FORMAL SPECIFICATION and VERIFICATION of SOFTWARE

Colin D. Walter & Stephen E. Eldridge

Department of Computation
UMIST
PO Box 88
Manchester M60 1QD, UK
www.co.umist.ac.uk

1 Importance of Formal Specification

The normal task of a software company is to construct programs and systems from an informal natural language description of what they should do, and to maintain them. Bridging the gap between the natural language description and the programming language code are scripts written in more or less formal specification languages (see also articles "Problem Domain System Analysis" and "Software Life Cycles"). Here we consider those specification languages that are formal. Their grammar, or syntax, and their meaning, or semantics, should therefore be precise and unambiguous.

Scripts in such languages are not usually executable, that is, they do not say *how* to do anything. Their aim is to capture exactly what the user wants so that the software engineer builds the right product, without relying entirely on a vague requirements description from the user. However, they contain little or no information as to how the necessary computations might be done. This is what the programmer adds to obtain the final code. There is always an initial dialogue between the customer and software engineer to determine what product is desired. Most of these discussions involve making the needs more precise.

Spending time on building the wrong product can be very costly especially if prototyping is not possible or the results cannot be seen until final assembly. It is therefore very important, cost-effective and time-saving to use formal specification languages at this early stage to determine all but the most trivial level of detail for what is required. Increasingly, both consumer and supplier are requiring formal specifications as part of their contract in order to make possible or defend against claims for damage in litigation.

1.1 Levels of Formality

One significant problem with specifications is that they appear unfriendly; it can be difficult to construct, read or understand them. This is normally overcome by having a hierarchy of levels of detail and formality, in which the initial informal description is moulded into the final formal specification over several intermediate stages, and decomposed into modules small enough to be fully comprehended. All those different levels are kept together as documentation for the code, and would be read in increasing order of formality and detail as introduction to aid comprehension of the end product. Thus, specifications are developed like software using normal engineering techniques such as stepwise refinement and top-down design. Indeed, both specification and software need to be produced simultaneously, the first documenting the second, for the specification to be of any practical use in maintenance or verification.

1.2 Levels of Cost

Another apparent problem with writing a formal specification is the cost. Although specification languages are no more complicated than programming languages, unfamiliarity with their apparently more abstract notions makes software engineers reluctant to tackle specification, and so experts may need to be used. Verification *is* more difficult, but that is a different subject, requiring the production of a formal specification first. There are many shortcuts avoiding expense. In particular, there is often no need to specify everything fully: the lowest level procedures might not require specifications distinct from their code, and many variables may not be sufficiently important to require precise definitions. Indeed, specification of any part of a piece of software is perhaps only justified when the product is going to be widely or frequently used, has safety-critical applications, malfunction has severe financial implications, or continuous maintenance is expected, i.e. wherever the need for correct code justifies the extra cost.

2 Basic Terminology

Specification has a number of purposes. In decreasing order of usage this includes construction, maintenance and verification. For each of these, the level of specification needed to document code is very similar. Initially one needs pre- and post- conditions written in predicate logic. If we call these P and Q respectively, and construct code C to meet this specification, then we may write

$$\{P\}\ C\ \{Q\}$$

where { } are the comment brackets for the programming language. This means that if the initial data satisfies P and C is executed successfully, then the final data satisfies Q. In this case we say C **satisfies** the given specification or that it is **partially correct**. There is no claim that the code will terminate or will do so without raising an exception, but if it always does for initial data satisfying the pre-condition, then it is said to be **totally correct**.

At any point within some code we can insert a predicate formula, usually written within comment brackets. Thus, the property P between sections of code C and D in

$$C\ \{P\}\ D$$

is called an **assertion**. It is something that is expected to be true of the data whenever control passes that point during execution. Within certain restrictions, such as avoiding unbounded quantification, it might be possible during run-time to evaluate P. This can be especially valuable when debugging because it can give much more information than strong typing, and can enable errors to be discovered much closer to their origin.

