KEY CENTRE FOR SOFTWARE TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF QUEENSLAND

St. Lucia
Queensland
Australia 4072

TECHNICAL REPORT

No. 148

# Specifications are not (necessarily) executable

I. J. Hayes and C. B. Jones

January, 1990

# Specifications are not (necessarily) executable[*]

I. J. Hayes[†], C. B. Jones[‡]

**Abstract**

Specifications can be written in languages which have formal semantics. Their very formality, and the similarities with some aspects of implementation languages, invites the idea that specifications might be executed. This paper presents a number of arguments against that idea. The aim is to warn of the dangers of limiting specification languages to the point where all of their constructs can be executed. While conceding the difficulties of relating specifications to an understanding of the "requirements" for a system, it is argued that other solutions should be sought than "executable specification languages".

## 1 Introduction

For the development of software, the starting point is usually a set of requirements typically given informally in natural language. The informal nature of the requirements means that misunderstandings are possible and that formal verification of an implementation is not possible. To overcome this problem the development process can be viewed as being split into phases. In fact, the so-called phases have to be iterated at least when changes to the requirements occur: it is however useful to view the required project documentation as being created by a series of idealised phases. Initially, a

[*] Copyright © 1989 The Institution of Electrical Engineers. This paper is published in the *Software Engineering Journal* Vol. 4 No. 6 pp330–338 and is reprinted here with the permission of the publisher.

[†] Department of Computer Science, University of Queensland, Australia.

[‡] Department of Computer Science, University of Manchester, England.

detailed functional[1] specification of what the system should do can be developed from the requirements. The specification can then be validated[2] against the requirements early in the development of the software. By this process many of the initial errors and misunderstandings can be detected when the cost of correction is low.

Currently specifications are most commonly written in a natural language. However, using a natural language leads to specifications that are vague and ambiguous. While such specifications do aid in detecting errors early in the development process, the imprecision of an informal specification leads to misunderstandings both in validating the specification against the requirements, and the implementation against the specification. For this reason many people have argued that a more formal approach to specification is required. Once a specification has been formalised there is a precise document against which further implementation can be verified; in fact, only then is it possible to formally verify that an implementation satisfies a specification.

There are three more-or-less distinct ways in which software design can be based on a formal specification: the "transformational" approach (e.g., [CIP85, CIP87]), the "constructive mathematics" approach (e.g., [C$^+$86]), and those methods where designs are posited and then justified at each development step. The arguments presented in this paper apply to all of these approaches, however, this presentation focuses on the third – posit and prove – approach.

Even when using a formal specification, one is left with the problem of validating the specification against the informal requirements. One approach that has been advocated is to use an executable specification and perform the validation by a series of tests on the specification. (A full discussion of notions of executability and specification can be found in [Kne89] which studies the use of symbolic execution for validating specifications.) Although considering individual test cases is useful, it is not as powerful as proving general properties about a specification. Requiring a specification notation to be directly executable restricts the forms of specification that can be used. In developing specifications of computing systems it is necessary to be able

[1] In the normal sense of the word and not in the more restricted sense as used in *functional* programming.

[2] We use the term *(formal) verification* to indicate that an implementation has been proved to satisfy a specification, and the more general term *validate* to indicate some form of check of satisfaction.

to precisely and concisely specify its desired properties. Any formalism that is applicable should be available to specify the properties of the system: one does not wish to be restricted to a notation that can be executed.

In general, a specification written in a notation that is not directly executable will contain less implementation detail than an executable one. The process of directly matching a specification to a set of requirements will be more straightforward with a specification phrased in terms of desirable properties of the system as opposed to one containing the algorithmic details necessary to make it directly executable. Similarly, it will be easier to verify that an implementation meets the more abstract specification than to try to match an executable specification against an implementation for which different data and program structure may have been chosen.

In fact, executable specifications tend to overspecify the problem. Firstly, because the implementor is tempted to follow the algorithmic structure of the specification (although this may not be desirable for efficiency or other reasons), and secondly because the executable specification will produce particular results in cases where a more implicit specification might allow a number of different results. The latter point is all the more important because of the danger of unnecessarily constraining the choice of possible implementations.

As well as their role at the top level of the system, specifications play an important part in defining the interfaces of modules internal to the system. The validation of a module specification is done with respect to a higher-level specification, or more likely a higher-level design that meets the higher-level specification. As both the higher-level specification and design can be formalised, the validation of the internal modular design can also be formalised; here one does not have the same problem of working from informal requirements that motivated the use of executable specifications.

## 2 Deterministic Operations

A deterministic specification requires a unique result to be produced for a given input, whereas a non-deterministic specification allows a number of possible alternative results. In Section 3 we explore the subject of non-determinism. This section argues that the aim of executability offers unnecessary constraints even for deterministic specifications.

Most of what is said in this paper could be presented either in a declarative or an imperative framework. The latter has been chosen and a program – or part thereof – which changes a state is referred to as an *operation*.

## 2.1 Specifying in terms of known functions

The authors and readers of any specification can be assumed to share an understanding of a set of "known" available functions (or operators) that may be used in a specification.

