

We take (3) and (4) as identical in meaning, but in (4) the constituents are exposed more clearly: $[pre, post]$ is the specification; $\sqsubseteq$ is the relation of refinement; and P is the program to be found. Thus we read (4) as "the specification $[pre, post]$ is refined by P."

The principal advantage of the alternative style is that $[pre, post]$ can take on a meaning independent of its particular use in (4) above: we will give it a weakest pre-condition semantics of its own. It is just this which removes the distinction between specification and program—not that they both are executable, but that they both are *predicate transformers*, being suitable first arguments to $wp(\ ,\ )$. Programs are just those specifications which we can execute directly.

The refinement relation $\sqsubseteq$ is likewise generalized, and we do this immediately below.

• *Refinement*

In (4) we have introduced an explicit symbol $\sqsubseteq$ for refinement, and we now give its precise definition (as given, e.g., in [4]):

Definition 1 For programs P and Q, we say that P is refined by Q, written $P \sqsubseteq Q$, iff for all post-conditions *post*:

$$wp(P, post) \Rightarrow wp(Q, post). \qquad \square$$

We justify the above informally by noting that any occurrence of P in a (proved correct) program is justified by the truth of $wp(P, post)$ at that point, for some predicate *post*. No matter what *post* it is, the relation $P \sqsubseteq Q$ gives us $wp(Q, post)$ as well, so that Q is similarly justified: thus Q can replace P. Operationally, $P \sqsubseteq Q$ whenever Q resolves nondeterminism in P, or terminates when P might not.

This refinement relation is independent of the notion of specification, and can be evaluated for *any* two constructs whose weakest pre-condition semantics are known. For example, we have in the guarded command language of [3]

$$\begin{aligned} & \textbf{if } a \leq b \rightarrow a := a - b \\ & \ [\!] \ b \geq a \rightarrow b := b - a \\ & \textbf{fi} \\ \sqsubseteq \ & \textbf{if } a \leq b \rightarrow a := a - b \\ & \ [\!] \ a \not\leq b \rightarrow b := b - a \\ & \textbf{fi} \end{aligned}$$

The first program is nondeterministic, executing either branch when $a = b$; the second program is a proper (i.e., nonidentical) refinement of it because this nondeterminism has been removed. Such refinement relations between *programs* allow us to implement the nondeterministic program above in more conventional (deterministic) languages; we transcribe the deterministic refinement as follows:

```
IF a < = b THEN a := a − b
          ELSE b := b − a
END
```

• *Specification statements*

From the preceding section we can see that in formal terms we should have $[pre, post] \sqsubseteq P$ iff for all R

$$wp([pre, post], R) \Rightarrow wp(P, R) \tag{5}$$

But for this to have meaning, we must define its antecedent; as in the definition (2) above for assignment statements, we express $wp([pre, post], R)$ as a syntactic transformation of the predicate R. We do this below, moving from simple to more general cases.

• *The simple case*

In the simplest case we have two predicates *pre* and *post* each over the program variables in a single state. We have the following:

Definition 2 Let the vector of currently declared program variables be $\vec{v}$; for any predicates *pre*, *post*, and R, we define

$$wp([pre, post], R) = pre \wedge (\forall \vec{v} \,.\, post \Rightarrow R) \qquad \square$$

Note that our quantifiers always extend in scope to the first enclosing parentheses $(\forall . \cdots)$. As indicated, we use $\vec{v}$ to refer to the vector of all program variables, and do not concern ourselves very much with how they are declared.

Section 2 discusses the consistency of Definition 2 and Formula (5); here we justify the definition only informally. We regard $[pre, post]$ as a statement, and its first component *pre* describes the states in which its termination is guaranteed; thus *pre* is a necessary feature of our desired weakest pre-condition, and in fact appears as the first conjunct there. But the weakest pre-condition must guarantee more than termination: it must ensure that on termination, R holds. From the second component of $[pre, post]$, we know that *post* describes the states in which it terminates—and so we require only that in all states described by *post* the desired R holds as well: this is the second conjunct.

We now continue with some notational extensions and abbreviations.

Confining change

We allow a list of variables $\vec{w}$, in which appear all the variables which the statement can change; variables not in $\vec{w}$ must retain their initial values. The precise definition of $\vec{w}$: $[pre, post]$ is as follows:

Definition 3 Let the vector of currently declared program variables be $\vec{v}$, and let $\vec{w}$ be a subvector of $\vec{v}$; for any predicates *pre*, *post*, and R, we define

$$wp(\vec{w}\text{: } [pre, post], R) = pre \wedge (\forall \vec{w} \,.\, post \Rightarrow R) \qquad \square$$

The only change from Definition 2 is that the vector of quantified variables is now $\vec{w}$ rather than $\vec{v}$. Taking, for example, $\vec{v}$ to be "x, y," we have

$$\begin{aligned} & wp(x : [true, x = y], R) \\ = \; & true \wedge (\forall x \,.\, x = y \Rightarrow R) \\ = \; & R[x \backslash y]. \end{aligned}$$

Since also $wp(x := y, R) = R[x \backslash y]$, we have shown "$x : [true, x = y]$" and "$x := y$" to have the same meaning.