3 Programming Language Semantics

3.1 Assignment

Specifications should enable us to write correct code, but this is only possible if we know what the various constructs in the programming language are supposed to do – they need specifying. Imperative programming is based on what the assignment statement does, namely,

$$\{Q(x/t)\}\quad x := t\quad \{Q\} \qquad \textit{(assignment)}$$

(The notation of the pre-condition is explained in the next paragraph.) In terms of manipulating numbers in a machine, this code states that the value of the expression t is to be assigned to the variable x. However, in terms of the properties of the data held in the machine, its specification declares that in order to obtain property Q for the data immediately after its execution we need the data to satisfy property $Q(x/t)$ immediately before its execution.

The precise meaning of $Q(x/t)$ in the above axiomatic definition of assignment is not important here. Roughly speaking, it is a slightly altered version of the formula Q in which every so-called "free" occurrence of x has been replaced by t. An occurrence of x is **bound** if it is in a subformula of Q with the form $\forall x Q_1$ or $\exists x Q_1$, where $\forall$ and $\exists$ are the usual quantifiers "for all" and "there exists". Occurrences which are not bound by a quantifier are **free**. However, the formation of $Q(x/t)$ may also involve renaming some bound variables in order to prevent confusion between occurrences of variables introduced into Q by the substitution and those already associated to the quantifiers appearing in Q.

The definition of assignment is an axiom scheme in the proof system we would use to verify any imperative code. From it we can deduce the partial correctness of, for example,

$$\{True\}\quad x := 2\quad \{x = 2\}.$$

This statement claims that any input satisfying the empty condition $True$ (i.e. no restriction on the input) will, after execution of $x := 2$, produce data satisfying $x = 2$. This is certainly what we would expect the code to do. To prove it, the assignment rule is applied. The pre-condition which yields the desired post-condition $x = 2$ is obtained by replacing free occurrences of x by 2. This gives $2 = 2$, which is equivalent to $True$. So the specified code $\{True\}\ x := 2\ \{x = 2\}$ is partially correct. A proof of total correctness is straightforward here because the expression 2 on the right side can certainly be computed on any machine without run-time errors or non-termination, and assignment of the result to memory should not cause problems.

3.2 Verification Conditions

The equivalence of $2 = 2$ to $True$ really just needed a proof of the implication $True \to 2 = 2$. In general, our specification provides a pre-condition P to the code, but partial verification, as above, generates a **weakest** pre-condition P'. In effect P' describes the widest set of input for which the code will produce output with the desired properties. So we are left with a formula $P \to P'$ which needs to be proved. Thus, the need to prove pure predicate formulae arises naturally in program verification, and is the point where theorem provers are required.

3.3 Sequencing and Branching

The other basic components of programming languages are sequencing, branching and looping. Each is constructed from smaller sections of code. If these subsections are known to satisfy particular specifications, then a specification for the whole construct can be deduced using an **inference rule**. For example, writing the list of hypotheses above a line, and the consequence below, we have the sequencing rule

$$\frac{\{P\}\ C\ \{Q\}, \quad \{Q\}\ D\ \{R\}}{\{P\}\ C\ ;\ D\ \{R\}} \qquad \textit{(sequencing)}$$

The two hypotheses above the line say we start with the assumptions (i) that if computation starts with data satisfying P and executes C successfully then Q will hold, and (ii) that if computation starts with data satisfying Q and executes D successfully then R will hold. Clearly, if we start with data satisfying P, and then successively and successfully execute first C, so that Q holds, and then D, then the output data will satisfy R. This is what is claimed by the conclusion given below the line.

Branching can be defined axiomatically in a similar fashion:

$$\frac{\{P\}\ C\ \{Q\}, \quad \{R\}\ D\ \{Q\}}{\{(P\&B)\ \mathit{or}\ (R\ \&\ \mathit{not}\ B)\}\ \ if\ B\ then\ C\ else\ D\ \ \{Q\}} \qquad \textit{(branching)}$$

In this case, the output from executing C or D is assumed to be Q, given input data with properties P and R respectively. The conclusion of the rule is that $(P\&B)$ *or* $(R\ \&\ \mathit{not}\ B)$ is the pre-condition required for the conditional statement to give output satisfying Q. This formula involves the logical operators &, *or* and *not* which are identical to the Boolean operators found in the programming language itself. The Boolean expression B is, of course, a respectable formula of predicate calculus. As observed in the assignment example with $True \rightarrow 2 = 2$ the apparently complicated pre-condition can often be simplified substantially.