The expressive power of a language required to succinctly specify a problem varies considerably with the complexity of the problem itself. Simple problems can be adequately specified in conventional (efficiently executable) programming languages. For example, using multiple assignment, an operation to swap two values may be specified thus:

$$i, j := j, i.$$

(This also has the virtue that it solves the so-called *frame problem* by making it clear that no variables other than $i$ and $j$ are to be changed.)

A very high-level language (e.g., SETL [FSS83]) expands the range of problems that can be succinctly specified by providing a richer set of available functions that can be used. A powerful technique is the use of functions that are available for objects which are more abstract than those of the final implementation language. For example, if $s_1$ and $s_2$ are sets, then

$$s_1, s_2 := s_1 \cap s_2, s_1 \cup s_2$$

defines a functional behaviour which might – when working on representations in terms of linked lists – have to be implemented in several procedures.

Apart from the lack of available functions, there is a more subtle problem with such specifications. If the functions used are *partial*, the assignment style of specification does not really provide a suitable way of recording a *pre-condition*. Such assumptions are often crucial in specifying a system.

As a larger example of an operation that can be specified in a functional manner consider a text file update: given a text file consisting of a sequence of lines, it is to be updated both by deleting a set of numbered lines and by adding sequences of lines after given line numbers in the original file. Let *Line* be the type of a line and *Lines*, a sequence of lines:

$$Lines = \textsf{seq of } Line.$$

Then a text file update can be defined abstractly as a function

$$\mathit{update}\colon \mathit{Lines} \times (\textsf{set of } \mathbf{N}_1) \times (\mathbf{N} \to \mathit{Lines}) \to \mathit{Lines},$$

where the first parameter is the initial file (a sequence of lines); the second parameter is the set of line numbers of lines to be deleted; the third parameter is the additions, which are modelled as a mapping from a line number to the text that is to be added after that line number; and the result is the updated file.

There is, here, an important pre-condition. For an update $\mathit{update}(f, d, a)$, the lines to be deleted must be in the original file: $d \subseteq \textsf{dom}\, f$; and the additions must go after line numbers in the original file or after the pseudo line number zero to insert text at the beginning of the file: $\textsf{dom}\, a = \{0\} \cup \textsf{dom}\, f$. (Additions at every point in the file have been required – typically many of these will be empty.) These two assumptions are the pre-condition to be able to successfully apply the *update* function.

Each line in the original file is replaced by a sequence of lines in the output; this sequence consists of either the empty sequence if the line is deleted, or just the original line if that line is unaffected by the update; in either case it is augmented with additions. If $n$ is a line number then the sequence of lines that it will be replaced by in the output file is given by

$$(\textsf{if } n \in d \textsf{ then } [\,] \textsf{ else } [f(n)]) \frown a(n),$$

where $s \frown t$ is the concatenation of the sequences $s$ and $t$.

To specify *update*, for each line in the original file the sequence of lines it is replaced by is constructed and all of these sequences are concatenated to form the output:

$$\begin{aligned} &\mathit{update}(f, d, a) \;\; \triangleq \\ &\qquad a(0) \frown \textsf{conc}\, \{n \mapsto (\textsf{if } n \in d \textsf{ then } [\,] \textsf{ else } [f(n)]) \frown a(n) \mid n \in \textsf{dom}\, f\}, \end{aligned}$$

where $\textsf{conc}\, ss$ forms the concatenation of all the sequences in the sequence of sequences $ss$.

Although the above is written in a very high-level fashion, it is still quite close to an executable program in a functional programming language. Such a high-level program could be executed and such programs can tempt one to consider requiring specifications to be executable. Although this particular specification could be executed, the approach does not generalise to all specifications as is shown in Section 3.1 for a specification closely related to this one. In addition, it is crucial to specify a pre-condition for *update*, otherwise it does not make sense for all possible inputs; such pre-conditions are not usually part of an executable language but are essential in a specification.

## 2.2 Specifying by Inverse

The next question is how to specify operations where no known function is available. Notice that writing something like

$$i := gcd(i, j)$$

only shifts the problem, unless *gcd* is already fully understood.

Some specifications of novel concepts can be constructed by using a known function to constrain the *inverse* of the operation. Suppose, for example, that (integer) square root is both unfamiliar and to be specified, but that squaring is known. It is possible to fix $r$ as the largest integer square root of $n$ $(r, n \in \mathbf{N})$ by writing

$$r^2 \leq n < (r+1)^2.$$

Although this example is very small, the general approach of specifying via an inverse function should not be dismissed. It is, for example, convenient (see [Jon80]) to express the task of constructing a parse tree by stating *inter alia* that collecting the terminal strings from the wanted tree should yield the string given as input. There is, in general, no way of executing such an inverse specification. Even where a search happens to be possible, it is likely to be enormously inefficient. The point is not that it is impossible to write the required operation but rather that a clear (inverse) specification should not be disallowed because it cannot be executed.