Initial values

So far, we can specify only that a certain relationship (e.g., *post*) is to hold between the *final* values of variables. We now adjust our definition so that 0-subscripted variables in the second component of a specification statement can be taken as referring to the *initial* values of variables.

Definition 4 Let the vector of currently declared program variables be $\vec{v}$, and let $\vec{w}$ be a subvector of $\vec{v}$; let *pre* and R as before be arbitrary predicates, and let *post* be a predicate referring optionally to 0-subscripted variables $\vec{v}_0$ as well. We define

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

—provided R contains no 0-subscripted variables $\vec{v}_0$. $\square$

By our definition we have *reserved* the use of 0-subscripts to denote initial values, and so must forgo their use for other purposes: this is why R should contain no $\vec{v}_0$. It is possible, however, to take the view that in R also the variables $\vec{v}_0$ refer to initial values; this leads in fact to the weakest *pre-specification* of Hoare and He [5]. Josephs [6] has investigated this.

We note that if *post* does not refer to initial values, then Definition 4 reduces to Definition 3.

The substitution $[\vec{v}_0 \backslash \vec{v}]$ may require renaming of the bound variables $\vec{w}$, but this is often unnecessary; for example, taking $\vec{v}$ to be "x, y" as before, we have

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

This is, of course, $wp(x := x + y, R)$, as one would hope.

Implicit pre-conditions

If the pre-condition is omitted, we supply a default condition for it as follows:

Definition 5 Let the vector of currently declared program variables be $\vec{v}$, and let $\vec{w}$ be a subvector of $\vec{v}$; let *post* be a predicate referring optionally to 0-subscripted variables $\vec{v}_0$. We define

$$\vec{w} : [post] \text{ abbreviates } \vec{w} : [(\exists \vec{w} \,.\, post)[\vec{v}_0 \backslash \vec{v}], post] \qquad \square$$

Thus the *implicit* pre-condition is simply "it is possible to establish the post-condition." This is exactly the view taken in Z specifications generally, where only a single predicate is given; in our original square-root example (1)—writing it n: $[n^2 = n_0]$—the implicit pre-condition is $(\exists n \,.\, n^2 = n_0)[n_0 \backslash n]$, which we can simplify to $(\exists k \,.\, k^2 = n)$. That is, termination is guaranteed only if n is a perfect square.

Generalized assignment

The assignment statement $x := E$ establishes the post-condition $x = E$ while changing only x—it has the same meaning, therefore, as the specification statement $x : [x = E[x \backslash x_0]]$ (in which the renaming $[x \backslash x_0]$ is necessary because occurrences of x in E are *initial* values). Exploiting this, we define below a *generalized* assignment statement in which the binary relation $=$ of ordinary assignment can be replaced by any binary relation desired.

Definition 6 If $\lhd$ is a binary relation symbol, then for any variable x and expression E,

$x :\lhd E$ abbreviates $x : [x \lhd E[x \backslash x_0]]$. □

Thus we have that

$x :< x$ decreases x; and that
$m :\in s$ chooses a member m from the set s.

Note that in the second case our implicit pre-condition is "the set s is not empty":

$$\begin{aligned} & m :\in s \\ = \; & m : [m \in s] \\ = \; & m : [(\exists m' \,.\, m' \in s), m \in s] \\ = \; & m : [s \neq \{\}, m \in s] \end{aligned}$$

This abbreviation was suggested by Jean-Raymond Abrial.

Invariants

Often a formula appears as a conjunct in both the pre- and the post-conditions, thus making it an *invariant* of the statement. The following convention, suggested in [4], allows us to write it only once; we abbreviate $[pre \wedge I, I \wedge post]$ by

$[pre, I, post]$

Thus $[pre, I, post] \sqsubseteq Q$ iff

$pre \wedge I \Rightarrow wp(Q, I \wedge post)$.

The above convention is useful when developing loops, as we see in Section 3.

2. The refinement theorems

The following theorems justify our choice of semantics for the specification statement. (Their full proofs may be found in [7].) The first theorem shows that for every specification there is a specification statement that satisfies it trivially.

