CHAPTER ONE

INTRODUCTION

The problems of producing reliable software quickly and economically have been widely discussed in recent years, and initial experience has indicated that the use of mathematical methods of specifying and designing software can contribute towards a solution. For these mathematical methods to become effective in industrial practice, certain prerequisites must be satisfied. First, the notations used to express specifications and designs must be standardized, so that workers on a large project can collaborate without misunderstanding, and so that text-books and training courses can be written. Second, software tools will be needed to assist with the storage and organization of specifications and designs, and to automate the manipulation of mathematical text. Both these prerequisites demand that the mathematical notation be stable and well-understood.

The style of software specification and design known as Z has had a long period of development, which has for the most part been characterized not by stability of notation but by its very opposite. The Z style has developed as a result of tackling practical examples and adapting the notation to their needs, and this has resulted in a style and a notation which are suitable for the specification of sizeable software systems. As the method has become mature, the need for radical changes in style and notation has lessened; development of Z is now at a stage when a standard notation can be fixed. As noted above, this will help to satisfy the needs of industry, as well as the needs of further research into specification-processing tools.

According to the principle of using mathematics to document specifications and designs, this means that we should try to describe the Z notation itself in a mathematical way. The central contention of this book is that such a mathematical description—a formal semantics of Z—is both feasible

and beneficial. Before turning to an account of the formal semantics of Z, we pause to review its main features, to consider the nature of formal semantics, and briefly to compare Z with some other specification languages.

1.1 Introducing Z

This is not the place for a detailed tutorial on the Z notation and method, but a short example may be a useful reminder of the style of specification they encourage. The example is a small database recording people's names and their telephone numbers.

Several features of Z are illustrated by the example. First among these is the use of *schemas* to record both static aspects of the database system—what states it can occupy—and dynamic aspects—what events are possible in the life of the system. The use of schemas allows the specification to be presented gradually, with a close correspondence between the mathematical text and prose commentary. This makes it easy to explain the mathematics as it is presented and to relate the variables in the mathematical text to the system they describe. The operation of the database with correct input will be described first, then the *schema calculus* will be used to extend the specification to deal with errors in the input.

1.1.1 *Example: a small database*

The database contains a number of people's names, and against each name is stored a telephone number. It has operations for adding a new name and telephone number, and for enquiring what number is stored against a given name. The *state-space* of the database system is described by a schema called *PhoneDB*:

$$
\begin{array}{|l}
\hline
\textit{PhoneDB} \\
\textit{known} : \mathbb{P}\, \textit{NAME} \\
\textit{phone} : \textit{NAME} \rightarrowtail\!\!\!\!\!\rightarrow \textit{PHONE} \\
\hline
\textit{known} = \operatorname{dom} \textit{phone} \\
\hline
\end{array}
$$

Two *observations* of the state are introduced as components of this schema: the set *known* of names known to the database, and the partial function *phone* which records a telephone number against certain names. An *invariant* relationship between these observations is documented: that the names

known to the database are exactly those for which a number is recorded. As an example, the following is a possible state of the system:

$$
\begin{aligned}
known &= \{\, \mathrm{Smith}, \mathrm{Jones}, \mathrm{Robinson} \,\} \\
phone &= \{\, \mathrm{Smith} \mapsto 01\text{–}325\text{–}4939, \\
&\qquad \mathrm{Jones} \mapsto 0865\text{–}54141, \\
&\qquad \mathrm{Robinson} \mapsto 0865\text{–}54141 \,\}
\end{aligned}
$$

Even simple mathematics like this allows a degree of exactness in specification which is not easy to achieve with ordinary prose. For example, we have specified precisely that each person can only have one number, and that two people may share a number, as Jones and Robinson do in the example state. We have also avoided undesirable implications in specifying the state-space: there is no implied limit on the number of entries, nor is there an implied order in which they are stored. It has also been possible to describe the state-space without making a premature decision about the format of names and numbers.

Having described the state-space of the database, we can begin to describe the events which can happen. One such event is the addition of a new number to the database, and we describe it with a schema:

$$
\begin{array}{|l}
\hline
\textit{AddPhone} \\
\Delta PhoneDB \\
name? : NAME \\
number? : PHONE \\
\hline
name \notin known \\
phone' = phone \cup \{name? \mapsto number?\} \\
\hline
\end{array}
$$

This schema describes a state change from the state with observations *known* and *phone* to the one with observations $known'$ and $phone'$; the declaration $\Delta PhoneDB$ introduces these four components of the schema. There are two inputs: *name*?, the new name to be added to the database, and *number*?, the number to be recorded against that name. By convention, these inputs are given identifiers which end in a question mark. The schema documents a pre-condition for the event to be successful: the new name must not already be one of those known to the database. It also gives the relationship which must hold between the state before the event and the state after it: the record of telephone numbers is extended with the new number.

