

# USING MATHEMATICS TO SPECIFY SOFTWARE

Ian Hayes  
Department of Computer Science,  
University of Queensland,  
St. Lucia, Queensland,  
AUSTRALIA 4067

May 1986

## 1 Introduction

When we try to understand computing systems we tend to build a mental model of the system. We use this model to predict the behaviour of the system in untested circumstances. Such mental models are useful but are usually limited because we cannot communicate them directly to other people, and more often than not the model is imprecise. The approach to specification described in this paper is to build a mathematical model of the system being specified. Mathematics provides a mechanism for writing down a precise model that can be communicated to others. Perhaps it should be noted that the main use of a specification is for communication between people, between users and implementors, between managers and programmers. Often communication problems occur because the language in use cannot express the desired information. To be able to write down and make precise the model of a system in a designer's head will go a long way to communicating to both the users and the implementors what the designer intended.

The mathematical model of a system should be abstract and not encumbered with algorithmic details that should be considered part of an implementation. This has the dual advantages that the specification is independent of its implementation(s), and because the designer is describing his system in abstract terms, the system is likely to be simpler; by simpler we mean simpler to understand and to use and not necessarily simpler to implement

although that may well be the case. If compared at the level of code two systems may be equally complex (however we manage to measure such things) but the system that is capable of a simpler abstract specification is going to be easier to learn (to transfer the model to the user's mind) and to use.

The specification also has an important role in the preparation of documentation. Unfortunately the number of users that are familiar with mathematical specification notations is quite small and hence the specification will probably not appear directly in the documentation. However, the documenter can make a conscious effort to recreate the model of the system in the user's mind; the mathematical model can be of great assistance in achieving this aim. For more technical audiences the inclusion of the mathematics along with natural language description will allow them access to a more precise model that will allow them to reason about the system more accurately.

## 2 Logic and set theory

Logic and set theory (including relations, functions, bags and sequences) are extremely useful mathematical tools for the specifier. To demonstrate their utility let us consider the problem of specifying time periods. These appear in many applications, e.g., timetables, diaries, project scheduling, hours worked etc. Let us look at a specific example first and then try to generalise. A typical example is the time period 10:00..11:30 today. Such a range can be represented by the start and finish times of the period with the constraint that the start time must be less than or equal to the finish time.

<i>Range</i>
<i>Start, Finish : Time</i>
$Start \leq Finish$

It is common in applications such as timetabling to want to represent more than one range. For example, we may want the period 10:00..11:30 every Monday. To represent this we could add a repetition interval to the above representation. But this is rather specific and more the sort of representation one would be looking at for an implementation rather than a specification. We need to be able to allow for other constraints like: "only in November," "except on public holidays," "subtract one hour during daylight saving time," "except in Queensland." We can model these and in fact any collection of ranges by using a set of ranges.

$$Period = \text{set of } Range$$

This model is more general than one using repetition intervals; it allows quite arbitrary periods to be represented. This is typical of what is desired at the specification level as opposed to implementation. At the specification level the simplest model is to be preferred. At the implementation level we would like to take advantage of any regular structures or common cases that appear in practice in order to provide a more efficient implementation.

If we look again at *Ranges* then we can observe that a *Range* could also be represented as a contiguous set of times in which case a period would be a set of sets of times. Typically, however, the ranges within such a set are disjoint. This allows us without any loss of information to use just a simple set of (not necessarily contiguous) times to represent the time periods; this set is just the union of all the times in the ranges. This provides us with an even simpler model of a time period. (Note that if we want to distinguish the ranges 10:00..11:00 and 11:00..12:00, which in the simple set model would be indistinguishable from the single range 10:00..12:00, then this model is not appropriate.)

With the set of times model we have a simple model of a time period; this allows us to represent almost any conceivable time period we care to. Further because the time period is simply a set, we can make use of set operators on sets of times within specifications. For example, we can take the union of the periods representing November and December:

$$NovDec = November \cup December.$$

We can take the intersection of November with the set of all Saturdays between 19:00 and 22:00:

$$ConcertTimes = November \cap Saturday19to22.$$

We can take the period *November* and subtract the holidays to give the working days in November:

$$NovWorking = November - Holidays.$$

At a more primitive level a period may be specified by giving a predicate satisfied by all (and only) the times in the period. For example, the period containing the range between 19:00 and 22:00 on every Saturday could be defined by

$$Saturday19to22 = \{t : Time \mid DayofWeek(t) = Saturday \wedge TimeofDay(t) \in 19:00 \dots 22:00\}$$

where *DayofWeek* and *TimeofDay* are functions which extract the day of the week time and the time of day, respectively, from a time.