## 2.3 Combining Clauses in a Specification

Although the technique of the previous section extends the repertoire of specifiable operations, there are many specifications which can be built up only from a combination of properties. It is widely accepted that such combinations can be built up using the operators of predicate calculus. A standard example is to specify a *SORT* operation on sequences without duplicates. Let *Useq* be the set of all sequences without duplicates:

$$Useq = \{s \in \textsf{seq of } \mathbf{N} \mid \forall i, j \in \textsf{dom}\, s \cdot i \neq j \;\Rightarrow\; s(i) \neq s(j)\}.$$

The *SORT* operation transforms a sequence *in* without duplicates to another sequence *out* without duplicates as follows:

$$\begin{array}{l} \mathit{in}, \mathit{out} \in \mathit{Useq} \\ \mathit{is\text{-}ordered}(\mathit{out}) \wedge \mathit{is\text{-}permutation}(\mathit{in}, \mathit{out}), \end{array}$$

where

$$\mathit{is\text{-}ordered} : \mathit{Useq} \to \mathbf{B}$$

$$\mathit{is\text{-}ordered}(s) \quad \underline{\triangle} \quad \forall i, j \in \mathsf{dom}\, s \cdot i < j \;\Rightarrow\; s(i) < s(j)$$

$$\mathit{is\text{-}permutation} : \mathit{Useq} \times \mathit{Useq} \to \mathbf{B}$$

$$\mathit{is\text{-}permutation}(s_1, s_2) \quad \underline{\triangle} \quad \mathsf{rng}\, s_1 = \mathsf{rng}\, s_2.$$

This version of the sorting (cf. *is-permutation*) problem is simplified by the assumption that the sequences (*Useq*) do not contain duplicate elements. (The general case is considered in Section 3.1.) But even here several interesting observations can be made. Most importantly, the conjunction of the ordering and permutation properties shows a specification technique which is just not available in an implementation language: it is essentially defining the valid outputs of *SORT* to be the intersection of the results of two processes one of which yields a very large set of permutations of *in* and the other of which can be thought of as yielding an infinite set of ordered sequences. In general, conjunction is not a construct of executable languages. With care, such conjunctions can *sometimes* be reformulated as Prolog programs. Another point about *is-permutation* is the way it is made concise by shifting between data types (here, sequences to sets). This technique can be useful in a range of specifications.

Even where a specification does not explicitly use a conjunction, the technique may be used implicitly. In both Z [Hay87, Spi89] and VDM [Jon86], *data type invariants* are considered to be conjoined to other properties over types using them.

An example of the use of conjunction in a non-trivial specification appears in work on *unification*: see chapters by Fitzgerald and Vadera in [JS90].

## 2.4 Negation in Specifications

Specifications can be built up using any expressions of predicate calculus. But, just as conjunction provides a particularly powerful extension to notions of executable languages, negation is also worthy of special mention. Consider, for example, the function to calculate the greatest common divisor (highest common factor). If one defines a common factor by the following predicate:

$$\begin{aligned}&\mathit{is\text{-}cd} : \mathbf{N}_1 \times \mathbf{N} \times \mathbf{N} \to \mathbf{B}\\ &\mathit{is\text{-}cd}(d, i, j) \quad \triangleq \quad d \ \mathsf{divides}\ i \wedge d \ \mathsf{divides}\ j,\end{aligned}$$

then one can specify the greatest common divisor as follows:

$$\begin{aligned}&\mathit{gcd}\colon \mathbf{N}_1 \times \mathbf{N}_1 \to \mathbf{N}_1\\ &\mathit{gcd}(i, j) = d \ \Leftrightarrow\\ &\quad \mathit{is\text{-}cd}(d, i, j) \wedge \neg(\exists e \in \mathbf{N}_1 \cdot \mathit{is\text{-}cd}(e, i, j) \wedge e > d).\end{aligned}$$

This specification makes use of both conjunction and negation. The structure of the specification does not lead directly to the structure of a program to calculate greatest common divisors. Although it is straightforward to implement the common divisor check *is-cd* from its specification, the same cannot be said for implementing *gcd* based directly on the structure of the specification.

If one treats the two conjuncts as each generating possible sets of results[3] (*d's*) then the value of the *gcd* must satisfy both constraints and hence must be in the intersection of the two sets. To calculate the second set based on the structure of the specification, one should calculate the finite set of *d's* that satisfy

$$\exists e \in \mathbf{N}_1 \cdot \mathit{is\text{-}cd}(e, i, j) \wedge e > d,$$

and then take the complement of this set relative to the natural numbers, giving an infinite set. As use has been made of an intermediate infinite set, this approach is not executable. One needs to reason about the problem and realise that the first set is finite so one can use it to generate possibilities, while the second predicate is used to check these possibilities; even this approach has problems as one then has to perform similar reasoning to limit

[3] This use of a Boolean function is actually an example of using the inverse of a function as discussed in Section 2.2 – some of the arguments to the Boolean function and the desired result (namely *true*) are supplied, and the other argument is generated.

the search space for possible values of $e$. Note that although it is possible to rewrite the negated existential quantification as the universal quantification

$$\forall e \in \mathbf{N}_1 \cdot \neg \textit{is-cd}(e, i, j) \vee e \leq d,$$

this only moves the problem; it does not resolve it.

All this reasoning about the problem is not necessary to just specify the task. Such reasoning is part of the process of coming up with an implementation, and in performing such reasoning one would hope to come up with a more efficient implementation than that based directly on the structure of the specification. To produce an executable specification it would be necessary both to make the specification more complicated than necessary, and to perform reasoning that would be better done at the time of designing an actual implementation.

