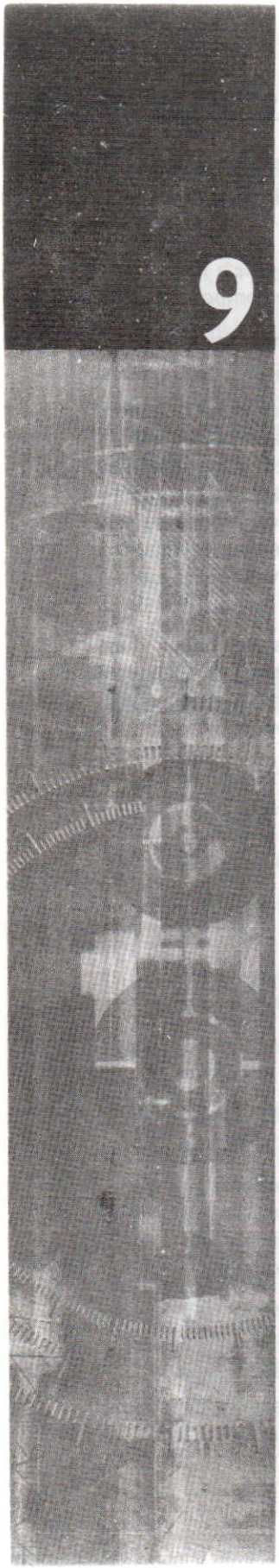


9 Formal specification

Objectives

The principal objective of this chapter is to introduce formal specification techniques that can be used to add detail to a system requirements specification. When you have read this chapter, you will:

- understand why formal specification techniques help discover problems in system requirements;
- understand the use of algebraic techniques of formal specification to define interface specifications;
- understand how formal, model-based formal techniques may be used for behavioural specification.

Contents

In 'traditional' engineering disciplines such as electrical and civil engineering, progress has usually involved the development of better mathematical techniques. The engineering industry has had no difficulty accepting the need for mathematical analysis and in incorporating mathematical analysis into its processes. Mathematical analysis is a routine part of the process of developing and validating a product design.

However, software engineering has not followed the same path. Although there has now been more than 25 years of research into the use of mathematical techniques in the software process, these techniques have had a limited impact. So-called 'formal methods' of software development are not widely used in industrial software development. Most software development companies do not consider it cost-effective to apply them in their software processes.

The term 'formal methods' includes a number of different activities, including formal system specification, specification analysis and proof, transformational development (discussed in Chapter 3) and program verification. All of these activities are dependent on a formal specification of the software. A formal software specification is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined. This need for a formal definition means that the specification languages must be based on mathematical concepts whose properties have been investigated and are well understood. The branch of mathematics which is used is called discrete mathematics and the mathematical concepts are drawn from set theory, logic and algebra.

In the 1980s, formal specification and more general formal methods were seen by many researchers as the most likely route to dramatic improvements in software quality. They argued that the rigour and detailed analysis that are an essential part of formal methods would lead to programs with fewer errors and which were more suited to user's needs. They predicted that, by the 21st century, a large proportion of software would be developed using formal methods.

Clearly, this prediction has not been fulfilled. There are a number of reasons for this:

1. *Successful software engineering* The use of software engineering methods such as structured methods, configuration management, information hiding, etc. in the design and development processes has resulted in improvements in software quality. This contradicts predictions that program proofs are essential for quality improvement.

2. *Market changes* In the 1980s, software quality was seen as the key software engineering problem. However, since then, the key issue for many classes of software development is not quality but time-to-market. Software must be developed quickly and customers are often willing to accept software with some faults if rapid delivery can be achieved. Techniques for rapid software development do not interface well with formal specifications. Of course, quality is an important factor but it must be achieved in the context of rapid delivery.

3. *Limitea scope of formal methods* Formal methods are not, in general, well suited to specifying user interfaces and user interaction. As the user interface

component has become a greater and greater part of many systems, any benefits gained from the use of formal methods are limited.

4. *Limited scalability of formal methods* Formal methods do not scale up well. Successful projects which have used these techniques have limited their use to relatively small critical kernel systems. This problem has been exacerbated by the lack of tool support for these techniques.

These factors mean that the risks of adopting formal methods on most software projects outweigh the possible benefits from their use. The costs and problems of introducing formal methods into software processes are very high. However, formal specification is an excellent way of discovering specification errors and presenting the system specification in an unambiguous way. All successful projects which have used formal methods have reported fewer errors in the delivered software.

Therefore, in systems where failure must be avoided, the use of formal methods can be justified and is likely to be cost-effective. The use of formal methods is increasing in the specialised area of critical systems development where emergent system properties such as safety, reliability and security are very important. These critical systems, as discussed in Part 4, have very high validation costs and the costs of system failure are large and are increasing. Formal methods are being used because they can reduce these costs.

Critical systems where formal methods have been applied successfully include an air traffic control information system (Hall, 1996), railway signalling systems (Dehbonei and Mejia, 1995), spacecraft systems (Easterbrook *et al.*, 1998) and medical control systems (Jacky, 1995; Jacky *et al.*, 1997). They have also been used for software tool specification (Neil *et al.*, 1998), the specification of part of IBM's CICS system (Wordsworth, 1991) and a real-time system kernel (Spivey, 1990). The Cleanroom method of software development (Mills *et al.*, 1987; Linger, 1994; Prowell *et al.*, 1999), which I discuss in Chapter 19, relies on formally based arguments that code conforms to its specification. In the UK, the use of formal methods is mandated by the Ministry of Defence for safety-critical systems (MOD, 1995).