### 3 Functions (Maps)

Another common specification tool is the function (or map). Here we use a mathematical function to represent a data structure rather than the more common notion of a function being a recipe to calculate some value. For example, a Pascal array

$$A : \mathbf{array}[1 \dots 10] \mathbf{of} \textit{char}$$

corresponds to the function

$$A : (1 \dots 10) \rightarrow \textit{char}$$

with domain the set of integers between 1 and 10 and range the set of characters. In specifications, however, we make full use of the concept of a function and allow any sets for the domain and the range. For example, a symbol table can be modelled as

$$\textit{symtab} : \textit{Sym} \mapsto \textit{Value}$$

where *Sym* is the set of all symbols and *Value* the set of all values. The arrow with a cross through it ( $\mapsto$ ) stands for a *partial* function: a partial function is not necessarily defined for all arguments. We can use function application to retrieve the value associated with a symbol *s* as *symtab*(*s*). The set of symbols in the table is given by the domain of the function:  $\text{dom}(\textit{symtab})$ . We can represent the empty symbol table by

$$\{ \}$$

and a table mapping *s*<sub>1</sub> to *v*<sub>1</sub> and *s*<sub>2</sub> to *v*<sub>2</sub> by

$$\{s_1 \mapsto v_1, s_2 \mapsto v_2\}.$$

As another example a function can be used to model a keyed file.

$$F : \textit{Key} \mapsto \textit{Record}$$

where *Key* is the set of keys and *Record* is the set of records. For every key *k* in the domain, *F*(*k*) is the corresponding record.

Functions are one of the most common and most useful tools for use in specifying computing systems. A number of powerful operators can be defined on functions which can be used in writing specifications. The most common of these are overriding one function with another, and restricting

the domain of a function. If  $f$  and  $g$  are functions of the same type then  $f$  overridden by  $g$  is a function of the same type which we denote by

$$f \oplus g.$$

If  $x$  is in the domain of  $g$  then the value of  $(f \oplus g)$  at  $x$  is  $g(x)$ :

$$x \in \text{dom } g \Rightarrow (f \oplus g)(x) = g(x),$$

and if  $x$  is not in the domain of  $g$  but is in the domain of  $f$  then the value of  $(f \oplus g)$  at  $x$  is  $f(x)$ :

$$x \notin \text{dom } g \wedge x \in \text{dom } f \Rightarrow (f \oplus g)(x) = f(x).$$

If  $x$  is neither in the domain of  $g$  nor the domain of  $f$  then  $(f \oplus g)(x)$  is not defined:

$$\text{dom}(f \oplus g) = \text{dom}(f) \cup \text{dom}(g).$$

As an example, if

$$f = \{s_1 \mapsto v_1, s_2 \mapsto v_2\}$$

and

$$g = \{s_2 \mapsto v_5, s_3 \mapsto v_4\}$$

then

$$f \oplus g = \{s_1 \mapsto v_1, s_2 \mapsto v_5, s_3 \mapsto v_4\}.$$

Updating the symbol table *symtab* so that symbol  $s$  is associated with value  $v$  regardless of the previous value or whether there was a previous value can be specified by

$$\text{symtab}' = \text{symtab} \oplus \{s \mapsto v\}$$

where *symtab* stands for the symbol table before updating and *symtab'* stands for the symbol table after. Here we use the simple case of a function  $\{s \mapsto v\}$  with domain consisting of the singleton set  $\{s\}$ . We prefer to use the general overriding operator rather than invent some specific notation for updating a single element of a function in order to economise on notation. The “ $\oplus$ ” operator is more powerful than this simple case illustrates.

The other common operator on functions that we will discuss here is domain restriction. If  $f$  is a function and  $S$  is a set of the same type as the domain of  $f$ , then

$$S \triangleleft f$$

is the function  $f$  with its domain restricted to elements in the set  $S$ :

$$x \in S \wedge x \in \text{dom } f \Rightarrow (S \triangleleft f)(x) = f(x).$$

If  $x \notin S$  or  $x \notin \text{dom } f$  then  $(S \triangleleft f)(x)$  is not defined:

$$\text{dom}(S \triangleleft f) = S \cap \text{dom}(f).$$

The complementary operator to domain restriction ( $\triangleleft$ ) is domain subtraction ( $\triangleleft$ );  $(S \triangleleft f)$  is the function  $f$  domain restricted to those elements not in the set  $S$ :

$$(S \triangleleft f) = (\text{dom } f - S) \triangleleft f$$

For example, to delete a symbol  $s$  from the symbol table *symtab* returning its corresponding value in  $v$  we can specify

$$\begin{aligned} s &\in \text{dom } \textit{symtab} \wedge \\ v &= \textit{symtab}(s) \wedge \\ \textit{symtab}' &= \{s\} \triangleleft \textit{symtab} \end{aligned}$$