The reader might like to consider the structurally similar specification of determining the least common multiple given below; this example is further complicated by the fact that the set of common multiples of two numbers is infinite, and – while the second conjunct generates a finite set – the set generated by the existentially quantified predicate before it is negated is infinite. Hence one cannot use either conjunct in an evaluation based on the structure of the specification without running into infinite sets.

Defining a common multiple by the predicate:

$$\textit{is-cm} : \mathbf{N} \times \mathbf{N}_1 \times \mathbf{N}_1 \rightarrow \mathbf{B}$$

$$\textit{is-cm}(m, i, j) \quad \triangleq \quad i \ \mathsf{divides} \ m \wedge j \ \mathsf{divides} \ m,$$

one can specify the least common multiple as follows:

$$\begin{array}{l} \textit{lcm}\colon \mathbf{N}_1 \times \mathbf{N}_1 \rightarrow \mathbf{N}_1 \\ \textit{lcm}(i, j) = m \ \Leftrightarrow \\ \quad \textit{is-cm}(m, i, j) \wedge \neg(\exists n \in \mathbf{N}_1 \cdot \textit{is-cm}(n, i, j) \wedge n < m). \end{array}$$

## 2.5 Quantifiers

Consider the following simple specification:

$$\textit{is-perfect-square}(i) \quad \triangleq \quad \exists j \in \mathbf{N} \cdot i = j^2.$$

A straightforward attempt to directly execute the above specification would probably enumerate the natural numbers testing each to see if $i$ is a perfect square. If it is, this will terminate; but if it is not, it will not terminate. We can guarantee termination in all cases by stopping when the enumeration gets to $i$, however, this relies on the property that the square of a natural number is always greater than or equal to the number itself.

Even this simple example involving a quantifier leads to problems for direct execution. In general, the property of the problem that is used to control the enumeration is not as simple as above; and one is required to reason about the problem (preferably in the mathematical system associated with the application area) in order to determine such properties before one can attempt execution.

## 2.6 Non-Computable Clauses in Specifications

The problem of calculating the so-called Hamming numbers is found in [Dij76]. The Hamming numbers are those whose only prime factors are 2, 3, and 5. The problem is to generate the sequence of Hamming numbers in increasing order. This sequence, *ham*, can be specified by

$$\begin{array}{l} \mathit{ham}\colon \mathbf{N}_1 \to \mathbf{N} \\ \mathit{ordered}(\mathit{ham}) \land \\ \quad \mathsf{rng}\, \mathit{ham} = \{n \in \mathbf{N} \mid \forall p \in \mathit{Primes} \cdot p \;\mathsf{divides}\; n \;\Rightarrow\; p \in \{2, 3, 5\}\}, \end{array}$$

where *Primes* is the set of all prime numbers. As this sequence is infinite, it cannot be computed in its entirety; but its prefixes can (e.g., the first 100 Hamming numbers). This can be done by stating that the output should be the prefix of *ham* of length 100. Note that if the first conjunct ($\mathit{ordered}(\mathit{ham})$) is used to generate possibilities, all of the ordered infinite sequences would have to be generated: this is not possible. However, by adding the condition that only the first 100 items are required, an implementation of the whole specification becomes possible.

In the paper entitled "Functional programs as executable specifications", Turner [Tur85, pages 43–44] makes use of the following non-executable specification for the problem of computing the Hamming numbers (given in Turner's notation):

$$\mathit{ham} = \mathit{SORT}\{2^a \times 3^b \times 5^c \mid a, b, c \leftarrow [0..]\}$$

where the notation in braces generates a sequence with no duplicates containing the given expression for $a$, $b$ and $c$ taking natural number values greater than or equal to zero. Note that, although this specification looks formal, it is not since $SORT$ on infinite sequences cannot be defined as a recursive function; the above use of $SORT$ is informal and the above specification is not directly executable as it involves sorting an infinite sequence. By relying on properties of the Hamming numbers, however, it can be transformed into a program that merges already ordered sequences and is executable, although as specified it never terminates.

Specifications can contain clauses that are not computable; when these clauses are conjoined with additional constraints the whole may be computable. However, the structure of the specification does not lead directly to the structure of an implementation as a component is not computable. The specification must be transformed (typically, by taking into account not necessarily obvious properties of the problem) to a form that has a different structure and is amenable to implementation.

A specification language should be expressive enough to specify non-computable problems such as the halting problem. If it is not, one cannot use the single specification notation to cover both theoretical aspects of computing and practical ones. It is possible to build a specification of implementable systems where a component of the specification is itself not computable. For example, if one takes the specification of the halting problem and adds the condition that the program being examined to determine whether or not it halts contains no loops or recursion, then the problem is trivially implementable.

One can also specify problems such as the following one related Fermat's last theorem: given $n \in \mathbf{N}$ can three natural numbers $x$, $y$ and $z$ be found such that

$$x^n + y^n = z^n.$$

Whether or not this is computable, at this time nobody has been able to determine whether or not this theorem holds. Again a specification notation should be able to specify such problems irrespective of these issues.