3.4 Repetition

Repetition is the most interesting construct as far as verification is concerned. Its axiomatic specification is this:

$$\frac{\{P\}\ C\ \{P\}}{\{P\}\quad while\ B\ do\ C\quad \{P\ \&\ not\ B\}} \qquad \text{(looping)}$$

It is really not at all obvious that this rule is the slightest bit helpful! Above the line, the hypothesis about what C does seems to say very little: execution of C does not do anything to the property P of the data. Worse still, the loop itself seems to do little more: the output data fails to satisfy B, but at least still satisfies P, as it did before. The key to seeing the usefulness of the rule is that *not* B provides information that allows P & *not* B to release the properties we want. This is illustrated in the procedure *Add* below.

However, the major difficulty of this rule in verification is that knowledge of the formula P & *not* B does not enable the formula P to be deduced. The problem is that the formula P & *not* B is not provided *syntactically* separated conveniently into components P and B, but as a *semantically* equivalent formula. Thus, $(P'$ & *not* $B)$ & *not* B is semantically equivalent to P' & *not* B, but syntactically we could extract either P' & *not* B or P' as the formula P. We cannot know what formula to pick for P without help from the programmer. The formula P is called a **loop invariant**, and in it the programmer needs to state the cumulative properties he intends the loop to have achieved at the end of a typical iteration.

Further constructs can be specified in a similar manner to obtain the full axiomatic semantics of the language. Procedure and function calls are complicated by such things as various parameter passing mechanisms, side effects, lazy or eager evaluation, and order of evaluation of expressions and parameters. This clearly requires a deeper treatment than can be given here. For this, the interested reader is referred to [2] or [3].

4 Programming Example

4.1 Informal Description & Code

Let us look at an example in Pascal, starting with a brief semi-formal description of the notation and an idea of the specification. This forms the bridge between the formal specification, which is presented by annotating the code, and an informal requirements description. It should explain both the code and its specification.

The main data structure of the program segment given below is $Register =$ Array$[Index]$ of Bit. It is used to hold the bits of numbers written in binary notation. So the bit element $A[i]$ of the array $A : Register$ is the coefficient of 2^i in the number held in A. Hence the value of the number in A is $\sum_{i=0}^{MaxIndex} A[i]\times 2^i$, which will also be written as A.

Part way through the hand calculation of the sum of two numbers, the sum of the numbers represented by the first few digits has been found. The procedure *Add* here mimics this, and so it is useful to define A_I to be the value of the number in the first $I+1$ bits, those with indices from 0 up to I. So we take $A_I = \sum_{i=0}^{I} A[i]\times 2^i$. The number represented by taking no bits at all is 0, and so we have $A_{-1} = 0$. In this notation the value of A is $A_{MaxIndex}$ since the array has a maximum index value of $MaxIndex$. The largest $Register$ value possible is given by an array in which every bit is 1. If we call this $MaxReg$, then $MaxReg[i] = 1$ for each $i : Index$. It represents the number $MaxReg = 2^{MaxIndex+1} - 1$.