Let us now consider a more realistic example that makes use of the operators that we have introduced. The problem is to update a file of keyed records with new records for some keys and to delete some of the old records. Each record in the file is indexed by a key. We can model such a file as a (partial) function from keys to records

$$F : \textit{Key} \mapsto \textit{Record}$$

The file is to be updated by deleting those records in the file whose keys are in a given set  $D$ , and by adding new records under given keys; a new record may replace an old record for a key or it may be for a key not originally in the file. We add a further restriction that we cannot both delete a record with a given key and provide a new record for that key. As an example, if the file originally contained

$$F = \{k_1 \mapsto r_1, k_2 \mapsto r_2, k_3 \mapsto r_3, k_4 \mapsto r_4\}$$

and the set of keys to be deleted is

$$D = \{k_2, k_4\}$$

and the new records to be added or updated are

$$U = \{k_3 \mapsto r_5, k_5 \mapsto r_6\}$$

then the resultant file  $F'$  will be

$$F' = \{k_1 \mapsto r_1, k_3 \mapsto r_5, k_5 \mapsto r_6\}.$$

The file update operation can be specified as

<p style="margin: 0;"><i>File Update</i></p> <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> <p style="margin: 0;"><math>F, F' : Key \leftrightarrow Record</math></p> <p style="margin: 0;"><math>D : \text{set of } Key</math></p> <p style="margin: 0;"><math>U : Key \leftrightarrow Record</math></p> <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> <p style="margin: 0;"><math>D \subseteq \text{dom}(F) \wedge</math></p> <p style="margin: 0;"><math>D \cap \text{dom}(U) = \{ \} \wedge</math></p> <p style="margin: 0;"><math>F' = (D \triangleleft F) \oplus U</math></p>
--

The keys to be deleted ( $D$ ) must be contained in the keys in the original file ( $F$ ). There must be no key which is both to be deleted and updated. The keys in the set  $D$  are removed and the updates in  $U$  are added.

## 4 Sequences

Sequences are useful for specifying objects where the order and multiplicity of components is important. This should be compared with sets where within a set the elements are not ordered and there is no concept of multiplicity of occurrence of an element: either an element is in a set or it is not. For example, the following identities hold for sets

$$\{a, b\} = \{b, a\} = \{a, a, b\}.$$

whereas for sequences both the order of the elements and multiple occurrences are significant, e.g.,

$$[a, b] \neq [b, a] \neq [a, a, b].$$

As a simple example of the use of sequences let us consider modelling a queue of *Values*:

$$q : \text{seq } \textit{Value}.$$

The operation to add a *Value*  $v$  to the end of a queue is

$$q' = q \hat{\ } [v]$$

where  $q$  is the value of the queue before the operation,  $q'$  is the value of the queue after the operation, “ $\hat{\ }$ ” is the concatenation operator on sequences, and  $[v]$  is the unit sequence containing  $v$ .

The operation to remove and return the head  $v'$  from a queue can be specified by

$$q = [v'] \hat{\ } q'$$

Note that we make full use of the power of predicates to specify the relationship between the inputs and outputs in a simple form which shows the symmetry of the enqueue and dequeue operations more clearly than the more operational specification:

$$v' = \textit{head}(q) \wedge q' = \textit{tail}(q)$$

This simple example gives a hint of the simplifications that can be attained by giving a specification in the form of a predicate relating inputs and outputs rather than a procedure to calculate the outputs from the inputs.

As another example let us consider specifying a simple spelling checker. It takes a document, which we will model as a sequence of words, and returns an ordered sequence of those words in the document which are not in a given dictionary of words.

<p style="text-align: center;"><i>Spelling Checker</i></p> <p><i>Doc</i> : seq <i>Word</i>  <i>Dict</i> : set of <i>Word</i>  <i>Unknown</i> : seq <i>Word</i></p> <hr style="border: 0.5px solid black;"/> <p><math>\text{rng}(\textit{Unknown}) = \text{rng}(\textit{Doc}) - \textit{Dict} \wedge</math>  <math>(\forall i, j : \text{dom}(\textit{Unknown}) \bullet</math>  <math>\quad i &lt; j \Rightarrow \textit{Unknown}(i) &lt; \textit{Unknown}(j))</math></p>
--

where the range (rng) of a sequence is the set of words contained as values in the sequence. The unknown words are those that are in the document but

not in the dictionary. The sequence of unknown words is in strictly ascending order.

