27 Formal Specification

Objectives

The 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 are used for behavioral specification.

Contents

27.1 Formal specification in the software process
27.2 Sub-system interface specification
27.3 Behavioral specification

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 30 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 development processes.

The term ‘formal methods’ is used to refer to any activities that rely on mathematical representations of software including formal system specification, specification analysis and proof, transformational development, 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 are well understood. The branch of mathematics used is discrete mathematics and the mathematical concepts are drawn from set theory, logic and algebra.

In the 1980s, many software engineering researchers proposed that using formal development methods was the best way to improve 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 users’ needs. They predicted that, by the 21st century, a large proportion of software would be developed using formal methods.

Clearly, this prediction has not come true. There are four main reasons for this:

1. *Successful software engineering* The use of other software engineering methods such as structured methods, configuration management and information hiding in software design and development processes have resulted in improvements in software quality. People who suggested that the only way to improve software quality was by using formal methods were clearly wrong.

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

3. *Limited scope of formal methods* Formal methods are not well suited to specifying user interfaces and user interaction. The user interface component has become a greater and greater part of most systems, so you can only really use formal methods when developing the other parts of the system.

4. *Limited scalability of formal methods* Formal methods still do not scale up well. Successful projects that have used these techniques have mostly been concerned with relatively small, critical kernel systems. As systems increase in size, the time and effort required to develop a formal specification grows disproportionately.

These factors mean that most software development companies have been unwilling to risk using formal methods in their development process. However, formal specification is an excellent way of discovering specification errors and presenting the system specification in an unambiguous way. Organizations that have made the investment in formal methods have reported fewer errors in the delivered software without an increase in development costs. It seems that formal methods can be cost-effective if their use is limited to core parts of the system and if companies are willing to make the high initial investment in this technology.

The use of formal methods is increasing in the area of critical systems development, where emergent system properties such as safety, reliability and security are very important. The high cost of failure in these systems means that companies are willing to accept the high introductory costs of formal methods to ensure that their software is as dependable as possible. As I discuss in Chapter 24, critical systems have very high validation costs and the costs of system failure are large and increasing. Formal methods 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 (Prowell*, et al.*, 1999), relies on formally based arguments that code conforms to its specification. Because reasoning about the security of a system is also possible if a formal specification is developed, it is likely that secure systems will be an important area for formal methods use (Hall and Chapman, 2002).

27.1 Formal specification in the software process

Critical systems development usually involves a plan-based software process that is based on the waterfall model of development discussed in Chapter 4. Both the system requirements and the system design are expressed in detail and carefully analysed and checked before implementation begins. If a formal specification of

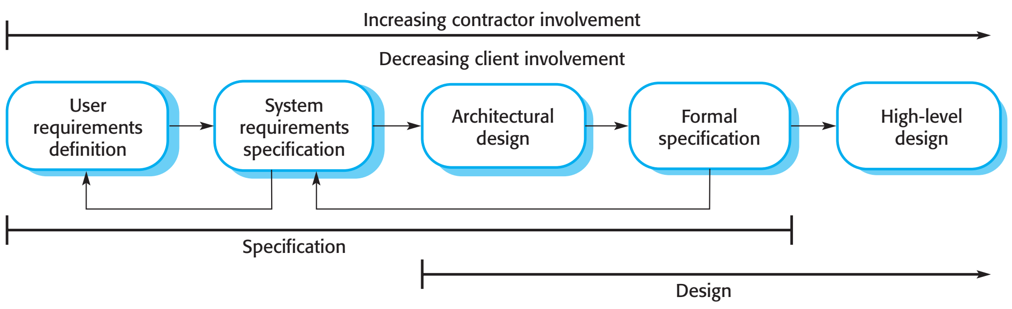
Increasing contractor involvement
Decreasing client involvement
User requirements definition
System requirements specification
Architectural design
Formal specification
High-level design
Specification
Design

Figure 27.1 Specification and design

the software is developed, this usually comes after the system requirements have been specified but before the detailed system design. There is a tight feedback loop between the detailed requirements specification and the formal specification. As I discuss later, one of the main benefits of formal specification is its ability to uncover problems and ambiguities in the system requirements.

The involvement of the client decreases and the involvement of the contractor increases as more detail is added to the system specification. In the early stages of the process, the specification should be 'customer-oriented'. You should write the specification so that the client can understand it, and you 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 precisely specifies the details of the system implementation. You may use a formal language at this stage to avoid ambiguity in the software specification.