9.1 Formal specification in the software process

In the discussion of the requirements engineering process in Chapter 6, I suggested that there were three levels of software specification which may be developed. These are user requirements, system requirements and a software design specification. The user requirements are the most abstract specification and the software design specification is the most detailed. In general, formal mathematical specifications lie somewhere between system requirements and software design specifications. They do not include implementation detail but should present a complete mathematical model of the system.

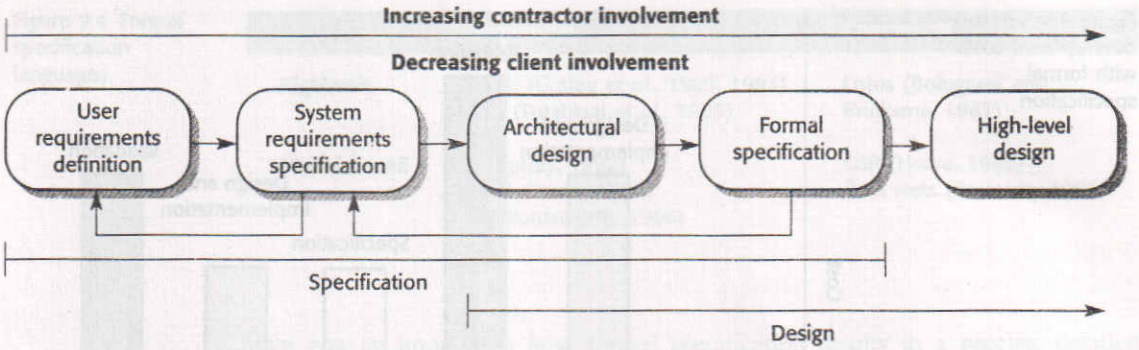


Figure 9.1 Specification and design

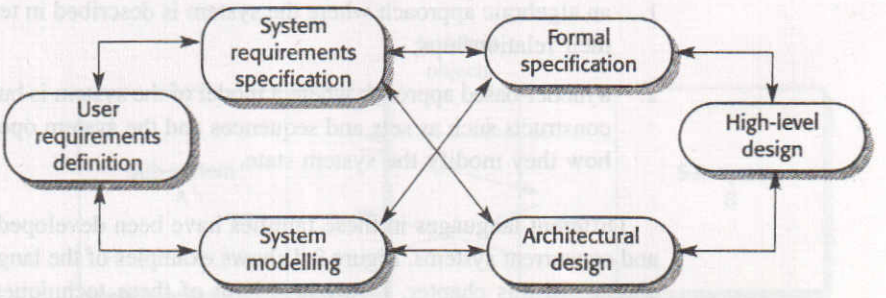
The involvement of the client for the software decreases and the involvement of the contractor increases as the specification is developed. In the early stages of the process, it is essential that the specification is 'customer-oriented'. It should be written so that it is understandable to the client and should make as few assumptions as possible about the software design. However, the final stage of the process, which is the construction of a complete, consistent and precise specification, is principally intended for the software contractor. It serves as a basis for the system implementation. This precise specification may be a formal specification.

Figure 9.1 shows the stages of software specification and its interface with the design process. The specification stages shown in Figure 9.1 are not independent nor are they necessarily developed in the sequence shown. Figure 9.2 shows that specification and design activities may be carried out in parallel streams. There is a two-way relation between each stage in the process. Information is fed from the specification to the design process and vice versa.

As a specification is developed in detail, the specifier's understanding of that specification increases. Creating a formal specification forces a detailed systems analysis that usually reveals errors and inconsistencies in the informal requirements specification. This error detection is the most potent argument for developing a formal specification (Hall, 1990). Requirements problems which remain undetected until later stages of the software process are usually expensive to correct.

Depending on the process used, specification problems discovered during formal analysis might influence changes to the requirements specification if this has not

Figure 9.2 Formal specification in the software process



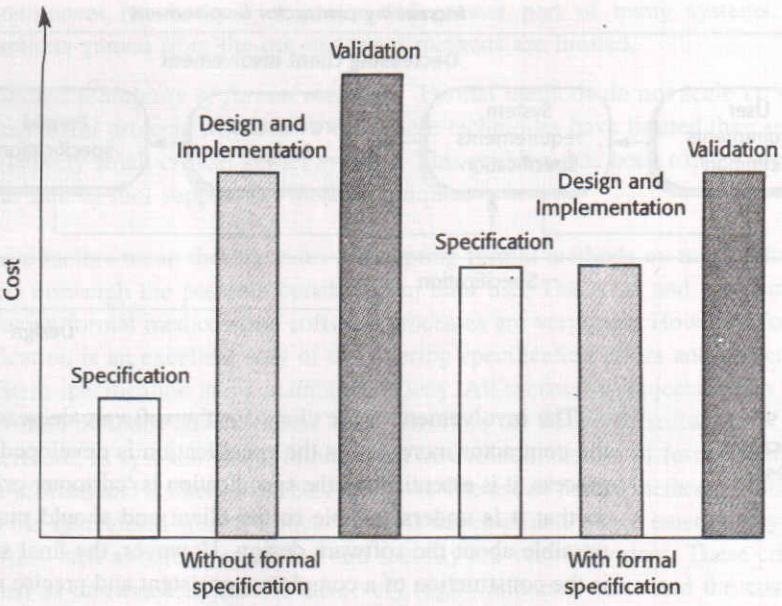


Figure 9.3 Software development costs with formal specification

already been agreed. If the requirements specification has been accepted as the basis of a contract, the problems which have been discovered should be raised with the customer for resolution before starting on the system design.