# 3 Non-Deterministic Operations

We hope the reader is by now aware of some of the expressive advantages of specifications which are not (necessarily) executable. The case against executable specifications changes from one of convenience to necessity when non-determinism is considered.

## 3.1 External Non-Determinism

There are some computer systems where even their external behaviour should not be too closely determined by the specification. Section 2.3 considers a simplified *SORT* problem without duplicate keys. One specification where it is reasonable to have a complete specification which does not determine a unique result is for sorting where records (*Rec*) can contain duplicate keys. The components of the records can be obtained using selector functions:

$key\colon Rec \to \mathbf{N}$
$data\colon Rec \to Data.$

*SORT* can then be specified by

$in, out \in \mathsf{seq\ of}\ Rec$
$is\text{-}ordr(out) \wedge is\text{-}permr(in, out),$

where

$$is\text{-}ordr : \mathsf{seq\ of}\ Rec \to \mathbf{B}$$

$$is\text{-}ordr(s) \quad \triangleq \quad \forall i, j \in \mathsf{dom}\, s \cdot i < j \;\;\Rightarrow\;\; key(s(i)) \leq key(s(i)),$$

and

$$is\text{-}permr : \mathsf{seq\ of}\ Rec \times \mathsf{seq\ of}\ Rec \to \mathbf{B}$$

$$is\text{-}permr(s_1, s_2) \quad \triangleq \quad bagof(s_1) = bagof(s_2),$$

where a sequence is converted into a *bag* (or multi-set) representation by

$$bagof : \mathsf{seq\ of}\ Rec \to (Rec \to \mathbf{N}_1)$$

$$bagof(s) \quad \triangleq \quad \{r \mapsto \mathsf{card}\,\{i \in \mathsf{dom}\, s \mid s(i) = r\} \mid r \in \mathsf{rng}\, s\},$$

where $\mathsf{card}$ gives the cardinality of a set: in this case the frequency of occurrence of $r$ in the sequence. Notice, here again, the advantage of finding a convenient operator (=) in another type – in this case bags.

In this example a deterministic executable sort algorithm, no matter how abstract and high-level, will yield a unique result for any given input. Such a "specification" would put a restriction on all implementations that they produce exactly the same ordering although this may not be a requirement as far as the user is concerned. Thus one cannot write a deterministic specification of the above sorting problem that allows the implementor to choose either a Quicksort or an insertion sort to implement it: an insertion sort is stable – records with identical keys retain their original order – while Quicksort is not stable. In fact, a deterministic specification may allow neither Quicksort nor an insertion sort as implementations.

A larger example is the specification of a differential file comparison, *diff*, which can be obtained by inverting the specification of *update* given in Section 2.1. The operation *diff* takes two files ($f_1, f_2$: *Lines*) as input and outputs a set of deletions ($d$: $\mathsf{set\ of}\ \mathbf{N}$) and additions ($a$: $\mathbf{N} \to Lines$) that will change the first file into the second. This can be specified by making use of the *update* function:

$$d \subseteq \mathsf{dom}\, f_1 \land \mathsf{dom}\, a = \{0\} \cup \mathsf{dom}\, f_1 \land f_2 = update(f_1, d, a)$$

Given any two input files the output deletions and additions are not, in general, uniquely determined. For example, if the first file contains two consecutive identical lines and the second file contains just one copy of the line in the same place, *diff* may either delete the first line or the second line; both choices will satisfy the specification. Hence it is not possible to use a functional program to specify *diff* without selecting a particular output and hence overconstraining the space of implementations. More importantly, the specification of *diff* above clearly describes *what diff* should do; it gives no indication of *how* it should do it. Any description of *diff* that can be executed will contain considerably more detail about how to go about computing the differences.

A particularly interesting specification which embodies such external nondeterminism is given in [Mar85]. One of the tasks considered is the constraints which must be put on the representation of lines on a raster display. It is obvious that the limitations of the pixel grid prevent, in general, a completely accurate portrayal of a line; the problem of "staircasing" is a well-known corollary of this limitation. Marshall proposes a series of consistency conditions which any acceptable implementation must fulfill. These conditions do not uniquely determine the output.

There are many examples of non-deterministic specifications for numerical algorithms; these specifications often contain constructs which are not representable in decimal (or binary) notation. Let **R** be the set of mathematical real numbers; these cannot be represented on a machine and hence we need a set *Float* of floating point approximations to real numbers which are available on the machine. Consider the example of finding the square root of a real number. Given a positive floating point number, $x\colon \mathit{Float}$, we wish to calculate its square root, $r$, so that $r^2 = x$. For positive $x$, $r$ is an element of **R** but it is not necessarily an element of *Float*. Hence the result of this operation may not be representable as an element of *Float* as the accuracy of the machine is limited. We must augment our specification to allow for the actual result to be an approximation to the square root

$$\mathit{sqrt}\colon \mathit{Float} \to \mathit{Float}$$

can be defined by

$$x \geq 0 \wedge \mathit{sqrt}(x) = r_1 \;\Rightarrow\; \exists r \in \mathbf{R} \cdot r^2 = x \quad \wedge \quad \mid r_1 - r \mid < 0 \cdot 01.$$

This is an example where we combine a deterministic clause ($r^2 = x$) involving an unrepresentable value $r$ with an additional clause that makes the actual result required not as well determined but representable as a *Float*. As the above specification contains an unrepresentable component we cannot consider the specification notation used to be directly executable.