Figure 27.1 shows the stages of software specification and its interface with the design process. The specification stages shown in Figure 27.1 are not independent nor are they necessarily developed in the sequence shown. Figure 27.2 shows specification and design activities may be carried out in parallel streams. There is a two-way relationship between each stage in the process. Information is fed from the specification to the design process and vice versa.

As you develop the specification in detail, your understanding of that specification increases. Creating a formal specification forces you to make a detailed systems analysis that usually reveals errors and inconsistencies in the informal requirements specification. This error detection is probably the most

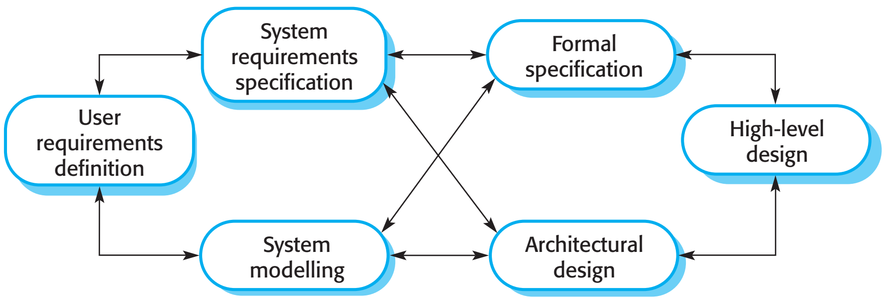

Figure 27.2 Formal specification in the software process

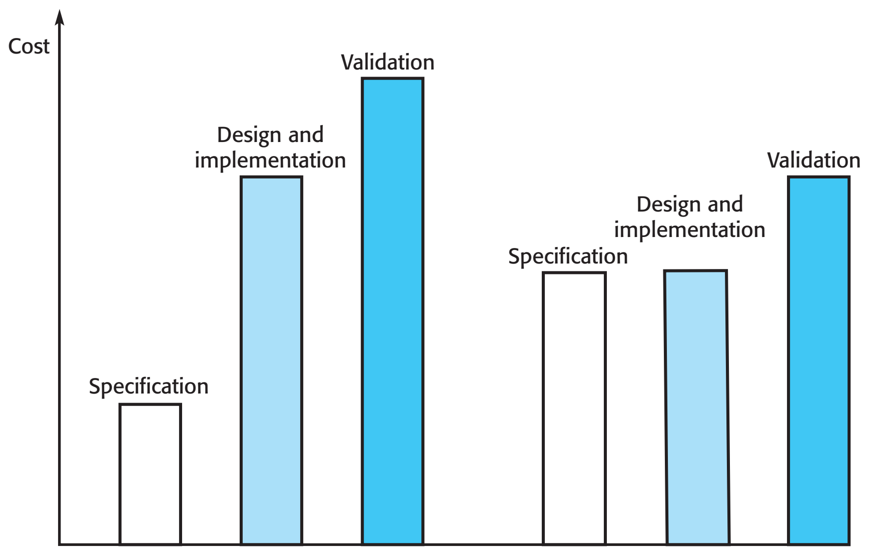

Figure 27.3 Software development costs with formal specification

potent argument for developing a formal specification (Hall, 1990). It helps you discover requirements problems that can be very expensive to correct later.

Depending on the process used, specification problems discovered during formal analysis might influence changes to the requirements specification if this has not already been agreed. If the requirements specification has been agreed and is included in the system development contract, you should raise the problems that you have found with the customer. It is then up to the customer to decide how they should be resolved before you start the system design process.

Developing and analysing a formal specification front-loads software development costs. Figure 27.3 shows how software process costs are likely to be affected by the use of formal specification. When a conventional process is used, validation costs are about 50% 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 uncovers requirements problems, rework to correct these problems after the system has been designed is avoided.

Two fundamental approaches to formal specification have been used to write detailed specifications for industrial 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 27.4 shows examples of the languages in

	Sequential	**Concurrent**
Algebraic	Larch (Guttag, *et al.*, 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)

Figure 27.4 Formal specification languages

each of these classes. You can see from this table that most of these languages were developed in the 1980s. It takes several years to refine a formal specification language, so most formal specification research is now based on these languages and is not concerned with inventing new notations.