Developing and analysing a formal specification front-loads software development costs. Figure 9.3 shows how software process costs are affected by the use of formal specification. When a conventional process is used, validation costs are about 50 per cent of development costs and implementation and design costs are about twice the costs of specification. With formal specification, specification and implementation costs are comparable and system validation costs are significantly reduced. As the development of the formal specification discovers requirements problems, rework to correct these problems after the system has been designed is avoided.

There are two approaches to formal specification that have been used to write detailed specifications for non-trivial software systems. These are:

1. an algebraic approach where the system is described in terms of operations and their relationships;
2. a model-based approach where a model of the system is built using mathematical constructs such as sets and sequences and the system operations are defined by how they modify the system state.

Different languages in these families have been developed to specify sequential and concurrent systems. Figure 9.4 shows examples of the languages in each of these classes. In this chapter, I introduce both of these techniques. The examples here

Figure 9.4 Formal specification languages

	Sequential	Concurrent
Algebraic	Larch (Guttag *et al.*, 1985, 1993) OBJ (Futatsugi *et al.*, 1985)	Lotos (Bolognesi and Brinksma, 1987)
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

give you an impression how formal specification results in a precise, detailed specification but I don't discuss specification language details or specification techniques. You can download a more detailed description of both of these techniques from the book's website.

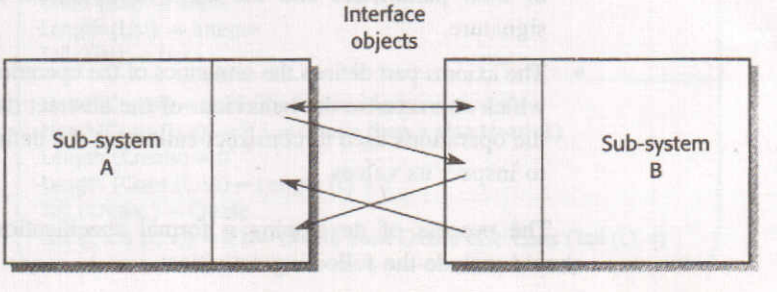
9.2 Interface specification

Large systems are usually decomposed into sub-systems that are developed independently. Sub-systems make use of other sub-systems, so an essential part of the specification process is to define sub-system interfaces. Once the interfaces are agreed and defined, the sub-systems can be developed independently.

Sub-system interfaces are often defined as a set of abstract data types or objects (Figure 9.5). These describe the data and operations that can be accessed through the sub-system interface. A sub-system interface specification can therefore be produced by combining the specifications of the components which make up the sub-system interface.

Precise sub-system interface specifications are important because sub-system developers must write code that uses the services of other sub-systems before these have been implemented. The interface specification provides information for sub-system developers so that they know what services will be available in other sub-systems and how these can be accessed. Clear and unambiguous sub-system

Figure 9.5 Sub-system interface objects



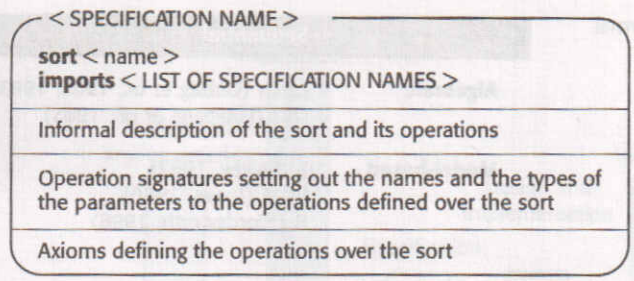


Figure 9.6 The structure of an algebraic specification

interface specifications reduce the chances of misunderstandings between a sub-system providing some service and the sub-systems using that service.

The algebraic approach was originally designed for the definition of abstract data type interfaces. In an abstract data type, the type is defined by specifying the type operations rather than the type representation. Therefore, it is very similar to an object class. The algebraic method of formal specification defines the abstract data type in terms of the relationships between the type operations.

Guttag (1977) first discussed this approach in the specification of abstract data types. Cohen *et al.* (1986) show how the technique can be extended to complete system specification using an example of a document retrieval system. Liskov and Guttag (1986) also cover the algebraic specification of abstract data types. LARCH (Guttag *et al.*, 1993) is probably the best known language for algebraic specification.