The specification of *sqrt* does not give any indication of how to compute the square root. A possible implementation is one based on Newton's method of successive approximation. This method is based on deeper mathematical results than anything immediately obvious in the specification. In addition, it is based on the theory of the real numbers rather than *Floats*.

## 3.2 Internal Non-Determinism

The preceding section begins with a hint that non-determinism is more important than the frequency of genuinely under-determined systems might suggest. Even where the external behaviour of a system is defined to be deterministic, non-deterministic specifications of its components can arise in design. This is obvious in the case of parallelism: components whose behaviour is influenced by interference can be composed in a way which yields a deterministic system. Obvious examples of this permeate the whole of our operating systems. For example, the non-deterministic paging behaviour of programs must not be allowed to influence the outcome of user's programs.

It is at first sight surprising – but is a very important fact – that non-deterministic specifications of sub-components of deterministic systems can be useful even where the eventual implementation is also deterministic. The resolution of this apparent paradox comes from the usefulness of non-deterministic specifications to leave freedom to the implementor. Thus it is possible to make and record some design decisions but postpone other decisions to later phases of development. An example of this would be a design which can be realised by introducing, say, a buffer pool manager. The essential properties of such a manager are easy to describe whilst leaving open the question of the algorithm which chooses which free buffer to allocate on the next request. The design decision involving the manager can be verified before work commences on the choice of a particular (deterministic!) algorithm. This use of non-deterministic specifications has been shown to be very useful in the design of larger systems.

## 3.3 Under-Determined versus Non-Determinism

The discussion in the preceding section might lead the reader to the suspicion that under-determined (but deterministic) behaviour is all that is required. Obviously, this does not work in the presence of parallelism. Intriguingly, it can even fail in its absence. In fact, a semantics needs to cover the genuinely non-deterministic case even where the final implementation language is deterministic. The important fact is the way in which levels of abstraction influence the notion of "behaviour". Consider the task of specifying an operation $ARB$ which is based on a state $s\colon \mathsf{set\ of}\ \mathbf{N}$. It delivers a result $i\colon \mathbf{N}$, and has post-condition

$$i \in s.$$

This specifies that a non-deterministic choice can be made. It would seem reasonable to accept an implementation $ARB_l$ based on a state $sq\colon \mathsf{seq\ of}\ \mathbf{N}$ with the post-condition

$$i = \mathsf{hd}\, sq,$$

as satisfying this specification. But the behaviour of $ARB_l$ is non-deterministic when viewed at the set level. The actual choice can be determined from the history of $ARB$ operations, but insufficient of this history is stored in the abstract (set) state to determine the choice. This is an example where the principle of information hiding leads to the abstract level of the system appearing to be non-deterministic while the implementation is deterministic.

In the following we need to be precise about what is meant by a mathematical function: a function has only one possible result for any given argument, and two calls on a function with the same argument must always return the same result. This definition of a function is the one used in mathematics and is consistent with that used in purely functional programming languages. This means, for example, that if $S = T$, then $ARB(S)$ must equal $ARB(T)$.

Often $ARB$ is taken to be the specification of a class of mathematical functions all of which satisfy the specification and any of which can be used as an implementation. This interpretation is restrictive since it does not allow the operation $ARB_l$ to be used as an implementation of $ARB$. The implementation $ARB_l$ is a mathematical function at the sequence level: any two calls with the same sequence return the same result; but it is not a function when viewed at the set level: two different sequences can represent the same set (with the elements in different orders) and hence two different calls on $ARB_l$ with the same set (but different representations of that set) can return different results.

The above argues that considering a specification as determining a set of possible (deterministic) functions is too restrictive. An approach that avoids this restriction while still using functions is to use a function that returns a set of possible results (see, for example [Tur85, page 31]). In fact, for our purposes this is theoretically equivalent to using a relational approach; however, a function returning a set is more complicated to deal with in practice. Consider, for example, whether deterministic functions should be treated specially or whether they should just be functions returning singleton sets, and whether functional composition should be redefined or whether the function should take sets of possible inputs as well as producing sets of possible outputs. In addition, if the functions return sets of possible results then we cannot use a function as a value in an expression and we cannot use the law of substitution which is the major reason put forward for the simplicity of reasoning using a functional model. Another problem when considering executable specifications is that the set of all possible results of an operation may be infinite; this would preclude computing the complete set, but it does not preclude computing one element of the set as is required of an implementation.

Another interesting example is a non-deterministic merge as required in operating systems or transaction processing systems. For example, we have streams of commands coming from a number of different terminals and we wish to merge these into a single stream to be executed. An implementation can be considered deterministically at the level where we know about the time at which the commands arrive from the terminals, but at the abstract level (where we hide information about arrival times) non-deterministic behaviour is apparent. See [Hen82, especially page 190] for a discussion of this with respect to purely functional operating systems.