In this chapter, my aim is to introduce both algebraic and model-based approaches. The examples here should give you an idea of how formal specification results in a precise, detailed specification, but I don't discuss specification language details, specification techniques or methods of program verification. You can download a more detailed description of both algebraic and model-based techniques from the book's website.

27.2 Sub-system 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 objects or components (Figure 27.5). These describe the data and operations that can be accessed through the sub-system interface. You can therefore define a sub-system interface specification by combining the specifications of the objects that make up the interface.

Precise sub-system interface specifications are important because sub-system developers must write code that uses the services of other sub-systems

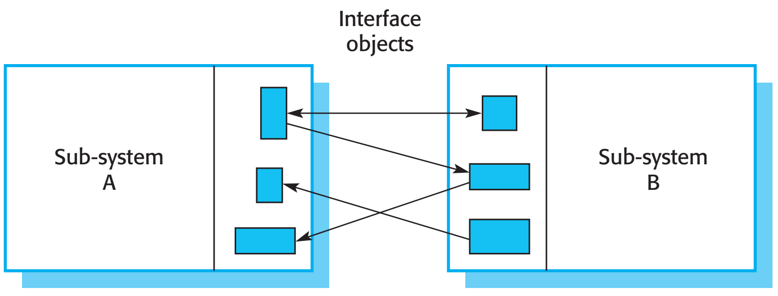

Figure 27.5 Sub-system interface objects

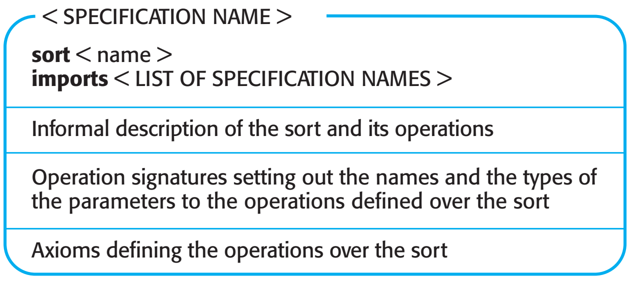
< SPECIFICATION NAME >
sort < name >
imports < LIST OF SPECIFICATION NAMES >
Informal description of the sort and its operations
Operation signatures setting out the names and the types of the parameters to the operations defined over the sort
Axioms defining the operations over the sort

Figure 27.6 The structure of an algebraic specification

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 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 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 (Guttag, 1977) first discussed this approach in the specification of abstract data types. Cohen *et al*. (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 (Liskov and Guttag, 1986) also cover the algebraic specification of abstract data types.

The structure of an object specification is shown in Figure 27.6. The body of the specification has four components.

1. An introduction that declares the sort (the type name) of the entity being specified. A sort is the name of a set of objects with common characteristics. It is similar to a type in a programming language. 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.

2. 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.

3. 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 sort of operation results are described in the signature.

4. The axioms part defines the semantics of the operations by defining a set of axioms that characterize the behavior 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 includes the following activities:

1. *Specification structuring* Organize 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. You 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. You should 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 use an example of a simple data structure (a linked list) as shown in Figure 27.7. Linked lists are ordered data structures where each element includes a link to the following element in the structure. I have used a simple list with only a few associated operations so that the discussion here is not too long. In practice, object classes defining a list would probably have more operations

Assume that the first stage of the specification process, 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 uppercase for the specification name (LIST) and lowercase with an initial capital for the sort name (List). As lists are collections of other types, the specification has a generic parameter (Elem). The name Elem can represent any type: integer, string, list, and so on.

In general, for each abstract data type, the required operations should include an operation to bring instances of the type into existence (Create) and to construct the type from its basic elements (Cons). In the case of lists, there should be an operation to evaluate the first list element (Head), an operation that returns

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)

Figure 27.7 A simple list specification

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 (Elem). The 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. Therefore, you must include an 'imports' declaration, declaring that the specification of integer is used in the specification.

To create the specification, you define a set of axioms that apply to the abstract type and these specify its semantics. You define the axioms using the operations defined in the signature part. These axioms specify the semantics by setting out what is always true about the behavior of entities with that abstract type.

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

1. Constructor operation*s* that 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 that evaluate attributes of the sort defined in the specification. Typically, these are given names such as Eval or Get.

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. That is, it may be possible to define them using other constructors and inspection operations. If you define a constructor operation using other constructors, then you only need to define the inspection operations using the primitive constructors.