The structure of an object specification is shown in Figure 9.6. The body of the specification has four components:

- An introduction that declares the sort (the type name) of the entity being specified. A sort is the name of a set of objects. It is usually implemented as a type. The introduction may also include an imports declaration where the names of specifications defining other sorts are declared. Importing a specification makes these sorts available for use.
- A description part where the operations are described informally. This makes the formal specification easier to understand. The formal specification complements this description by providing an unambiguous syntax and semantics for the type operations.
- The signature part defines the syntax of the interface to the object class or abstract data type. The names of the operations that are defined, the number and sorts of their parameters and the sorts of operation results are described in the signature.
- The axioms part defines the semantics of the operations by defining a set of axioms which characterise the behaviour of the abstract data type. These axioms relate the operations used to construct entities of the defined sort with operations used to inspect its values.

The process of developing a formal specification of a sub-system interface should include the following activities:

1. *Specification structuring* Organise the informal interface specification into a set of abstract data types or object classes. You should informally define the operations associated with each class.
2. *Specification naming* Establish a name for each abstract type specification, decide whether or not they require generic parameters and decide on names for the sorts identified.
3. *Operation selection* Choose a set of operations for each specification based on the identified interface functionality. You should include operations to create instances of the sort, to modify the value of instances and to inspect the instance values. You may have to add functions to those initially identified in the informal interface definition.
4. *Informal operation specification* Write an informal specification of each operation. This should describe how the operations affect the defined sort.
5. *Syntax definition* Define the syntax of the operations and the parameters to each operation. This is the signature part of the formal specification. Update the informal specification at this stage if necessary.
6. *Axiom definition* Define the semantics of the operations by describing what conditions are always true for different operation combinations.

To explain the technique of algebraic specification, I will specify a simple data structure (a linked list) with a limited number of operations associated with it. This is illustrated in Figure 9.7.

Figure 9.7 A simple list specification

LIST (Elem)

sort List
imports INTEGER

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.

Create → List
Cons (List, Elem) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → List

Head (Create) = Undefined **exception** (empty list)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

To develop the example of the list specification, assume that the first stage, namely specification structuring, has been carried out and that the need for a list has been identified. The name of the specification and the name of the sort can be the same although it is useful to distinguish between these by using some convention. I use upper case for the specification name (LIST) and lower case with an initial capital for the sort name (List). As lists are collections of other types, the specification has a generic parameter (Elem). The type Elem can represent an integer, a string, a list, etc.

In general, for each abstract data type, the required operations must include an operation to bring instances of the type into existence (Create) and to construct the type from its elements (Cons). In the case of lists, there should be an operation to evaluate the first list element (Head), an operation which returns the list created by removing the first element (Tail) and an operation to count the number of list elements (Length).

To define the syntax of each of these operations, you must decide which parameters are required for the operation and the results of the operation. In general, input parameters are either the sort being defined (List) or the generic sort. Results of operations may be either of those sorts or some other sort such as Integer or Boolean. In the list example, the Length operation returns an integer. An imports declaration declaring that the specification of integer is used should therefore be included in the specification.

The axioms that define the semantics of an abstract data type are written using the operations defined in the signature part. They specify the semantics by setting out what is always true about the behaviour of entities with that abstract type.

Operations on an abstract data type usually fall into two classes:

1. *Constructor operations* which create or modify entities of the sort defined in the specification. Typically, these are given names such as Create, Update, Add or, in this case, Cons meaning construct.
2. *Inspection operations* which evaluate attributes of the sort defined in the specification. Typically, these are given names which correspond to attribute names or names such as Eval, Get, etc.

A good rule of thumb for writing an algebraic specification is to establish the constructor operations and write down an axiom for each inspection operation over each constructor. This suggests that if there are m constructor operations and n inspection operations there should be m * n axioms defined.

However, the constructor operations associated with an abstract type may not all be primitive constructors. It may be possible to define them using other constructors and inspection operations. If a constructor operation can be defined using other constructors, it is only necessary to define the inspection operations using the primitive constructors.

In the list specification, the constructor operations are Create, Cons and Tail which build lists. The inspection operations are Head and Length which are used to discover list attributes. The Tail operation, however, is not a primitive constructor. There

is therefore no need to define axioms over the Tail operation for Head and Length operations but Tail must be defined using the primitive constructors.

Evaluating the head of an empty list results in an undefined value. The specifications of Head and Tail show that Head evaluates the front of the list and Tail evaluates to the input list with its head removed. The specification of Head states that the head of a list created using Cons is either the value added to the list (if the initial list is empty) or is the same as the head of the initial list parameter to Cons. Adding an element to a list does not affect its head unless the list is empty.

Recursion is commonly used when writing algebraic specifications. The value of the Tail operation is the list which is formed by taking the input list and removing its head. The definition of Tail shows how recursion is used in constructing algebraic specifications. The operation is defined on empty lists then recursively on non-empty lists with the recursion terminating when the empty list results.

It is sometimes easier to understand recursive specifications by developing a short example. Say we have a list [5, 7] where 5 is the front of the list and 7 the end of the list. The operation Cons ([5, 7], 9) should return a list [5, 7, 9] and a Tail operation applied to this should return the list [7, 9]. The sequence of equations which results from substituting the parameters in the above specification with these values is:

```
Tail ([5, 7, 9]) =
  Tail (Cons ([5, 7], 9)) =
  Cons (Tail ([5, 7]), 9) =
  Cons (Tail (Cons ([5], 7)), 9) =
  Cons (Cons (Tail ([5]), 7), 9) =
    Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
    Cons (Cons ([Create], 7), 9) =
  Cons ([7], 9) =
    [7, 9]
```

The systematic rewriting of the axiom for Tail illustrates that it does indeed produce the anticipated result. The axiom for Head can be verified in a similar way.