If the *AddPhone* operation is activated with an input which does not satisfy the pre-condition, the specification says nothing about what happens:

the system may break. This means that the system is not very robust: it is easy to break it by trying to add a name twice. The task of understanding the specification is made easier, however, by initially ignoring the possibility that incorrect input will break the system. In the next subsection, we will extend the specification given here to make it robust.

From this specification of the *AddPhone* operation we can prove a simple theorem about how the set of names known to the database changes. As we expect, this set expands to include the new name:

$$known' = known \cup \{name?\}.$$

We prove this as follows, using the invariants on the states before and after the event:

$$\begin{aligned}
& known' \\
& \quad = \operatorname{dom} phone' && \text{[Invariant after]} \\
& \quad = \operatorname{dom}(phone \cup \{name? \mapsto number?\}) && \text{[Spec. of } AddPhone\text{]} \\
& \quad = (\operatorname{dom} phone) \cup \{name?\} && \text{[Facts about dom]} \\
& \quad = known \cup \{name?\}. && \text{[Invariant before]}
\end{aligned}$$

Stating and proving properties like this one is an important way of increasing confidence that the specification is accurate.

Another event is the operation of finding a number in the database. Again we describe it with a schema:

$$\begin{array}{|l}
\hline
\textit{FindPhone} \\
\Delta PhoneDB \\
name? : NAME \\
number! : PHONE \\
\hline
name? \in known \\
number! = phone(name?) \\
phone' = phone \\
\hline
\end{array}$$

This operation has one input *name*?, the name to be looked up, and one output *number*!, the number recorded for that name; by convention, outputs have identifiers ending in an exclamation mark. The pre-condition of this operation is that the name be known to the database. The output of the operation is the corresponding telephone number, and the state of the database is unchanged by the operation.

To complete the specification, we describe the state occupied by the database when it is first brought in to use. In this initial state, no numbers are recorded:

$$
\begin{array}{|l}
\hline
\textit{Init} \\
\textit{PhoneDB} \\
\hline
\textit{known} = \varnothing \\
\hline
\end{array}
$$

1.1.2 *Handling errors*

The specification so far describes a simple database of names and telephone numbers. But this database is not robust, because incorrect input could cause it to break. The *schema calculus* can be used to extend the specification to say how errors in the input are to be handled.

We begin by describing a number of new events for the database. The pre-condition of each event describes circumstances under which one of the operations on the database may fail, and the post-condition specifies that the state of the system is unchanged. Each event has a shape described by the schema $\Xi PhoneDB$:

$$
\begin{array}{|l}
\hline
\Xi PhoneDB \\
\Delta PhoneDB \\
\hline
phone' = phone \\
\hline
\end{array}
$$

Each of the events will have an output *report*! for an error message, so the event corresponding to an *AddPhone* operation for a number already known might look like this:

$$
\begin{array}{|l}
\hline
AlreadyKnown \\
\Xi PhoneDB \\
name? : NAME \\
report! : MESSAGE \\
\hline
name? \in known \\
report! = \text{`}\texttt{Name\ already\ known}\text{'} \\
\hline
\end{array}
$$

The output *report*! is of type *MESSAGE*, and we assume for the sake of the example that it is a character string.

There is also a schema describing the message which acknowledges a successful operation:

$$
\begin{array}{|l}
\hline
\textit{Ok} \\
\textit{report}! : \textit{MESSAGE} \\
\hline
\textit{report}! = \text{`}\texttt{Ok}\text{'} \\
\hline
\end{array}
$$

Now we can describe a robust version of the AddPhone operation as follows:

$$\textit{RAddPhone} \mathrel{\widehat{=}} (\textit{AddPhone} \land \textit{Ok}) \lor \textit{AlreadyKnown}.$$

Here the three schemas *AddPhone*, *Ok*, and *AlreadyKnown* are put together with the logical operators $\land$ and $\lor$ to give a new schema *RAddPhone*. This schema describes an operation which, whenever possible, behaves like *AddPhone* and gives the report '`Ok`'; but if the name given is already known, it leaves the state of the database unchanged and issues an error report.