The above specification determines the value of the sequence *Unknown* by giving its properties: it is strictly ordered and its range is the set of words in the document that are not in the dictionary. This implicit style of specification allows the specifier to express the properties of the desired result rather than giving an algorithm to calculate the result; this allows the specification to be phrased in terms closer to those of the domain of application of the system.

## 5 Non-deterministic specifications

In all of the above specifications the results of the operations have been uniquely determined by the inputs. With the style of specification we are using this does not have to be the case: by specifying the properties of the operations we specify conditions that the results must satisfy but there may be many possible results that satisfy the given conditions.

Let us look at a memory allocator as an example which requires a non-deterministic specification. We will model blocks of memory by contiguous sets of addresses:

$$Block = \{b : \text{set of } Adr \mid \exists start, finish : Adr \bullet b = start .. finish\}$$

The operation to allocate a block of memory of size  $s$  can be specified by

$\begin{array}{l} \text{Alloc} \\ \hline allocated, allocated' : \text{set of } Block \\ s : \mathbb{N} \\ newblock : Block \\ \hline \#newblock = s \wedge \\ newblock \cap (\cup allocated) = \{ \} \wedge \\ allocated' = allocated \cup \{newblock\} \end{array}$
---

*allocated* and *allocated'* represent the set of allocated blocks of memory before and after the operation, respectively. The new block must be of size  $s$  (the operator “#” returns the size of a set) and must not overlap with any of the previously allocated blocks (the unary operator “ $\cup$ ” takes a set of sets as its parameter and forms the union of all these sets). The newly allocated block is added to the set of all allocated blocks.

The above specification is non-deterministic in that the value chosen for *newblock* can be any contiguous set of addresses of size  $s$  that does not

overlap the previously allocated blocks. It is clear that this is a desirable level of specification; it guarantees the user of the allocator certain properties but does not constrain the implementor to use a particular form of memory allocation strategy (e.g., first fit).

## 6 Building specifications

So far we have looked at using mathematics to specify small units of software. The notation we have used has been a subset of Z: a specification notation introduced to the Programming Research Group of Oxford University by J.-R. Abrial [Abr82] and further developed by that group [MS84, SSMH85].

Z also contains facilities for constructing larger specifications from smaller ones using building blocks known as schemas; the definition of *Alloc* above was given in the form of a Z schema: it contains declarations of inputs, outputs, before state and after state, above the centre line and a predicate relating these variables below the line. Specifications of larger operations can be built up using operators to combine schemas which may contain some variables in common. For example, if we have the operations given by the schemas:

$$\begin{array}{|l}
 \hline
 \textit{STRemove} \\
 \hline
 \textit{symtab}, \textit{symtab}' : \textit{Sym} \leftrightarrow \textit{Value} \\
 \textit{s} : \textit{Sym} \\
 \textit{v} : \textit{Value} \\
 \hline
 \textit{s} \in \text{dom } \textit{symtab} \wedge \\
 \textit{v} = \textit{symtab}(\textit{s}) \wedge \\
 \textit{symtab}' = \{\textit{s}\} \triangleleft \textit{symtab} \\
 \hline
 \end{array}$$

and

$$\begin{array}{|l}
 \hline
 \textit{QAdd} \\
 \hline
 \textit{q}, \textit{q}' : \text{seq } \textit{Value} \\
 \textit{v} : \textit{Value} \\
 \hline
 \textit{q}' = \textit{q} \hat{\wedge} [\textit{v}] \\
 \hline
 \end{array}$$

then the operation to remove the symbol *s* from the symbol table and add its corresponding value to the queue can be specified by

$$\textit{STRemove} \wedge \textit{QAdd}$$

where “ $\wedge$ ” is the schema *and* operator. The above schema conjunction is equivalent to

$$\frac{\begin{array}{l} \text{--- } STRemove \wedge QAdd \text{ ---} \\ symtab, symtab' : Sym \leftrightarrow Value \\ s : Sym \\ v : Value \\ q, q' : seq Value \end{array}}{\begin{array}{l} s \in \text{dom } symtab \wedge \\ v = symtab(s) \wedge \\ symtab' = \{s\} \triangleleft symtab \wedge \\ q' = q \hat{\wedge} [v] \end{array}}$$

Note that the variables  $v$  in *STRemove* and in *QAdd* merge to give a single variable  $v$  in the conjunction. Schema conjunction allows us to build up a specification by specifying operations on parts of the overall state and then put these together to give the specification of the whole operation.