Now let us look at how this technique may be used in a critical system. Assume that, in an air traffic control system, an object has been designed to represent a controlled sector of airspace. Each controlled sector may include a number of aircraft each of which has a different aircraft identifier. For safety reasons, all aircraft must be separated by at least 300 metres in height. The system warns the controller if an attempt is made to position an aircraft so that this constraint is breached.

To simplify the description, I have only defined a limited number of operations on the sector object. In a practical system, there are likely to be many more operations and more complex safety conditions related to the horizontal separation of the aircraft. The critical operations on the object are:

1. *Enter* This operation adds an aircraft (represented by an identifier) to the airspace at a specified height. There must not be other aircraft at that height or within 300 metres of it.

2. *Leave* This operation removes the specified aircraft from the controlled sector. This operation is used when the aircraft moves to an adjacent sector.

3. *Move* This operation moves an aircraft from one height to another. Again, the safety constraint that vertical separation of aircraft must be at least 300 metres is checked.

4. *Lookup* Given an aircraft identifier, this operation returns the current height of that aircraft in the sector.

It makes it easier to specify these operations, if some other interface operations are defined. These are:

1. *Create* This is a standard operation for an abstract data type. It causes an empty instance of the type to be created. In this case, it represents a sector that has no aircraft in it.

2. *Put* This is a simpler version of the *Enter* operation. It simply adds an aircraft to the sector without any associated constraint checking.

3. *In-space* Given an aircraft call sign, this Boolean operation returns true if the aircraft is in the controlled sector, false otherwise.

4. *Occupied* Given a height, this Boolean operation returns true if there is an aircraft within 300 metres of that height, false otherwise.

The advantage of defining these simpler operations is that you can then use them as building blocks to define the more complex operations on the Sector sort. The algebraic specification of this sort is shown in Figure 9.8.

Essentially, the basic constructor operations are Create and Put and I use these in the specification of the other operations. Occupied and In-space are checking operations that are defined using Create and Put and are then used in other specifications. I don't have space to explain all operations in detail here but I will discuss two of them (Occupied and Move). With this information, you should be able to understand the specification of the other operations.

1. The specification of Occupied states that in an empty airspace (Create) a level is always vacant. When applied to an airspace which has been populated using a Put operation, Occupied checks if the specified height is within 300 metres of the height of the aircraft added to the sector by the Put operation. If so, the height is already occupied so the value of Occupied is true. If not, the operation checks the sector recursively. You can think of this check being carried out on the last aircraft put into the sector. If the height is not within range of the height of that aircraft, the operation then checks against the previous aircraft which has been put into the sector and so on. Eventually, if there are no aircraft within range of the specified height, the check is carried out against an empty sector so returns false.

Figure 9.8 The specification of a controlled sector

```
SECTOR
sort Sector
imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfied
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector
Put - adds an aircraft to a sector with no constraint checks
In-space - checks if an aircraft is already in a sector
Occupied - checks if a specified height is available

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Enter (S, CS, H) =
   if    In-space (S, CS ) then  S exception (Aircraft already in sector)
   elsif Occupied (S, H) then  S exception (Height conflict)
   else  Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
   if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
   if    S = Create then Create  exception (No aircraft in sector)
   elsif not In-space (S, CS) then S  exception (Aircraft not in sector)
   elsif Occupied (S, H) then S exception (Height conflict)
   else  Put (Leave (S, CS), CS, H)

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) =  NO-HEIGHT  exception (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
   if CS = CS1  then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
   if    (H1 > H and H1 - H ≤ 300) or (H > H1 and  H - H1 ≤ 300)  then true
   else  Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS ) =
   if CS = CS1 then true else In-space (S, CS)
```

2. The specification of Move states that, if a Move operation is applied to an empty airspace (Create), the airspace is unchanged and an exception is raised to indicate that the specified aircraft is not in the airspace. When applied to an airspace which is populated using Put, the operation first checks (using In-space) if the aircraft is in the sector. If not, an exception is raised. If it is in the sector, it then checks that the specified height is available (using Occupied), raising an exception if there is already an aircraft at that height. If the space is available, the Move operation is equivalent to the specified aircraft leaving the airspace and being put into the sector at the new height.

9.3 Behavioural specification

The simple algebraic techniques described in the previous section are particularly suited to describing interfaces where the object operations are independent of the object state. That is, the results of applying an operation should not depend on the results of previous operations. Where this condition does not hold, algebraic techniques can become cumbersome. Furthermore, I think that algebraic descriptions of system behaviour are often artificial and difficult to understand.

An alternative approach to formal specification that has been more widely used in industrial projects is model-based specification. Model-based specification is an approach to formal specification where the system specification is expressed as a system state model. System operations are specified by defining how they affect the state of the system model. Hence, the behaviour of the system may be defined.