Theorem 1 If $\vec{u}$ and $\vec{w}$ partition the vector $\vec{v}$ of program variables, then for any predicates *pre* and *post*

$pre \wedge \vec{v} = \vec{v}_0 \Rightarrow wp(\vec{w} : [pre, post], post \wedge \vec{u} = \vec{u}_0)$

Proof (outline) The result follows by straightforward application of Definition 4 and predicate calculus, except for the possible occurrences of 0-subscripted variables in $post \wedge \vec{u} = \vec{u}_0$. Since these are not *program variables* (we never declare, e.g., x_0 in a program), we can avoid the problem by a systematic renaming, proving instead that

$pre \wedge \vec{v} = \vec{v}_1 \Rightarrow wp(\vec{w} : [pre, post], post[\vec{v}_0 \backslash \vec{v}_1] \wedge \vec{u} = \vec{u}_1)$

This technique is used also in the proof of Theorem 3 in Section 5, given in full. □

The consistency mentioned in Section 1 follows easily from the above, taking $\vec{w} = \vec{v}$ and *post* free of $\vec{v}_0$; clearly other specializations are profitable as well.

The complementary problem is refining further a given specification statement; the following theorem shows how this can be done.

Theorem 2 If $\vec{w}$ and $\vec{u}$ partition the program variables $\vec{v}$, and if

$pre \wedge \vec{v} = \vec{v}_0 \Rightarrow wp(P, post \wedge \vec{u} = \vec{u}_0)$

then

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

Proof (outline) The proof again simply applies definitions, this time Definitions 1 and 4; the 0-subscripts are avoided as before. □

To summarize: Theorem 1 shows that $\vec{w} : [pre, post]$ is always a solution to the specification (of P):

$pre \wedge \vec{v} = \vec{v}_0 \Rightarrow wp(P, post \wedge \vec{u} = \vec{u}_0)$

Theorem 2 shows it to be *more general* than any other solution; thus overall we have that it is the most general solution.

3. The refinement calculus

We now move to our main concern. With the definitions of Section 1 we can mix specifications and executable constructs freely, and program development becomes a process of transformation within the one framework. But this is only the beginning—the definitions supply the "first principles" from which more specialized techniques spring, and we can use these derived *laws of refinement* directly in our development of programs. Each law is designed to introduce a particular feature into our final program, and the process overall comes to resemble the *natural deduction* style of formal proof, where our goals are not axioms but rather directly executable constructs (the Vienna Development Method [2] has a similar flavor).

We present the laws in the form

$$\frac{\textit{before-refinement}}{\textit{after-refinement}}\ \textit{side-condition}$$

and by this we mean: "if *side-condition* is universally valid, then *before-refinement* $\sqsubseteq$ *after-refinement*."

Often, there is no side-condition—this indicates that the stated refinement always obtains.

• *Strengthening the specification*

Generally speaking, refinement *strengthens* a specification, and it is characteristic of our refinement calculus that no check is made against strengthening a specification too much. The advantage of this is simplicity of the laws (Law 11 provides a striking example); a disadvantage is that unproductive refinement steps may go longer unnoticed. But there is no danger of invalidity resulting from over-strengthened specifications, for we will see that they can never provably be refined to executable code.

There is a simple *feasibility test* that can be applied to any specification, and its failure predicts the failure of the refinement process: we simply check that the specification satisfies Dijkstra's *Law of the Excluded Miracle* [3, p. 18] (paraphrased)

"For all executable programs P,
$wp(P, false) = false$"

If the specification failed this law, then so would any refinement of it; and since no *executable* program fails the law, we are forced to conclude that such a specification can never be refined to an executable program. For specifications, direct calculation yields that $\vec{w} : [pre, post]$ is feasible iff $pre \Rightarrow (\exists \vec{w} \,.\, post)[\vec{v}\backslash\vec{v}_0]$.

The essence of our advantage is therefore that our laws do *not* force us implicitly to apply a feasibility test at their every application: very often the correctness of a development step is obvious. Further discussion on this topic can be found in [7].

Our first two laws deal with weakening the pre-condition and/or strengthening the post-condition of a specification.

Law 1 Weakening the pre-condition; the new specification is more robust than the old (i.e., it terminates more often):

$$\frac{\vec{w} : [pre, post]}{\vec{w} : [pre', post]}\ pre \Rightarrow pre' \qquad \square$$

For example, $n : [n > 0, n = n_0 - 1]$
$\sqsubseteq n : [n \geq 0, n = n_0 - 1]$.

Law 2 Strengthening the post-condition; the new specification allows less choice than the old:

$$\frac{\vec{w} : [pre, post]}{\vec{w} : [pre, post']}\ pre \Rightarrow (\forall \vec{w} \,.\, post' \Rightarrow post)[\vec{v}_0\backslash\vec{v}] \qquad \square$$

For example, $n : [true, n \geq 0] \sqsubseteq n : [true, n > 0]$.