The procedure below is to output an overflow condition when the sum of the inputs is greater than $MaxReg$ and otherwise output the sum $In1+In2$ of the inputs. The addition is done bit-wise as in a simple hardware adder or as in a hand calculation, with the same definition of carries. (Here the latter analogy is slightly more appropriate as the bit values are expressed in terms of mathematical functions rather than logic gates.)

```
Const MaxIndex    = 31 ;
Type  Index       = 0..MaxIndex ;
      IndexPlus1  = 0..MaxIndex+1 ;
      Bit         = 0..1 ;
      Register    = Array[Index] of Bit ;
Procedure Add(In1,In2 : Register ;
              Var SumOut : Register; Var Overflow : Bit) ;
{ Write only: SumOut, Overflow }
{ Pre-Add:  True }
{ Post-Add: (In1+In2 ≤ MaxReg ↔ Overflow = 0
                              ↔ SumOut = In1+In2 }
Var I     : IndexPlus1 ;
    Carry : Bit ;
Begin { Add }
    I     := 0 ;
    Carry := 0 ;
    While I <= MaxIndex do
    Begin
        SumOut[I] := (Carry + In1[I] + In2[I]) mod 2 ;
        Carry     := (Carry + In1[I] + In2[I]) div 2 ;
        I         := Succ(I)
        { ($In1_{I-1}$+ $In2_{I-1}$ = $SumOut_{I-1}$+ Carry*$2^I$) &
                                   (0 ≤ I ≤ MaxIndex+1) }
    End ;
    Overflow := Carry
    End ; { Add }
```

4.2 The Specification

As in this example, code should contain in-line the formal specification. This includes, first of all, against each type declaration, **data invariants** which are properties expected to hold for all variables of that type. There are no restrictions for the types used here, but examples of this are given below where we describe some specification languages. Next, pre- and post-conditions for the procedure need to be given in the procedure heading. These are of the kind described at the beginning of this article, and refer to the functional properties of the body of the procedure.

Also in the heading there should be information about any use made of variables global to the procedure. Lists of those variables whose values are accessed or updated must be provided. This enables one to deduce the following. If P is a property which holds before a call to the procedure and P contains no free occurrences of any global variables which are updated, then P will still hold after execution of the call. In other words, property P will hold after the procedure call if it held beforehand and none of its free variables has had its value changed. The consequence of including these lists is that pre- and post-conditions for procedures can be made simpler because they do not need to include such properties. Indeed, the post-condition need only describe what changes have been made to variables which are updated, that is, those in the *write* lists.

With the detail in the heading fixed, the programmer can complete the code. This construction demands that he or she decides how the addition is to be done, and, in particular, what needs to have been achieved at the end of each iteration of the loop. This is inserted in the code as an assertion, which, in this specific instance is called a **loop invariant** and is the property named P which we use when applying the looping inference rule above to verify the code. It contains algorithmic information which a program verifier cannot be expected to deduce.

In the procedure *Add*, there is one loop invariant, namely,

$$(In1_{I-1} + In2_{I-1} = SumOut_{I-1} + Carry * 2^I) \;\; \& \;\; (0 \leq I \leq MaxIndex{+}1)$$

The loop inference rule states that at the end of the loop this property holds together with *not B* where B is the Boolean condition in the loop. Since *not B* is $I > MaxIndex$ and the loop invariant gives $I \leq MaxIndex{+}1$, it is easy to deduce $I = MaxIndex{+}1$. Substituting this value into the loop invariant yields $In1_{MaxIndex} + In2_{MaxIndex} = SumOut_{MaxIndex} + Carry * 2^{MaxIndex+1}$ at the end of the loop, that is,

$$In1 + In2 = SumOut + Carry * 2^{MaxIndex+1}$$

This illustrates how the rule for loops really does produce something useful.

It is the pre- and post- conditions and loop invariants which are essential to enable automatic program verification, for human understanding of the code, and for maintenance purposes. However, the informal specification was also useful, making it easier to understand the formal one which documents the code, and being fairly important to the understanding of the code.

4.3 Partial Verification

A program verification tool starts with the post-condition, applies the inference rules and axioms as above which define program constructs, and makes use of loop invariants to deduce the weakest pre-condition, say Q, which the initial data has to satisfy. This may not match the pre-condition P supplied in the specification. Clearly, to complete the proof we need $P \to Q$ to hold initially. This is called a **verification condition**, and is a pure predicate calculus formula. We had the example $2 = 2 \to True$ above. Verification conditions arise in particular at points where assertions have to be supplied. Consider loops as an example. To prove the loop against its specification, the verifier must first prove the hypothesis in the looping rule and then apply that rule to conclude the loop is correct. In proving this hypothesis, the supplied loop invariant P is used as the post-condition on the loop body, and the verifier deduces the weakest pre-condition Q. This may not coincide with the pre-condition, also P, required by the inference rule for loops. In such a case, $P \to Q$ would need to be proved in order to show that input satisfying P will indeed satisfy the pre-condition Q.