A subtle example of the use of non-determinism occurs in giving the semantics of programming languages. The goal is to leave the implementor free to allocate storage addresses to variables. Thus, a language description should not dictate a particular stack implementation for Pascal. In, for example, [BJ82] the choice of locations (*Loc*) is under-determined for precisely this reason. This example manifests the problem of something being essentially non-deterministic at the level of abstraction of the specification in spite of its being deterministic in terms of the representation chosen for an implementation.

This raises the question of the semantics which can be used, for example, to verify proof rules in the presence of "true non-determinacy". This subject is pursued in [Jon87] and [Nip86].

# 4 Other Issues

## 4.1 Specification Variables

In the specifications that we have given up to this point the specification variables have stood for the values of program variables (or abstractions thereof). However, it is useful to use specification variables that one would never think of implementing as program variables. These variables do not play a part in the actual execution of the program, rather they are used to specify the required behaviour.

In specifying a real-time system we need to be able to specify real-time constraints on operations. For example, to specify that an operation must be completed in less than two seconds we can introduce variables into the specification that represent the time before ($t$) and after ($t'$) an operation and specify that $t' - t < 2$. Such mathematical variables are used for specification and are not directly reflected in the variables of the program. They are part of the specification that an implementation has to satisfy, but will an executable specification satisfy such a constraint? Can such a specification be considered to be valid if it does not satisfy the constraint when we execute it? The point here is that we should clearly distinguish a specification and an implementation. An executable specification tends to confuse the two issues.

In specifying concurrent systems such as communications protocols it is necessary to introduce specification variables that contain histories of messages on communication channels. For example, to specify a simple communication channel we can introduce specification variables:

$$in, out \colon \mathsf{seq\ of}\ Message$$

which record the histories of messages passed in to and out of the channel, respectively. We require that the output is always a prefix of the input:

$$\exists b \in \mathsf{seq\ of}\ Message \cdot out \frown b = in.$$

The variables *in* and *out* do not correspond directly to any variables that would be found in a typical implementation, rather they are used purely to specify the desired operation of the channel in terms of observable histories. In an implementation, we are likely to have a buffer plus indices and counters. Here again we run into a problem if we insist on executable specifications, as history variables are used to help specify the problem and are not intended to be reflected by program variables.

## 4.2 Inferences from Specifications

Being able to reason about a specification is important for two reasons: firstly, one needs to be able to validate user requirements by inferring that the specification has the properties desired by the user; and secondly, one needs to be able to verify that an implementation meets the specification or alternatively derive the implementation from the specification via a sequence of refinement steps.

While an executable specification allows straightforward validation of individual test cases, it may be more difficult to validate more general properties of the system. Using a specification phrased as the conjunction of the desired properties of the system, the validation of a property may well be trivial if it is one of those used in the specification. If the particular property to be validated is not one of those used in the specification, it will typically be more difficult to derive the property from a specification complicated by the necessary algorithmic detail to make it executable.

For the task of verifying an implementation against a specification, executability of the specification is of little value. For a deterministic specification it may be possible to check that the implementation produces the same results as an executable specification for particular test cases. To verify that the implementation is correct it will typically be simpler to show that it satisfies a property-oriented specification, rather than show that it is equivalent to a more detailed executable specification. In fact, the algorithmic structures for the executable specification and the implementation may be quite different. For example, it is easier to show that Quicksort and an insertion sort both satisfy the specification given earlier than to show that either satisfies the other, and for the non-deterministic case given in Section 3.1 neither satisfies the other: they are both over specifications of the problem.

To derive an efficient implementation from a specification, the fact that the specification is (inefficiently) executable is not usually a benefit. The process of refining to an efficient implementation typically starts by inferring additional properties of the problem from the properties given in the specification. From these properties an algorithmic structure is developed that may by widely variant from the structure of the specification. For the example of calculating the square root of a real given in Section 3.1 the most interesting part of deriving the solution using Newton's method is done in the theory of real numbers; once this is done it remains to be approximated using floating point representations.

In general the structure of a specification does not correspond to the structure of an efficient implementation. System designers have to be careful to consider other, possibly more efficient structures than that of the specification before committing themselves. With an executable specification there is a greater temptation to stay with the structure of the specification and to improve its efficiency, rather than starting from the properties of the problem and deriving an alternative more efficient solution. With a property-oriented specification, although the specification may be structured, that structure has not been chosen to enable execution and hence one is encouraged to consider alternative structures.

# 5 Summary

Software can be specified in terms of a relationship between inputs and outputs. In many cases the software not only produces an explicit output but also modifies an (implicit) state of the system; such operations can also be specified in terms of a relationship between inputs and outputs, if we consider the state before the operation as an input and the state after the operation as an output.

Programs do precisely define a relationship between inputs and outputs and hence can be considered to be specifications. Indeed for many systems the program code is the only precise specification of what the system does. There are two problems, however, with using a program as a specification. Firstly, the relationship between inputs and outputs that a program specifies is typically more restrictive than is required, and secondly, the way in which the relationship is specified tends to be complicated by algorithmic details of how to compute the result.

At the level of a relationship between inputs and outputs, programs typically restrict the allowable results when compared to the results that would satisfy the real requirements of the users of the system.