Similarly, for *FindPhone*, we can describe the conditions under which it is appropriate to issue the message '`Name not known`':

$$
\begin{array}{|l}
\hline
\textit{NotKnown} \\
\Xi \textit{PhoneDB} \\
\textit{name}? : \textit{NAME} \\
\textit{report}! : \textit{MESSAGE} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{report}! = \text{`}\texttt{Name not known}\text{'} \\
\hline
\end{array}
$$

Combining this with the previous specification of *FindPhone* gives a robust operation:

$$\textit{RFindPhone} \mathrel{\widehat{=}} (\textit{FindPhone} \land \textit{Ok}) \lor \textit{NotKnown}.$$

The two robust operations have been specified by putting together separate fragments of specification corresponding to normal operation and error handling. Sometimes it possible to implement operations in a way that reflects this separation, but often it is necessary to structure the implementation in a different way; the practicalities of implementation should not, however, dictate the structure of the specification, which should be designed for maximum clarity.

These are the chief features of the Z style of specification. Schemas are used to describe all aspects of the system under discussion: the states it can occupy, the transitions it can make from one state to another, and even, as we transform the specification into a design for an implementation, the

relationship between one view of the state and another. Schemas may be combined in the schema calculus to build descriptions of complex entities from descriptions of their components. This allows a complex state-space to be broken into manageable pieces, and makes it possible to describe separately different aspects of a system's behaviour.

1.2 Why formal semantics?

A guiding principle of the Z approach to specification has been the use of the ordinary structures of mathematics in the writing of software specifications. There are several advantages in this: the familiar language of sets and relations proves to be sufficient to describe succinctly the abstract structures needed in programming, and is already known to every mathematician and also to many non-specialists. In following ordinary mathematical practice, we may notice that mathematicians—at least those who call themselves applied mathematicians—spend little time worrying about the 'formal semantics' of the notations they use and the 'rules of inference' used to manipulate them. Why should we be concerned with these things when we try to apply mathematics to the new sphere of software design?

A first answer lies on the nature of the notations themselves. The ordinary mathematician can regard his notation mainly as a system of abbreviations, so that the parts of a text written in mathematical notation are not different in kind from the parts written in English, at least as far as *content* is concerned. Of course, the *form* of mathematical notation suits it for purposes for which plain English text would be highly unsuited: the compactness and regularity of formulae make them easy to manipulate algebraically in a way that English text is not. But in their content, mathematical formulae can finally be regarded as abbreviations for—admittedly very lengthy—English sentences, and by carefully introducing his notations one by one, the mathematician can maintain his confidence that they are free from ambiguity. Connected with this character of mathematical notation as a system of abbreviations is the way in which the constant symbols in a formula refer to abstract mathematical objects, almost in the same way that proper names refer to people, streets and cities.

When we begin to construct mathematical descriptions of software, it soon becomes plain that one of the main difficulties lies in the sheer size of the descriptions which result. For this reason, it becomes necessary to introduce explicit means for giving structure to the description by naming parts of it,

and referring to these named parts from other parts of the description, where they may be combined and extended in various ways.

The conceptual power of these notations for modularizing specifications should come as no surprise, for the importance of modularity is one of the clearest conclusions to emerge from experience with programming languages such as MODULA (Wirth, 1977). It is important to notice the difference between these notations for modularity and those introduced in the course of ordinary mathematical practice. In mathematical discourse, the names which are introduced refer directly to the mathematical objects under study. Here, the names refer instead to parts of the description itself: a schema is a piece of mathematical text, and its name refers to this text, rather than to any mathematical object. The structuring notations describe ways of combining and extending these fragments of text, rather than abbreviating direct constructions on mathematical objects.

So these specification-structuring facilities represent a deviation from, or rather an extension of, the ordinary practice of mathematics, and they need some explanation. It is this explanation which the formal semantics provides, beginning with a mathematical model for the content of a fragment of specification.

A second argument in favour of formal semantics is its consequences for the practice of specification. A formal semantics provides a foundation for a logical calculus for reasoning about specifications and deriving consequences from them. Deriving consequences from a specification is an important aid in checking that a specification captures correctly a customer's requirements, and in validating proposed implementations. Formal semantics also provides a view of specifications which abstracts from inessential details of syntax and presentation, and this view makes it possible to compare specification techniques in an objective way, and to investigate new specification language constructs.