It is worth noting that a special case of Law 2 occurs when $\vec{v}$ and $\vec{w}$ are the same; then we have for the side-condition

$$pre \Rightarrow (\forall \vec{v} \,.\, post' \Rightarrow post)[\vec{v}_0\backslash\vec{v}]$$

Renaming $\vec{v}$ to $\vec{v}_0$ throughout, this is equivalent to

$$pre[\vec{v}\backslash\vec{v}_0] \Rightarrow (\forall \vec{v} \,.\, post' \Rightarrow post)[\vec{v}_0\backslash\vec{v}][\vec{v}\backslash\vec{v}_0]$$

which we may simplify to

$$pre[\vec{v}\backslash\vec{v}_0] \Rightarrow (\forall \vec{v} \,.\, post' \Rightarrow post)$$

The quantifier $\forall \vec{v}$ can be discarded since the antecedent contains no $\vec{v}$, and propositional calculus then gives us as our special case the appealing

$$pre[\vec{v}\backslash\vec{v}_0] \wedge post' \Rightarrow post$$

Law 3 Restricting change; the new specification can change fewer variables than the old:

$$\frac{\vec{w}, x : [pre, post]}{\vec{w} : [pre, post]} \qquad \square$$

For example, $x, y : [x = y_0] \sqsubseteq x : [x = y_0]$.

In Law 4 below, we use the compact symbols |[and]|, instead of the more conventional **begin** and **end,** to delimit the scope of local variable declarations.

Law 4 Introducing fresh local variables (where "fresh" means not otherwise occurring free):

$$\frac{\vec{w} : [pre, post]}{|[\textbf{var}\ x;\ \vec{w}\ x : [pre, post]\ ]|}\ x \text{ is a fresh variable} \qquad \square$$

For example, $f : [f = n!] \sqsubseteq |[\textbf{var}\ i; f, i : [f = n!]\ ]|$.

• *Introducing executable constructs*

The following laws allow us to introduce constructs from our target programming language.

Law 5 Introducing **abort**:

$$\frac{\vec{w} : [false, post]}{\textbf{abort}} \qquad \square$$

Since **abort** $\sqsubseteq P$ for any P, we can by transitivity of $\sqsubseteq$ have *any* program as the target of Law 5. Thus for any predicate *difficult*(n), we still have the easy refinement $n : [n < 0 \wedge n > 0, \textit{difficult}(n)] \sqsubseteq n := 17$.

Law 6 Introducing **skip**:

$$\frac{\vec{w} : [post[\vec{v}_0\backslash\vec{v}], post]}{\textbf{skip}} \qquad \square$$

For example, $x, y : [x = y, x = y_0] \sqsubseteq$ **skip**.

Law 7 Introducing assignment:

$$\frac{\vec{w} : [post[\vec{v}_0, \vec{w}\backslash\vec{v}, \vec{E}], post]}{\vec{w} := \vec{E}} \qquad \square$$

For example, $x : [true, x = x_0 + y] \sqsubseteq x := x + y$.

The next two laws are the *weakest pre-specification* and *weakest post-specification* constructions of Hoare and He [5], with which one can "divide" one specification A by another B, leaving a specification Q such that

$A \sqsubseteq Q; B$ (Law 8: weakest pre-specification)
$A \sqsubseteq B; Q$ (Law 9: weakest post-specification)

Law 8 Introducing sequential composition (weakest pre-specification):

$$\frac{\vec{w} : [pre, post]}{\vec{w} : [pre, wp(P, post)]; P} \quad \vec{w} : [true] \sqsubseteq P \qquad \square$$

The side condition $w : [true] \sqsubseteq P$ can be read "P changes only $\vec{w}$." For example, we have

$$x, y : [true, x = y + 1]$$
$$\sqsubseteq x, y : [true, x = 2];$$
$$y := 1$$

Law 9 Introducing sequential composition (weakest post-specification):

$$\frac{\vec{w} : [pre, post]}{\vec{w} : [pre, mid]; \vec{w} : [mid, post]} \quad mid, post \text{ contain no free } \vec{v}_0 \qquad \square$$

For example,

$$x : [true, x = y + 1]$$
$$\sqsubseteq x : [true, x = y];$$
$$x : [x = y, x = y + 1]$$

Law 9 can be generalized to the case in which variables $\vec{v}_0$ do appear (as shown in [8]); in that case, one has effectively supplied in *mid* the first component of the sequential composition. For our larger example to follow (Section 4), we need only the simpler version.