Specifications are intended for human consumption – they provide a communication link between the specifier and the user, and the specifier and the implementor. For this role programs have too much detail of *how* to solve the problem, rather than specifying *what* problem is to be solved. Programs are only suitable for very simple problems where the specification of the problem is as easily expressed in a programming language as in any other medium.

We can supply a program with a legal input – just how the legality of the input is determined is not clear unless the specification includes a pre-condition and pre-conditions are not generally regarded as part of a programming language – and determine what output it returns. If the programming language is deterministic then any implementation should return the same result. If the language is not deterministic – for example, Dijkstra's guarded command language [Dij76, Dij75] or languages with concurrency constructs

– then that output may be one of a number of possible outputs for that input, but it is not necessarily the output that will be produced by another implementation or even a second run on the same implementation. Running a non-deterministic program with a particular input will not allow us to determine all possible outputs for the given input. If the output is not what was required we know the program is not correct, but if it is suitable it still does not guarantee that the program will always produce suitable results for that particular input. "Testing shows the presence of bugs not their absence" – in the case of non-deterministic programs even testing a particular input does not guarantee that the program is correct for that input! Readers who are familiar with bugs in operating systems will be well aware of this problem.

To determine that a program meets a user's requirements one has to examine the program itself as opposed to runs of the program. Using knowledge of the meaning of the programming language constructs (i.e., the semantics of the programming language) one can determine the possible outputs of a program for a particular input, or classes of input, even the class of all legal inputs. However, any reasoning about a program is difficult as the program is not concise nor is it expressed in a notation suitable for reasoning. A specification should provide the properties of the desired system, from which a program is developed to implement the system.

Specifications also play an important role for component modules of a system. Even if an implementation of a module exists, it is desirable to have a precise and concise specification of the module to avoid users of the module having to read the more complex code of the implementation and to avoid users making unwarranted assumptions about the function of the module. These aspects are further reinforced if the module implements a data abstraction; in this case, the specification is given in terms of the abstract data type, while the implementation involves more detailed programming language data structures. In a situation where an implementation exists, it is clear that the important property of a specification is that it communicate the function of the module to the users as clearly as possible; executability of a specification provides no benefit.

Both functional and logic-programming have been suggested as possible bases for executable specifications [Hen86, Kow85]. While these have the advantage of being formal and of being higher-level than most programming languages, they are too restrictive when compared to using the full power of whatever mathematical systems are applicable to the problem.

There is another – more psychological – argument against attempts to use specifications as prototypes. It has been argued above that the restriction of a specification notation so that it be executable is bound to result in less clear specifications; an actual executable specification is open to the further injury of "tuning" for increased performance. The resulting destruction of the clarity of the "specification" would lose what the current authors believe is the principal benefit of the construction of a formal specification: its ability to make the essential concepts of the specified system clear.

A wide-spectrum language [CIP85, CIP87] is one which includes facilities for specification as well as an executable subset. This approach has advantages for program refinement as a single notation can be used throughout the development process. Care has to be taken, however, with the use of such a language to avoid the pitfall of confusing the objectives of specification and prototyping.

It is relevant to draw a distinction between specification and prototyping. For user interface decisions a mock up (not necessarily a full implementation) is useful to give the user a feel for the system. Here executability is important but the function is that of a prototype rather than a specification. However, the executable prototype is typically considerably more detailed in describing *how* to compute, as opposed to the specification's *what* to compute.

Our final suggestion is that perhaps much of what is described in the literature as executable specifications would be better classified as rapid prototyping – a valuable area in its own right. The plea in this paper is that the positive advantages of specification should not be sacrificed to the separable objective of prototyping.

## Acknowledgements

The authors are grateful to many people for discussions on this topic. In particular meetings of IFIP WG 2.3 and members of the (ESPRIT-funded) RAISE project have stimulated one of the authors. We would also like jointly to thank Ralf Kneuper, Ketil Stølen and Carroll Morgan for constructive comments on drafts of this paper. Cliff Jones is grateful to SERC for support both via research grants and his Senior Fellowship and to the Wolfson Foundation for financial support.

# References

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice Hall International, 1982.

[C$^+$86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, 1986.

[CIP85] CIP Language Group. *The Munich Project CIP—Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[CIP87] CIP System Group. *The Munich Project CIP—Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of ACM*, 18(8):453–457, August 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[FSS83] S.M. Freudenberger, J.T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[Hay87] I. J. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, 1987.

[Hen82] P. Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[Hen86] P. Henderson. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering*, SE-12(2):241–250, February 1986.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, 1980.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[Jon87] C. B. Jones. Program specification and verification in VDM. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design — NATO ASI Series F: Computer and Systems Sciences, Vol. 36*, pages 149–184. Springer-Verlag, 1987.

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development.* Prentice Hall International, 1990.

[Kne89] R. Kneuper. *Symbolic Execution as a Tool for Validation of Specifications.* PhD thesis, Department of Computer Science, University of Manchester, 1989.

[Kow85] R. Kowalski. The relation between logic programming and logic specification. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27. Prentice Hall, 1985.

[Mar85] L. S. Marshall. A formal specification of straight lines on graphics devices. *Lecture Notes in Computer Science*, 186:129–147, March 1985.

[Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[Tur85] D. A. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall, 1985.