In the list specification, the constructor operations that build lists are Create, Cons and Tail. The inspection operations are Head (return the value of the first element in the list) and Length (return the number of elements in the list), 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 you do have to define Tail using the primitive constructor operations.

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 that 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. You can check that axiom for Head is correct using the same rewriting technique.

Now let us look at how you can use algebraic specification of an interface in a critical system specification. 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 unique 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 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 27.8.

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 I have defined using Create and Put, and I then use them in other specifications. I don't have space to explain all operations in detail here but I discuss two of them (Occupied and Move). With this information, you should be able to understand the other operation specifications.

1. The Occupied operation takes a sector and a parameter representing the height and checks if any aircraft have been assigned to that height. Its specification states that:

   - In an empty sector (one that has been create by a Create operation) every level is vacant. The operation returns false irrespective of the value of the height parameter.

```
SECTOR
sort Sector
imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfed
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)
```

Figure 27.8 The specification of a controlled sector

- In a non-empty sector (one where there has been previous Put operations) the Occupied operation checks whether the specified height (parameter H) is within 300 metres of the height of aircraft that was last added to the sector by a **Put** operation. If so, that height is already occupied so the value of **Occupied** is true.

- If it is not occupied, 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 that 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.

2. The **Move** operation moves an aircraft in a sector from one height to another. Its specification states that:

    - If a **Move** operation is applied to an empty airspace (the result of **Create**), the airspace is unchanged and an exception is raised to indicate that the specified aircraft is not in the airspace.
    - In a non-empty sector, the operation first checks (using In-space) whether the given aircraft is in the sector. If it is not, an exception is raised. If it is in the sector, the operation checks that the specified height is available (using **Occupied**), raising an exception if there is already an aircraft at that height.
    - If the specified height is available, the **Move** operation is equivalent to the specified aircraft leaving the airspace (so the operation **Leave** is used) and being put into the sector at the new height.

27.3 Behavioural specification

The simple algebraic techniques described in the previous section can be used to describe 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, as they increase in size, I find that algebraic descriptions of system behavior become increasingly 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. You can specify the system operations by defining how they affect the state of the system model. The combination of these specifications defines the overall behavior of the system.

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

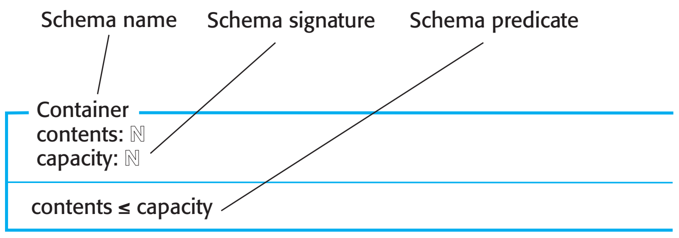

Figure 27.9 The structure of a Z schema

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 (Potter*, et al.*, 1996) and Jacky (Jacky, 1997).

Formal specifications can be difficult and tedious to read especially when they are presented as large mathematical formulae. 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*) that are distinguished from associated text using graphical highlighting. Schemas are used to introduce state variables and to define constraints and operations on the state. Schemas can themselves be manipulated using operations such as schema composition, schema renaming and schema hiding.

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 27.9).

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

To illustrate the use of Z in the specification of a critical system, I have developed a formal specification of the control system of the insulin pump that I introduced in Chapter 3.

Recall that this system monitors the blood glucose level of diabetics and automatically injects insulin as required. 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. I only include some of the schemas defining the system here; the complete specification can be downloaded from the book's website.

To develop a model-based specification, you have to define state variables and predicates that model the state of the system that you are specifying and define invariants (conditions that are always true) over these state variables.

The Z state schema that models the insulin pump state is shown in Figure 27.10. You can see 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 the invariants.