In Laws 10 and 11, we use a quantifier-like notation for generalized disjunction and alternation: If I for example were the set $\{1..n\}$, then $(\vee i: I .G_i)$ would abbreviate $G_1 \vee \cdots \vee G_n$, and **if** $(\| \; i .G_i \rightarrow S_i)$ **fi** would abbreviate

if $G_1 \rightarrow S_1$
$\| \cdots$
$\| \; G_n \rightarrow S_n$
fi

Law 10 Introducing alternation (**if**):

$$\frac{\vec{w} : [pre \wedge (\vee \; i: I .G_i), post]}{\textbf{if } (\| \; i: I .G_i \rightarrow \vec{w} : [pre \wedge G_i, post]) \textbf{ fi}} \qquad \square$$

The predicate *pre* is that part of the pre-condition irrelevant to the case distinction being made by the guards G_i: it is passed on to the branches of the alternation. For example, taking *pre* to be true, we have

$$y : [x = 0 \vee x = 1, x + y = 1]$$
$$\sqsubseteq \textbf{if } x = 0 \rightarrow y : [x = 0, x + y = 1]$$
$$\| \; x = 1 \rightarrow y : [x = 1, x + y = 1]$$
$$\textbf{fi}$$
$$\sqsubseteq \textbf{if } x = 0 \rightarrow y := 1$$
$$\| \; x = 1 \rightarrow y := 0$$
$$\textbf{fi}$$

Law 11 Introducing iteration (**do**):

$$\frac{\vec{w} : [true, inv, \neg(\vee i : I .G_i)]}{\textbf{do}} \qquad \square$$
$$(\| \; i : I .G_i \rightarrow \vec{w} : [G_i, inv, 0 \leq var < var_0])$$
$$\textbf{od}$$

The predicate *inv* is, of course, the loop invariant, and the expression *var* is the variant. We use var_0 to abbreviate $var[\vec{v} \backslash \vec{v}_0]$.

An example of Law 11 is given in Section 4; for now, we note that *inv* can be any predicate and *var* any integer-valued expression. Surprisingly, there are no side-conditions—a bad choice of *inv* or *var* or indeed G_i simply results in a loop body from which no executable program can be developed (see the remarks in Section 3).

Law 11 is proved in Section 5.

4. An example: square root

For an example, we take the square-root development of [3, pp. 61–65]; but our development here is deliberately terse, because we are illustrating not how to *find* such developments (properly the subject of a whole book), but rather how experienced programmers could *record* such a development.

• *Specification*

We are given a nonnegative integer *sq*; we must set the integer variable *rt* to the greatest integer not exceeding $\sqrt{sq}$, where the function $\sqrt{}$ takes the nonnegative square root of its argument.

• *Specification*

$$rt := \lfloor \sqrt{sq} \rfloor$$

$\lfloor x \rfloor$ —the "floor" of x—is the greatest integer not exceeding x.

• *Refinement*

We assume of course that $\sqrt{}$ is unavailable to us, and proceed as follows to eliminate it from our specification; we eliminate $\lfloor \; \rfloor$ also. "Stacked" predicates denote conjunction.

$$rt := \lfloor \sqrt{sq} \rfloor$$

$$= rt : [rt = \lfloor \sqrt{sq} \rfloor] \qquad \text{Definition 6}$$

$$= rt : [sq \geq 0, rt = \lfloor \sqrt{sq} \rfloor] \qquad \text{Definition 5}$$

$$= rt : [sq \geq 0, rt \leq \sqrt{sq} < rt + 1] \qquad \text{Definition of } \lfloor\ \rfloor$$

$$\sqsubseteq rt : \left[sq \geq 0, \quad \begin{matrix} 0 \leq rt \\ rt^2 \leq sq < (rt+1)^2 \end{matrix} \right] \qquad \text{Law 2}$$

• *Refinement*

Using Laws 4 and 2, we introduce a new variable *ru*, and strengthen the post-condition; our technique is to approach the result from above (*ru*) and below (*rt*):

$$\sqsubseteq \ |[\textbf{var } ru. \quad rt, ru : \left[sq \geq 0, \quad \begin{matrix} 0 \leq rt \\ rt^2 \leq sq < ru^2 \\ rt + 1 = ru \end{matrix} \right] \ ]|$$

We now work on the inner part.

• *Refinement*

Anticipating use of $rt + 1 \neq ru$ as a loop guard, we concentrate on the remainder of the post-condition, using Law 9 with

$$mid = \begin{pmatrix} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \end{pmatrix}$$

to proceed:

$$\sqsubseteq rt, ru : \left[sq \geq 0, \quad \begin{matrix} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \end{matrix} \right]; \quad rt, ru : \left[\begin{matrix} 0 \leq rt < ru, \\ rt^2 \leq sq < ru^2, \end{matrix} \quad \begin{matrix} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \\ rt + 1 = ru \end{matrix} \right] \qquad (6)$$

Using Laws 1 and 7, we can show that for the first component of the sequential composition above—establishing *mid*, to become the loop invariant—we have

$$\sqsubseteq rt, ru := 0, sq + 1$$

We now concentrate on the second component.