There are a number of schema combinators. The most common are conjunction (above) and disjunction. Disjunction is useful for specifying different alternatives of an operation (usually on the same state); a common use of this is for specifying error alternatives. For example, if the symbol is not in the symbol table we may want the symbol table and queue to remain unchanged:

$$\frac{\begin{array}{l} \text{--- } SymbolNotFound \text{ ---} \\ s : Sym \\ symtab, symtab' : Sym \leftrightarrow Value \\ q, q' : seq Value \end{array}}{\begin{array}{l} s \notin \text{dom } symtab \wedge \\ symtab' = symtab \wedge q' = q \end{array}}$$

The operation with error handling is given by

$$(STRemove \wedge QAdd) \vee SymbolNotFound$$

The schema disjunction operator provides a useful mechanism for factoring out different cases of an operation, specifying them separately, and then bringing them together for the specification of the whole operation.

## 7 Specification style

Specifications produced using Z consist of a mixture of formal material interleaved with natural language descriptions. The natural language is used

both for explanation of the formal material and as the link between the formalism and the real world. The specifications are presented in a bottom-up manner. A specification proceeds from primitive data types and operations and builds on these to produce the final specification.

A common use of Z is for specifying interfaces to modules. For a module interface we must define the abstract state on which the module operates (including any data type invariant), any constraints on the initial value of this state, and the operations on the state; the latter are defined in terms of the relationship between the inputs, outputs, before state and after state.

## 8 Experience

The formal specification techniques outlined above have been applied in a number of collaborative projects between industry in the UK and the Programming Research Group of Oxford University as well as to projects within the Programming Research Group. The projects include a conference data base done in collaboration with STL [FS85], a specification of the UNIX filing system [MS84], a specification of the ICL Data Dictionary done in collaboration with ICL [Suf84], a simple assembler [SS85], a distributed computing system [GM84], and specification of parts of the IBM CICS application programmer's interface done in collaboration with IBM (UK) Laboratories [Hay85].

Of the above the UNIX filing system, ICL Data Dictionary, and the IBM CICS work were retrospective specifications of existing systems. For this work the specification techniques were being used in the role of analysing and documenting tools; the latter two projects uncovered inconsistencies in the current systems and indicated ways in which both the user interfaces and the documentation of the systems could be improved.

The remainder of the projects involved using the specification techniques as a tool for designing new systems and for communicating the new designs between the participants in the projects. It was generally agreed that the designs so created were better thought out, more consistent, and simpler than designs done without the aid of such tools.

The Programming Research Group has expanded its role in industrial collaboration recently: it has given six courses for UK industry in 1984–85 and is starting collaborative projects with RACAL ITD, Plessey and BP Research as well as continuing its close ties with IBM (UK) Laboratories and ICL.

## 9 Acknowledgements

The evolution of specification techniques outlined in this paper has been the work of many; of special mention are the originator Jean-Raymond Abrial and my colleagues from the Programming Research Group of Oxford University including Tony Hoare, Carroll Morgan, Ib Holm Sorensen and Bernard Sufrin. The work of Cliff Jones of Manchester on VDM [Jon80] has also had an influence on the development of Z.

## References

- [Abr82] J. R. Abrial. The specification language Z: Basic library. Internal report, Programming Research Group, Oxford University, 1982.
- [FS85] L.W. Flinn and I.H. Sorensen. CAVIAR: A case study in specification. Technical Monograph PRG-48, Programming Research Group, Oxford University, July 1985.
- [GM84] Roger Gimson and C. Carroll Morgan. Ease of use through proper specification. In David A. Duce, editor, *Distributed Computing Systems Programme*. Peter Peregrinus, 1984.
- [Hay85] Ian J. Hayes. Applying formal specification to software development in industry. *IEEE Transactions on Software Engineering*, SE-11(2):169–178, February 1985.
- [Jon80] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International Series in Computer Science, 1980.
- [MS84] C. Carroll Morgan and Bernard A. Sufrin. Specification of the Unix filing system. *IEEE Trans. on Software Engineering*, SE-10(2):128–142, March 1984.
- [SS85] Ib Holm Sorensen and Bernard A. Sufrin. Formal specification and design of a simple assembler. Internal report, Programming research Group, Oxford University, 1985.
- [SSMH85] Bernard A. Sufrin, Ib Holm Sorensen, C. Carroll Morgan, and Ian J. Hayes. Notes for a Z handbook. Internal report, Programming Research Group, Oxford University, July 1985.
- [Suf84] Bernard A. Sufrin. Towards a formal specification of the ICL data dictionary. *ICL Technical Journal*, August 1984.