Mature notations for developing model-based specifications are VDM (Jones, 1980; Jones, 1986), B (Wordsworth, 1996) and Z (Hayes, 1987; Spivey, 1992). I use Z (pronounced Zed, not Zee) here. In Z, systems are modelled using sets and relations between sets. However, Z has augmented these mathematical concepts with constructs that specifically support software specification.

In an introduction to model-based specification, I can only give an overview of how a specification can be developed. A complete description of the Z notation would be longer than this chapter. Rather, I present some small examples to illustrate the technique and introduce notation as it is required. A full description of the Z notation is given in textbooks such as those by Diller (1994), Woodcock (Woodcock and Davies, 1996) and Jacky (1997).

Formal specifications can be difficult and tedious to read, especially when they are presented as large mathematical formulae. Understandably, this has inhibited many software engineers from investigating their potential in systems development. The designers of Z have paid particular attention to this problem. Specifications are presented as informal text supplemented with formal descriptions. The formal description is included as small, easy to read chunks (called schemas) which are distinguished from associated text using graphical highlighting. Schemas are used to introduce state variables and to define constraints and operations on the state. Schema operations include schema composition, schema renaming and schema hiding. These operations allow schemas to be manipulated. They are a powerful mechanism for system specification.

To be most effective, a formal specification must be supplemented by supporting, informal description. The Z schema presentation has been designed so that it stands out from surrounding text (Figure 9.9).

The schema signature defines the entities which make up the state of the system and the schema predicate sets out conditions which must always be true for these entities. Where a schema defines an operation, the predicate may set out pre- and post-conditions. The difference between these pre- and post-conditions defines the action specified in the operation schema.

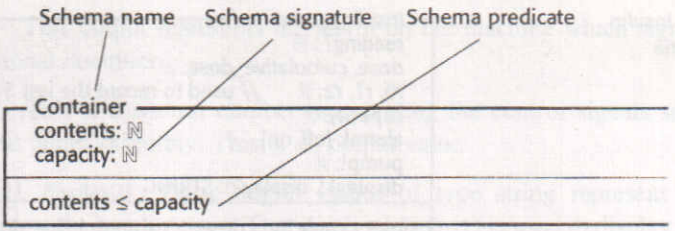


Figure 9.9 The structure of a Z schema

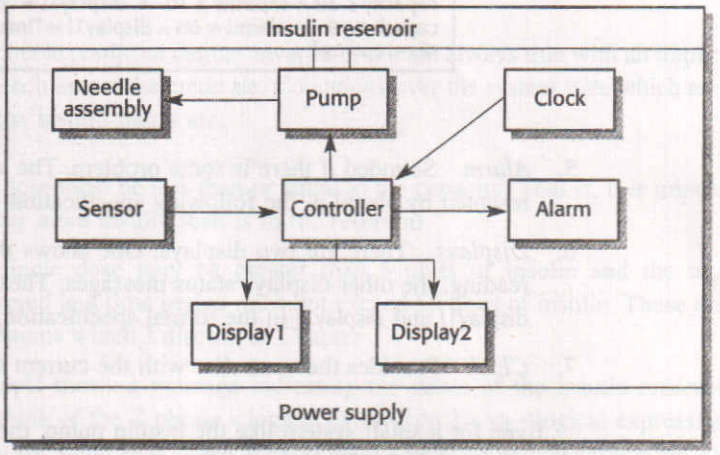


Figure 9.10 A block diagram of an insulin pump

To illustrate the use of Z in the specification of a critical system, I use a simplified example of an insulin pump system used by diabetics. Diabetics cannot metabolise glucose naturally and require injections of genetically engineered insulin, a hormone which is essential for glucose metabolism. This system monitors the blood glucose level of diabetics and automatically injects insulin as required. I also use this example in Part 4 where I cover the development of critical systems.

Figure 9.10 illustrates the structure of this insulin pump. The blood glucose is monitored at regular intervals and, if it is increasing, insulin is injected to bring down the glucose level.

1. *Needle assembly* Connected to pump. Component used to deliver insulin into the diabetic's body.
2. *Sensor* Measures the level of glucose in the user's blood. The input from the sensor is represented by reading? in the formal specification.
3. *Pump* Pumps insulin from a reservoir to the needle assembly. The value representing the number of increments of insulin to be administered is represented by dose in the formal specification.
4. *Controller* Controls the entire system. Has on/off switch plus an override button plus a button to set the amount to be delivered. For simplicity, I haven't modelled this override capability in the formal description.

Figure 9.11 Insulin pump schema

```
Insulin_pump
reading? : ℕ
dose, cumulative_dose: ℕ
r0, r1, r2: ℕ      // used to record the last 3 readings taken
capacity: ℕ
alarm!: {off, on}
pump!: ℕ
display1!, display2!: STRING
---
dose ≤ capacity ∧ dose ≤ 5 ∧ cumulative_dose ≤ 50
capacity ≥ 40  ⇒  display1! = " "
capacity ≤ 39 ∧ capacity ≥ 10 ⇒  display1! = "Insulin low"
capacity ≤ 9 ⇒  alarm! = on ∧ display1! = "Insulin very low"
r2 = reading?
```

5. *Alarm* Sounded if there is some problem. The value sent to the alarm is represented by alarm! in the following specification.
6. *Displays* There are two displays. One shows the last measured blood sugar reading, the other displays status messages. These displays are represented by display1! and display2! in the formal specification.
7. *Clock* Provides the controller with the current time.