• *Refinement*

We now introduce the loop, rewriting the second component of the sequential composition (6) to bring it into the form required by Law 11; writing *inv* now for our *mid* above, we have

$$= rt, ru : [true, inv, rt + 1 = ru]$$

and then by Law 11, with variant $ru - rt$, we proceed

$$\sqsubseteq \textbf{do } rt + 1 \neq ru \rightarrow rt, ru : [rt + 1 \neq ru, inv, 0 \leq ru - rt < ru_0 - rt_0] \textbf{ od}$$

• *Refinement*

For the loop body, we use Law 4 again to introduce a local variable *rm* to "chop" the interval *rt..ru* in which the result lies:

$$\sqsubseteq \ |[\textbf{var } rm; \quad rt, ru, rm : [rt + 1 \neq ru, inv, 0 \leq ru - rt < ru_0 - rt_0] \ ]|$$

We first choose *rm* between *rt* and *ru*, using Law 9, then Law 3, twice to develop

$$\sqsubseteq rm : [rt + 1 \neq ru, inv, rt < rm < ru]; \quad rt, ru : [rt < rm < ru, inv, 0 \leq ru - rt < ru_0 - rt_0]$$

Then with Laws 1 and 7 we quickly dispose of the first component, deciding to make our choice of *rm* divide the interval evenly:

$$\sqsubseteq rm := (rt + ru) \div 2$$

We proceed with the second component.

• *Refinement*

The natural case analysis is now to consider $rm^2 \leq sq$ versus $rm^2 > sq$; accordingly, with Law 10, we so divide our task and immediately apply Law 3 to each case; we have

$$\sqsubseteq \textbf{if } rm^2 \leq sq \rightarrow rt : \left[\begin{matrix} rt < rm < ru, \\ rm^2 \leq sq, \end{matrix} \quad inv, 0 \leq ru - rt < ru_0 - rt_0 \right] \ [\!] \ rm^2 > sq \rightarrow ru : \left[\begin{matrix} rt < rm < ru, \\ rm^2 > sq, \end{matrix} \quad inv, 0 \leq ru - rt < ru_0 - rt_0 \right] \textbf{ fi}$$

For the first branch, we have by Law 7

$$\sqsubseteq rt := rm$$

For the second branch, we have similarly

$$\sqsubseteq ru := rm$$

This completes our development.

• *Consolidation: the implementation*

Developments in this style generate a tree structure in which children collectively refine their parents; to obtain the program "neat," we simply flatten the tree. For the square root program, the result is as follows:

```
|[var ru;
    rt, ru := 0, sq + 1;
    do rt + 1 ≠ ru →
        |[var rm;
            rm := (rt + ru) ÷ 2;
            if rm² ≤ sq → rt := rm
            [] rm² > sq → ru := rm
            fi
        ]|
    od
]|
```

It is to be stressed that this consolidated presentation is not to be carried off as the only relic of our development. The development itself must remain as a record of design steps taken and their justifications (and in industrial practice, of who took them!). Mistakes will still be made, and corrections applied; only when a complete record is kept can we make those corrections reliably, without introducing further errors—and learn from the process.

5. Derivation of laws

In this section we prove Laws 2 and 11 of Section 3. We do this for several reasons: to reassure the reader, who may doubt their validity; to demonstrate the use of the weakest pre-condition formula for specifications; and to suggest that the collection of laws can easily be extended by similar proofs.

• *Proof of Law 2*

Law 2 allows us to strengthen the post-condition of a specification; in simplest terms, this means replacing *post* by *post'* as long as we know that $post' \Rightarrow post$. The side-condition is weaker than this, however: It takes both the pre-condition and changing variables into account, making the law more widely applicable.

In the proof below, we assume that free-standing formulae are *closed*—that is, that their free variables are implicitly quantified (universally). It is this that allows us to rename variables when necessary.

Theorem 3 Proof of Law 2: if the following side-condition holds:

$$pre \Rightarrow (\forall \vec{w} \,.\, post' \Rightarrow post)[\vec{v}_0 \backslash \vec{v}]$$

then so does this refinement:

$$\vec{w} : [pre, post] \sqsubseteq \vec{w} : [pre, post']$$

Proof By Theorem 2, we need only show

$$pre \wedge \vec{v} = \vec{v}_0 \Rightarrow wp(\vec{w} : [pre, post'], post \wedge \vec{u} = \vec{u}_0)$$

Since in Definition 4 the predicate R must not contain $\vec{v}_0$, we rename those above to $\vec{v}_1$ (we may do this because the formula is closed); we must show

$$pre \wedge \vec{v} = \vec{v}_1 \Rightarrow wp(\vec{w} : [pre, post'], post \wedge \vec{u} = \vec{u}_1)$$