4.4 Termination

Total verification of the example above requires a proof that everything terminates properly. Assuming that the range of implemented integers includes the values of $MaxIndex{+}1$ and 3, it is fairly straightforward to check that everything respects the type restrictions, including all intermediate calculations. So the only possible source of improper termination would be if the **while** loop were infinite. Normally, to prove termination of loops we need to exhibit a function of the data with certain properties in respect of its values at the end of each iteration. The function needs to reach an acceptable value in a finite number of steps. With real number computing, this function might be an estimate of error, which we must show tends to zero so that it is eventually small enough. In discrete computing, as here, the function is often a monotonically decreasing natural number valued function. Thus, in the example, the function $MaxIndex{+}1{-}I$ decreases strictly on each iteration, is initially positive, and is always at least 0 (by the type constraints, which ought to have been checked). As the number of values that the function can have is at most one more than its initial value, there are at most that number of iterations of the loop: a finite number. So the code terminates properly.

Further detail about specification of languages and programs and their verification is to be found in references [1], [2], [3] and [6].

5 Specification Languages

There is much more to verification than the total correctness considered so far. Two specification languages, Z and VDM, provide notation which makes easy a systematic treatment of further aspects.

Specifications of operations in VDM, and in the similar specification method Z, make use of a **state** which, in terms of Pascal, may be thought of as the set of values of the global variables which our operations or procedures may use. This is often expressed by saying that VDM and Z are **model-oriented** approaches, meaning that their specifications define operations by their effect on external variables from a state. This requires

1. definition of the set of states
2. definition of the initial state(s)
3. specification of implementable operations whose external variables are parts of the state.

We shall give an example, which will also serve to show what kind of notation is used. It concerns the storage manager of an operating system. The manager must associate each available block of storage with its user.

5.1 Example in VDM: The Specification

Let B be the set of storage blocks and U be the set of users. The association of blocks with users is specified by a partial function from B to U. The function is partial because some blocks may not be used. It helps to keep explicit track of the unused, or free, blocks. Hence we are led to consider a state whose external variables are the partial function $dir : B \to U$ and the set $free \subseteq B$.

First, we shall show how to make use of this state using the notation of VDM. The types of the variables in the state are written

$dir : map\ B\ to\ U$ and
$free : set\ of\ B.$

The free blocks are precisely those which are not in the domain of definition of dir. Hence we have the **data invariant**:

$$free = B - dom(dir).$$

We could define a record, or composite, type to store this information if we liked. The VDM notation for such a type is

$$\begin{aligned} SM :: dir &: map\ B\ to\ U \\ free &: set\ of\ B \end{aligned}$$

or

$$\begin{array}{ll} SM \ \underline{\triangle} & \mathit{compose}\ SM\ \mathit{of} \\ & \quad \mathit{dir} : \mathit{map}\ B\ \mathit{to}\ U \\ & \quad \mathit{free} : \mathit{set}\ \mathit{of}\ B \\ & \mathit{end} \end{array}$$

where each value, $sm : SM$, must satisfy the data invariant

$$\mathit{inv}{-}SM(sm) \ \underline{\triangle}\ \mathit{free}(sm) = B - \mathit{dom}(\mathit{dir}(sm)).$$

(The symbol $\underline{\triangle}$ is shorthand for "is defined by".) The initial state, with no blocks allocated, has $\mathit{dir} = \emptyset$ (the empty map) and $\mathit{free} = B$.