Finally, the successful application of formal methods in industry will be helped by software tools for storing, editing, checking, and manipulating specifications. These software tools, if they are to be effective, must be designed in the same rigorous way as any other program, and this means starting from a formal specification. Part of this specification must be a description of the specification language to be supported. For some tools, such as specification editors, a formalized syntax of the specification language will be sufficient, but other tools, such as type-checkers, and, more particularly, any tools which assist with the construction of proofs about specifications, will need to depend on a formal semantics.

1.3 Meta-circularity

The formal semantics of the Z notation given in this book is itself written using Z as a meta-language. This idea of using a notation to give a 'meta-circular' description of its own semantics forms a long-standing tradition in Computer Science, beginning with McCarthy's definition of LISP (McCarthy, 1962) by means of an interpreter itself written in LISP. But the technique is open to the criticism that it doesn't really define anything. In informal terms, if someone didn't understand Z at all, we could hardly expect his understanding to be improved by a look at a formal semantics written in Z, although perhaps if he had a partial understanding, he could use the semantics to clear up some remaining areas of doubt. More formally, we might hope that the desired semantics might be found as a 'fixed-point' of the definition, but as Stoy (1977) points out, the least fixed point is bound to be 'bottom', the semantics in which every text is meaningless.

A more subtle problem is that two slightly different but inconsistent suppositions about the semantics can both be supported by a meta-circular definition. For example, unless special precautions are taken, a meta-circular definition of a programming language will not tell us whether parameters to subroutines are passed by name or by value. If we read the text of the definition under the assumption that parameters are passed by name, then the definition will appear to describe call-by-name, and if we assume call-by-value semantics, then the definition will appear to describe call-by-value: compare (Reynolds, 1972).

Why do these criticisms not invalidate a meta-circular definition of Z? One way of answering them would be to point out that the semantics consists simply of the development of a certain mathematical theory, and this development could, at least in principle, be carried out without using Z, but rather, say, the basic language of first-order logic. Indeed, we might hope that such a development of the theory would follow very closely the development given here using Z; this hope is encouraged by the fact that potentially problematic Z constructs such as generic schemas and the combining operators of the schema-calculus are avoided in our semantic meta-language, so that the description differs from an entirely first-order description mainly in the richness of the mathematical notation used.

This argument encourages us to see the meta-circular semantics ultimately as an informal sketch of a semantics which might be formalized fully in some more primitive but less expressive logical language. But in encouraging this view it seems to miss the point, because our aims in giving

the semantics are rather different from those of logicians who seek a formal foundation for mathematics. Our purpose is not to give a grand consistency proof for the entire mathematical enterprise, nor to reduce mathematics to the most elementary terms possible, but rather to give a mathematical model which helps us to understand Z specifications and to reason about them, and this purpose is served just as well by a semantics expressed in Z as it would be by one expressed in more elementary terms.

What is more, if we are to use the formal semantics as part of the specification of software tools to assist with the process of writing and refining specifications, it is appropriate that the definition be already written in a notation designed for expressing software specifications. Writing the semantic definition in Z also provides a useful example of the flexibility of Z as a framework for developing mathematical theories.

1.4 Z and other methods

A number of different styles of mathematical specification are gaining popularity, and it is worth comparing Z with some of these. Broadly speaking, the styles are divided into *model-oriented* methods, where the aim of a specification is to construct an abstract model of the information system being specified, and *property-oriented* or *algebraic* methods, where the aim is to describe a system in terms of its desired properties, without constructing an explicit model. Among the model-oriented methods are Z and VDM (Jones, 1978, 1980, 1986; Bjørner & Jones, 1982); prominent algebraic methods are Clear (Burstall & Goguen, 1980, 1981; Sannella 1981, 1982), OBJ (Goguen, Thatcher & Wagner, 1978; Goguen & Tardo, 1979) and ACT ONE (Ehrig, Fey & Hansen, 1983).

This distinction between model-oriented and property-oriented methods is not as clear-cut as it might at first appear; in practice, Z specifications often describe certain aspects of systems by giving axioms which must be satisfied by the system, and this amounts to a property-oriented specification. Algebraic specifications often describe a collection of basic data-types in a property-oriented way, then use these to build a model of the system being specified.

1.4.1 *VDM*

The closest method to Z is the 'Vienna Development Method' (VDM), which originated at the IBM Vienna Laboratory, and has been developed in the work of Dines Bjørner and Cliff Jones. In their aims, VDM and Z are quite