Definition 4 is now applied; we must show

$$pre \wedge \vec{v} = \vec{v}_1 \Rightarrow pre \wedge (\forall \vec{w} \,.\, post' \Rightarrow post \wedge \vec{u} = \vec{u}_1)[\vec{v}_0 \backslash \vec{v}]$$

Clearly we can remove the conjunct *pre* in the consequent, because it occurs in the antecedent; we can remove $\vec{u} = \vec{u}_1$ because $\vec{u}$ and the quantified $\vec{w}$ are disjoint, and $\vec{v} = \vec{v}_1$ appears in the antecedent. It remains to prove

$$pre \wedge \vec{v} = \vec{v}_1 \Rightarrow (\forall \vec{w} \,.\, post' \Rightarrow post)[\vec{v}_0 \backslash \vec{v}]$$

and this follows directly from the side-condition. □

• *Proof of Law 11*

We deal with the following restricted version of Law 11, in which we consider a single guard only and take $\vec{v}$ and $\vec{w}$ the same; we must show

$$\frac{[true,\ inv,\ \neg guard]}{\textbf{do}\ \ guard \rightarrow [guard,\ inv,\ 0 \le var < var_0]\ \ \textbf{od}}$$

Our proof is based on the loop semantics given in [2]; we show that for $k \ge 1$

$$inv \wedge (guard \Rightarrow var < k) \Rightarrow H_k(inv \wedge \neg guard) \qquad (7)$$

From this will follow

$$\begin{aligned} & inv \\ = \ & inv \wedge (guard \Rightarrow (\exists k \ge 1 \,.\, var < k)) \\ = \ & (\exists k \ge 1 \,.\, inv \wedge (guard \Rightarrow var < k)) \\ \Rightarrow \ & (\exists k \,.\, H_k(inv \wedge \neg guard)) \\ = \ & wp(\textbf{do} \cdots \textbf{od}, inv \wedge \neg guard) \end{aligned}$$

Thus by Theorem 1 we have as required that

$$[inv,\ inv \wedge \neg guard] \sqsubseteq \textbf{do} \cdots \textbf{od}$$

It remains therefore to prove (7), and this we do by induction over k. We note first that $H_0 = inv \wedge \neg guard$, and continue by direct calculation (writing *pre'* for $pre[\vec{v} \backslash \vec{v}']$, etc., and H_k for $H_k(inv \wedge \neg guard)$):

$$\begin{aligned} & H_1 \\ = \ & H_0 \vee \left(\begin{array}{c} guard \\ guard \wedge inv \\ (\forall \vec{v} \,.\, inv \wedge 0 \le var < var_0 \Rightarrow H_0)[\vec{v}_0 \backslash \vec{v}] \end{array} \right) \\ = \ & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ (\forall \vec{v}' \,.\, inv' \wedge 0 \le var' < var \Rightarrow H_0') \end{array} \right) \\ \Leftarrow \ & (\neg guard \wedge inv) \vee (guard \wedge inv \wedge var < 1) \\ = \ & inv \wedge (guard \Rightarrow var < 1) \end{aligned}$$

Our inductive step now concludes the argument:

$$\begin{aligned} & H_{k+1} \\ = \ & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ (\forall \vec{v}' \,.\, inv' \wedge 0 \le var' < var \Rightarrow H_k') \end{array} \right) \\ \Leftarrow \ & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ \left(\forall \vec{v}' \,.\, inv' \wedge 0 \le var' < var \Rightarrow \left(\begin{array}{c} inv' \\ guard' \Rightarrow var' < k \end{array} \right) \right) \end{array} \right) \\ \Leftarrow \ & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ var < (k+1) \end{array} \right) \\ = \ & inv \wedge (guard \Rightarrow var < (k+1)) \end{aligned}$$

□

The puzzling thing about Law 11 is that it has no side-condition, whereas one might expect to find the condition

$$guard \wedge inv \Rightarrow 0 \le var$$

But closer inspection reveals that whenever the above formula fails, the loop body is infeasible: it must terminate

(since *guard* $\wedge$ *inv* holds initially) and must establish $0 \leq var < 0$ (since $0 \not< var$ holds initially). By the law of the excluded miracle (see [3]), no executable program can do this—the refinement, though valid, is barren.

For the practicing developer, perhaps the side-condition should be explicit; indeed, Law 11 can be rewritten this way, with the $0 \leq var$ dropped from the post-condition of the loop body. For the historical record of our development, however, we want to prove the very minimum necessary—and feasibility is of no interest. There would be no program, and hence no record, if a feasibility check had failed.

6. Conclusion

We have claimed that the integration of specifications and executable programs improves the development process. In earlier work [7], the point was made that all the established techniques of refinement are of course still applicable; their being based on weakest pre-condition semantics automatically makes them suitable for *any* construct so given meaning. Indeed an immediate but modest application of this work is our writing, for example, "choose e from s" directly in our development language as "$e :\in s$."