Consider the operation, $REQUEST$, which finds an unused block b for a user u (and updates dir and free appropriately). We specify it by using a heading, rather like a function head in Pascal, which shows the names and types of the inputs and outputs. We then list the external variables from the state which the operation uses. These are marked rd if they are read only or wr if they may also be written to or changed. Finally we write a pre-condition which must be satisfied by the inputs and state values before the operation is done and a post-condition which must be satisfied by the outputs and state variables after the operation is done. The post-condition is likely to have to refer to the values of the inputs and values of state variable before the operation is done. To distinguish values of state variables before and after the operation we decorate the previous values with a hook. This decoration is only necessary in post-conditions since pre-conditions can only refer to initial state values. The specification of $REQUEST$ may be written in this style as follows.

$$\begin{array}{l} REQUEST(u : U)\ b : B \\ \mathit{ext}\ \mathit{wr}\ \mathit{dir} : \mathit{map}\ B\ \mathit{to}\ U \\ \mathit{ext}\ \mathit{wr}\ \mathit{free} : \mathit{set}\ \mathit{of}\ B \\ \mathit{pre}\ \ \mathit{free} \neq \emptyset \\ \mathit{post}\ b \in \overleftharpoon{\mathit{free}} \wedge \mathit{free} = \overleftharpoon{\mathit{free}} - \{b\} \\ \qquad\qquad \wedge\ \mathit{dir} = \overleftharpoon{\mathit{dir}} \dagger \{b \mapsto u\} \end{array}$$

In this specification we implicitly assume that dir and free satisfy the data-invariant: $\mathit{free} = B - \mathit{dom}(\mathit{dir})$. $\dagger$ is the override operator which here gives precedence to the new association of b with u rather than any previous association given by dir. In this case we could equally well have used $\cup$ but it would then not have been so clear that our new value for dir is still well-defined.

5.2 Example in VDM: Proof Obligations

Such a specification immediately gives rise to a proof obligation. We must prove that the operation is **implementable**. This does not usually mean writing a computer program which satisfies the specification but one should show that, given a state and input satisfying the pre-condition, there is a state and output satisfying the post-condition. In particular one must show that the resulting state does not contain variables which fail to satisfy the appropriate data invariants. In our example this entails showing that $free = B - dom(dir)$ still holds after the operation $REQUEST$ is performed.

We usually start with an implicit specification which is very abstract and does not say how to implement the operation, merely what it should do. This has great advantages. Such a specification is likely to be more concise than an explicit definition which contains implementation detail. It also is more adaptable, leaving us free to change the actual types and algorithms used in an implementation without having to rewrite our specification from scratch. Nevertheless we shall have to make our specification more concrete in order to make sure that an implementation really does satisfy the specification. This process, which usually proceeds in several steps, is called **data-reification**. In our example we would probably not be able to use sets to implement $free$ or functions in order to implement *dir* but might have to use some sort of list and list of pairs respectively. We first define a new type for storing information about the storage manager and then rewrite the specifications of our operations, $REQUEST$ etc., to suit this new type. In order to show that this reification step has worked properly we must discharge several more proof obligations.

First we must show that our new type contains at least one representative for each member of our previous storage manager type SM. If we can show this than our new type is called an **adequate** representation of SM.

Then we must prove various properties of our new operations. First they must be shown to be implementable. Then we must show that our new operations correspond to the old ones. A proof of such a property is called an **implementation modelling proof**.

Without going into detail about what an implementation modelling proof entails, we should note that it is considerably harder if the original type contains two different elements which are not distinguishable by any finite sequence of its operations. This undesirable property is called **implementation bias**. In our example SM is unbiased but our new type is likely to be biased since there are several lists with the same elements as any given set *free* with more than one element, and we do not wish to distinguish between them. The absence of bias in SM allows us to define the correspondence between the new type and SM by means of a function from the new type to SM. This function is called a **retrieve function**. The presence of a retrieve function makes implementation modelling proofs simpler and also gives us a simple criterion for adequacy, namely that the retrieve function should map the new type *onto* SM.