Even for a small system like the insulin pump, the formal specification is fairly long. Although the basic operation of the system is simple, there are many possible alarm conditions that have to be considered. Therefore, I include only some of the principal schemas of the specification here and explain what they mean.

The basic schema which models the state of the insulin pump is shown in Figure 9.11. We can see here how the two basic parts of the schema are used. In the top part, names and types are declared and in the bottom part of the schema some conditions which are always true are set out.

The state is modelled using a number of state variables. By convention, names in Z which are followed by a ? are used to represent inputs and names which are followed by a ! represent outputs. However, these are simply naming conventions and the variables represented are treated in exactly the same way as any other state variables. The names declared in the schema are:

1. reading? This is a natural number (i.e. a non-negative integer) that represents the reading from the blood glucose sensor. This is an input value.
2. dose, cumulative_dose These are also natural numbers representing the dose of insulin to be delivered and the cumulative dose of insulin delivered over some period of time.
3. r0, r1, r2 These represent the last three readings and are used to compute the rate of change of blood glucose.
4. capacity A natural number representing the capacity of the insulin reservoir in the pump.

5. alarm! This output represents the alarm on the machine which signals some exceptional condition.

6. pump! This is a natural number representing the control signals sent to the physical pump assembly. This is an output value.

7. display1!, display2! These output values of type string represent two text displays on the insulin pump. One display (display1) is used to display text messages, the other (display2) is used to show the dose of insulin delivered.

The schema predicate defines invariants that are always true with an implicit 'and' between each line of the predicate. Conditions over the system state which are always true for the insulin pump are:

1. The dose must be less than or equal to the capacity. That is, it is impossible to deliver more insulin than is in the reservoir.

2. No single dose may be greater than 5 units of insulin and the total dose delivered in a time period must not exceed 50 units of insulin. These are safety conditions which I discuss in Chapter 17.

3. display1! shows a message indicating the status of the insulin reservoir. You can think of the Z phrase <logical expression 1> ⇒ <logical expression 2> as being the same as **if** <logical expression 1> **then** <logical expression 2>. If the reservoir holds 40 units or more, then the display is blank, if between 10 and 40 units, then a warning is displayed, if less than 10 units, then an alarm is sounded and a more urgent warning is displayed.

The insulin pump operates by checking the blood glucose every 10 minutes and (simplistically) insulin is delivered if the rate of change of blood glucose is increasing. The amount of insulin to be delivered is computed as shown in Figure 9.12 which shows the DOSAGE schema. If the rate of change is static, a fixed amount of insulin is delivered. You can see from this schema that there are various different situations that affect the dose delivered. The details of these are not really relevant here.

I don't model the temporal behaviour of the system (i.e. the fact that the glucose sensor is checked every 10 minutes) using Z. Although this is certainly possible, it is rather clumsy and, in my view, an informal description actually communicates the specification more concisely than a formal specification.

Figure 9.12 illustrates a commonly used feature of Z, namely the delta schema. If a schema name is included in the declarations part, this is equivalent to including all the names declared in that schema in the declaration and the conditions are included in the predicate part. They are conjoined (anded) with the predicates that are declared in the DOSAGE schema. If the schema name is preceded by a Δ, this introduces a new set of values whose names are the same as those declared in the included schema but which are 'decorated' with a prime (′) symbol. These primed values represent the values of the associated state variables after an operation.

Figure 9.12
DOSAGE schema

```
DOSAGE
ΔInsulin_Pump
(
dose = 0 ∧
  (
        (( r1 ≥ r0) ∧ ( r2 = r1)) ∨
        (( r1 > r0) ∧ (r2 ≤ r1)) ∨
        (( r1 < r0) ∧ ((r1-r2) > (r0-r1)))
  ) ∨
 dose = 4 ∧
  (
        (( r1 ≤ r0) ∧ (r2=r1)) ∨
        (( r1 < r0) ∧ ((r1-r2) ≤ (r0-r1)))
  ) ∨
dose =(r2 - r1) * 4 ∧
  (
        (( r1 ≤ r0) ∧ (r2 > r1)) ∨
        (( r1 > r0) ∧ ((r2 - r1) ≥ (r1 - r0)))
  )
)
capacity' = capacity - dose
cumulative_dose' = cumulative_dose + dose
r0' = r1 ∧ r1' = r2
```

Figure 9.13
Output schemas

```
DISPLAY
ΔInsulin_Pump
display2!' = Nat_to_string (dose) ∧
(reading? < 3 ⇒ display1!' = "Sugar low" ∨
reading? > 30 ⇒ display1!' = "Sugar high" ∨
reading? ≥ 3 and reading? ≤ 30 ⇒ display1!' = "OK")
```

```
ALARM
ΔInsulin_Pump
( reading? < 3 ∨ reading? > 30 ) ⇒ alarm!' = on ∨
( reading? ≥ 3 ∧ reading? ≤ 30 ) ⇒ alarm!' = off
```

Therefore, if an operation modifies a value val (say), the value after the operation is referred to as val′. The delta schema in DOSAGE therefore introduces the names capacity′, cumulative_dose′, etc.