The refinement calculus is a step further. We are not claiming that it makes algorithms easier to discover, although we hope that this will be so; but it clearly does make it easier to avoid trivial mistakes in development and to keep a record of the steps taken there. A professional approach to software development must record the development *process*, and it must do so with mathematical rigor. We propose the refinement calculus for that at least.

Another immediate possibility is the systematic treatment of Z "case studies" as exercises in development, and we hope to learn from this. (There are a large number of case studies collected in [1].) Such systematic development is already under way, for example, at the IBM Laboratories at Hursley Park, UK [9].

The techniques of *data* refinement, in which high-level data structures (sets, bags, functions . . .) are replaced with structures of the programming language (arrays, trees . . .), fit extremely well into this approach. Also facilitated is the introduction of procedures and functions into a development: the body of the procedure is simply a specification statement "yet to be refined," and the meaning of procedures can once more be given by the elegant *copy rule* of Algol-60. These ideas are explored in [8] and [10], and we hope to publish them more widely.

7. Acknowledgments

It is clear that our approach owes its direction to the steady pressure exerted by the work of Abrial, Back, Dijkstra, Hoare, and Jones. More direct inspiration came from the *weakest pre-specification* work of Hoare and He [5], who provide a relational model and a calculus for development; they strongly advocate the *calculation* of refinements as an alternative to refinements proposed, then proved. Robinson [10] has done earlier work on the refinement calculus specifically.

We believe the earliest embedding of specifications within Dijkstra's language of weakest pre-conditions to be that reported in Back's thesis [11], and to him we freely give the credit for it. His *descriptions* are single predicates, rather than the predicate pairs we use here, and he gives a very clear and comprehensive presentation of the resulting refinement calculus. Our work extends his in that we consider predicate *pairs*, and we do not require those pairs always to describe feasible specifications. Because of this, we obtain a significant simplification in the laws of our refinement calculus.

In [12] L. Meertens explores similar ideas, and we are grateful to him for making us aware of Back's work.

We have benefited from collaboration with the IBM Laboratory at Hursley Park; the joint project [9] aims to transfer research results directly from university to development teams in industry.

Morris [13] independently has taken a similar approach to ours (even to allowing infeasible *prescriptions*); we recommend his more abstract view, which complements our own.

To the referees, and to Stephen Powell of IBM, we are grateful for their helpful suggestions.

References

1. I. J. Hayes, *Specification Case Studies*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.
2. C. B. Jones, *Systematic-Software Development Using VDM*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.
3. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.
4. E. C. R. Hehner, *The Logic of Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.
5. C. A. R. Hoare and JiFeng He, "The Weakest Pre-Specification," *Fundamenta Informatica* **IX,** 51–84 (1986).
6. M. Josephs, "Formal Methods for Stepwise Refinement in the Z Specification Language," Programming Research Group, Oxford, UK.
7. Carroll Morgan, "The Specification Statement," submitted to *Trans. Programming Lang. & Syst.*
8. Carroll Morgan, *Software Engineering Course Notes*, Programming Research Group, Oxford, UK.
9. J. E. Nicholls (for IBM) and I. H. Sørensen (for Oxford), *Collaborative Project in Software Development.*
10. Ken Robinson, "From Specifications to Programs," Department of Computer Science, University of New South Wales, Kensington, Australia.
11. R.-J. Back, "On the Correctness of Refinement Steps in Program Development," *Report A-1978-4*, Department of Computer Science, University of Helsinki, Finland, 1978.
12. L. Meertens, "Abstracto 84: The Next Generation," *Proceedings of the 1979 Annual Conference, ACM.*
13. J. Morris, "A Theoretical Basis for Stepwise Refinement and the Programming Calculus," submitted to *Sci. Computer Programming.*

Received October 28, 1986; accepted for publication April 29, 1987

Carroll Morgan *Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, United Kingdom.* Dr. Morgan received a B.Sc. (Hons.) from the University of New South Wales, Kensington, and a Ph.D. from the University of Sydney, both in computer science. He worked for two years as a consultant in Sydney before returning to the University of Sydney as a part-time lecturer. In 1982 he joined the Programming Research Group at Oxford, working initially on their Distributed Computing Project. Since 1985 Dr. Morgan has been a university lecturer at Oxford and a Fellow of Pembroke College.

Ken Robinson *Department of Computer Science, University of New South Wales, P.O. Box 1, Kensington 2033, Australia.* Mr. Robinson received a B.Sc. and a B.E. from the University of Sydney, and has since been mathematics tutor, systems programmer, lecturer, and finally senior lecturer in computer science at the University of New South Wales. He spent the year 1985–1986 on sabbatical in Oxford.