5.3 The Same Example in Z: The Specification

The specification language Z is a variant of VDM notation. It uses a generalisation of set comprehension notation, $\{x \in T | P(x)\}$, called the **schema**. A schema S has form $S \underline{\triangle} [declarations | predicate]$ and is usually written vertically.

$$\begin{array}{|l|}\hline S \\ \quad \mathit{declarations} \\ \hline \quad \mathit{predicate} \\ \hline\end{array}$$

Such a schema may be used to define a composite type by putting the fields of the type in the declaration part and the data-invariant in the predicate part. Thus our storage manager type SM can be defined by:

$$\begin{array}{|l|}\hline S \\ dir : B \rightarrow U \\ free : \mathbb{P}(B) \\ \hline free = B \setminus dom(dir) \\ \hline\end{array}$$

The notation used by Z is often more like standard mathematical notation than is the notation of VDM. For example the type of *dir* is written $B \rightarrow U$ (sometimes with a line through the arrow to stress that we are using partial functions) instead of *map B to U*, and the power set of B (i.e. the set of its subsets) is written $\mathbb{P}(B)$ instead of *set of B*.

Schemas are more versatile than the example above suggests; they may be used not only for defining composite types but also for specifying operations. The following decoration conventions are used:

Decoration with ! denotes an input to an operation.
Decoration with ? denotes an output from an operation.
Decoration with $'$ denotes a state after variable.

For example, if s is the value before an operation then s' is the value afterwards. Both s and s' must be declared in a schema defining an operation involving s because the predicate part of the schema must show how s is changed by the operation (even if there is no change and $s = s'$).

The schema specifying our operation $REQUEST$ is:

$$\begin{array}{|l}
\hline
\textit{REQUEST} \\
dir,\ dir' : B \to U \\
free,\ free' : \mathbb{P}(B) \\
b! : B \\
u? : U \\
\hline
free = B - dom(dir)\ \wedge \\
free' = B - dom(dir')\ \wedge \\
free \neq \emptyset\ \wedge \\
b! \in free\ \wedge \\
free' = free - \{b!\}\ \wedge \\
dir' = dir \oplus \{b! \to u?\} \\
\hline
\end{array}$$

Note that the override operator is now written $\oplus$. The predicate part of this schema could be simplified. For example it follows from $b! \in free$ that $free \neq \emptyset$.

There is a rich schema calculus for combining schemas and making specifications look acceptably concise. For example, a schema may have other schemas in its declaration part. The convention is that if S has schema T in its declaration part then we may expand S by merging the declarations of T with those explicitly present in S and *and*ing the predicate part of T with the explicit predicate part of S.

Hence the following definitions:

$$\begin{array}{|l}
\hline
SM' \\
dir' : B \to U \\
free' : \mathbb{P}(B) \\
\hline
free' = B - dom(dir') \\
\hline
\end{array}$$

and

$$\begin{array}{|l}
\hline
\Delta SM \\
SM \\
SM' \\
\hline
\end{array}$$

i.e.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\Delta SM} \\
dir,\ dir' : B \rightarrow U \\
free,\ free' : \mathbb{P}(B) \\
\hline
free = B - dom(dir)\ \wedge \\
free' = B - dom(dir') \\
\hline
\end{array}$$

permit the following, more concise, specification of *REQUEST*:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{REQUEST} \\
\Delta SM \\
b! : B \\
u? : U \\
\hline
free \neq \emptyset\ \wedge \\
b! \in free\ \wedge \\
free' = free - \{b!\}\ \wedge \\
dir' = dir \oplus \{b! \rightarrow u?\} \\
\hline
\end{array}$$

Readers interested in pursuing the specification languages VDM and Z further will find very readable accounts in references [4], [5] and [7].

References

1. E.W. Dijkstra, C.S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.
2. R. Dowsing, V. Rayward-Smith, C.D. Walter, *A First Course in Formal Logic and its Applications in Computer Science*, Blackwell Scientific, 1986, ISBN 0-632-01308-7.
3. D. Gries, *The Science of Programming*, Springer-Verlag, 1981, ISBN 0-387-90641-X.
4. D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification*, Oxford University Press, 1988, ISBN 0-19-859664-2.
5. C.B. Jones, *Systematic Software Development using VDM*, (2nd Edition) Prentice/Hall International, 1990, ISBN 0-13-880733-7.
6. A. Kaldewaij, *Programming*, Prentice/Hall International, 1990, ISBN 0-13-204108-1.
7. M. Spivey, *The Z Notation – A Reference Manual*, Prentice Hall, 1989.