The schemas modelling the outputs of the insulin pump device are shown in Figure 9.13. These model the machine displays and the built-in alarm. Again a delta schema is used to introduce the decorated variable names although, in this case, only the output variables are referenced. The DISPLAY schema states that display2! shows the dose which has been computed (Nat_to_string is a conversion function) and that display1! either shows a warning message or 'OK'. The ALARM schema sets out the conditions when the alarm is activated. It is switched on if the blood glucose reading is too low (less than 3) or too high (more than 30).

The predicates in all of the Z schemas must be consistent. That is, there should not be a statement made in one schema which contradicts the predicate in another

schema. When there are inconsistencies, these often indicate specification problems and various mathematical techniques may be applied to a Z specification for inconsistency analysis. I won't go into these here. However, by examining the four Z schemas, we can see an inconsistency which I have deliberately introduced but which could plausibly be introduced by accident in a real specification.

In the overall Insulin_pump schema, it is stated that display1! should show the state of the insulin reservoir. However, the DISPLAY schema states that this display should show a different type of warning message or should indicate that the blood glucose level is within acceptable bounds. There is a conflict here. What therefore should be shown on this display? This has to be resolved by discussing the problem with medical experts and potential users of the system.

Highlighting problems which must be resolved is one of the main advantages of using a formal specification. With an informal specification, it is much easier to overlook these conflicts and they must be resolved at a later stage of the development process.

KEY POINTS

- Methods of formal system specification complement informal specification techniques. They may be used to refine a detailed but informal system requirements specification. They therefore help bridge the gap between requirements and design.
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification and avoid some of the problems of language misinterpretation. However, non-specialists may find formal specifications difficult to understand.
- The principal value of using formal methods in the software process is that it forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system.
- Formal specification techniques are most applicable in the development of critical systems where safety, reliability and security are particularly important. They may also be used to specify standards.
- Algebraic techniques of formal specification are particularly suited to specifying interfaces where the interface is defined as a set of object classes or abstract data types. These techniques conceal the system state and specify the system in terms of relationships between the interface operations.
- Model-based techniques model the system using mathematical constructs such as sets and functions. They may expose the system state and this simplifies some types of behavioural specification.

FURTHER READING

IEEE Transactions on Software Engineering, January 1998. This issue of the journal includes a special section on the practical uses of formal methods in software engineering. It includes papers on both Z and LARCH.

'Formal methods: Promises and problems'. This article is a realistic discussion of the potential gains from using formal methods and the difficulties of integrating the use of formal methods into practical software development. (Luqi and J. Goguen, *IEEE Software*, 14(1), January 1997.)

'Strategies for incorporating formal specifications in software development'. In this article, the authors classify different approaches to formal specification. They suggest that the lack of methodological and tool support is a significant factor in the non-use of formal methods. (M. D. Fraser, K. Kumar and V. Vaishnavi, *Comm. ACM*, 37(10), October 1994.)

'A specifier's introduction to formal methods'. Although this is now more than 10 years old, fundamental specification techniques have not changed significantly. This is a good introductory article which describes the principles of formal specification and introduces a number of different approaches. (J. M. Wing, *IEEE Computer*, 23(9), September 1990.)

EXERCISES

9.1 Suggest why the architectural design of a system should precede the development of a formal specification.

9.2 You have been given the task of 'selling' formal specification techniques to a software development organisation. Outline how you would go about explaining the advantages of formal specifications to sceptical, practising software engineers.

9.3 Explain why it is particularly important to define sub-system interfaces in a precise way and why algebraic specification is particularly appropriate for sub-system interface specification.

9.4 An abstract data type representing a stack has the following operations associated with it:

New:	Bring a stack into existence
Push:	Add an element to the top of the stack
Top:	Evaluate the element on top of the stack
Retract:	Remove the top element from the stack and return the modified stack
Empty:	True if there are no elements on the stack

Define this abstract data type using an algebraic specification.

9.5 In the example of a controlled airspace sector, the safety condition is that aircraft may not be within 300 m of height in the same sector. Modify the specification shown in Figure 9.8 to allow aircraft to occupy the same height in the sector so long as they are separated by at least 8 km of horizontal difference. You may ignore aircraft in adjacent sectors. *Hint*: You have to modify the constructor operations so that they include the aircraft position as well as its height. You also have to define an operation that, given two positions, returns the separation between them.

9.6 Bank teller machines rely on using information on the user's card giving the bank identifier, the account number and the user's personal identifier. They also derive account information from a central database and update that database on completion of a transaction. Using your knowledge of ATM operation, write Z schemas defining the state of the system, card validation (where the user's identifier is checked) and cash withdrawal.

9.7 Modify the Insulin_pump schema shown in Figure 9.11 and the DOSAGE schema shown in Figure 9.12 so that the system user may override the computed dose and specify the dose to be delivered. The safety conditions specified in the Insulin_pump schema should still hold. The fact that the system is operating in override mode should be indicated on a display.

9.8 You are a systems engineer and you are asked to suggest the best way to develop the safety-critical software for a heart pacemaker. You suggest formally specifying the system but your suggestion is rejected by your manager. You think his reasons are weak and based on prejudice. Is it ethical to develop the system using methods that you think are inadequate?