INSULIN_PUMP_STATE

//Input device definition
switch?: (off, manual, auto)
ManualDeliveryButton?: $\mathbb{N}$
Reading?: N
HardwareTest?: (OK, batterylow, pumpfail, sensorfail, deliveryfail)
InsulinReservoir?: (present, notpresent)
Needle?: (present, notpresent)
clock?: TIME

//Output device definition
alarm! = (on, off)
display1!, string
display2!: string
clock!: TIME
dose!: $\mathbb{N}$

// State variables used for dose computation
status: (running, warning, error)
r0, r1, r2: N
capacity, insulin_available : $\mathbb{N}$
max_daily_dose, max_single_dose, minimum_dose: $\mathbb{N}$
safemin, safemax: $\mathbb{N}$
CompDose, cumulative_dose: $\mathbb{N}$

---

r2 = Reading?
dose! $\leq$ insulin_available
insulin_available $\leq$ capacity

// The cumulative dose of insulin delivered is set to zero once every 24 hours
clock? = 000000 $\Rightarrow$ cumulative_dose = 0

// If the cumulative dose exceeds the limit then operation is suspended

cumulative_dose $\geq$ max_daily_dose $\wedge$ status = error $\wedge$
display1! = "Daily dose exceeded"

// Pump configuration parameters
capacity = 100 $\wedge$ safemin = 6 $\wedge$ safemax = 14
max_daily_dose = 25 $\wedge$ max_single_dose = 4 $\wedge$ minimum_dose = 1

display2! = nat_to_string (dose!)
clock! = clock?

Figure 27.10 State schema for the insulin pump

The names declared in the schema are used to represent system inputs, system outputs and internal state variables:

```
RUN
--------------------------------------------------------------
        ΔINSULIN_PUMP_STATE

        switch? = auto
        status = running ∨ status = warning
        insulin_available ≥ max_single_dose
        cumulative_dose < max_daily_dose
    // The dose of insulin is computed depending on the blood sugar level
        (SUGAR_LOW Ú SUGAR_OK ∨ SUGAR_HIGH)
    // 1. If the computed insulin dose is zero, don't deliver any insulin
        CompDose = 0 ⇒   dose! = 0
    ∨
    // 2. The maximum daily dose would be exceeded if the computed dose
    was delivered so the insulin dose is set to the difference between the
    maximum allowed daily dose and the cumulative dose delivered so far
        CompDose + cumulative_dose > max_daily_dose ⇒ alarm! = on ∧
        status' = warning ∧ dose! = max_daily_dose - cumulative_dose
        ∨
    // 3. The normal situation. If maximum single dose is not exceeded
    then deliver the computed dose. If the single dose computed is too high,
    restrict the dose delivered to the maximum single dose
        CompDose + cumulative_dose < max_daily_dose ⇒
          (CompDose ≤ max_single_dose ⇒ dose! = CompDose
          ∨
        CompDose > max_single_dose ⇒  dose! = max_single_dose )
        insulin_available' = insulin_available - dose!
        cumulative_dose' = cumulative_dose + dose!

        insulin_available ≤ max_single_dose * 4 ⇒  status' = warning ∧
        display1! = "Insulin low"

        r1' = r2
        r0' = r1
--------------------------------------------------------------
```

Figure 27.11 The RUN schema

1. System inputs where the convention in Z is for all input variable names to be followed by a ? symbol. I have declared names to model the on/off switch on the pump (switch?), a button for manual delivery of insulin (ManualDeliveryButton?), the reading from the blood sugar sensor (Reading?), the result of running a hardware test program (HardwareTest?), sensors that detect the presence of the insulin reservoir and the needle (InsulinReservoir?, Needle?), and the value of the current time (clock?).

2. System outputs where the convention in Z is for all output variable names to be followed by a ! symbol. I have declared names to model the pump alarm (alarm!), two alphanumeric displays (display1! and display2!), a display of the current time (clock!), and the dose of insulin to be delivered (dose!).

3. State variables that are used for dose computation. I have declared variables to represent the status of the device (status), to hold previous values of the blood sugar level (r0, r1 and r2), the capacity of the insulin reservoir and the amount of insulin currently available (capacity, insulin_available), several variables used to impose limits on the dose of insulin delivered (max_daily_dose, max_single_dose, minimim_dose, safemin, safemax), and two variables used in the dose computation (CompDose and cumulative_dose). The type ℕ means a non-negative number.

   The schema predicate defines invariants that are always true. There is an implicit 'and' between each line of the predicate so all predicates must hold at all times. Some of these predicates simply set limits on the system but others define fundamental operating conditions of the system. These include:

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

2. The cumulative dose is reset at midnight each day. 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>. In this case, <logical expression 1> is 'clock? = 000000' and <logical expression 2> is 'cumulative_dose = 0'.

3. The cumulative dose delivered over a 24-hour period may not exceed max_daily_dose. If this condition is false, then an error message is output.

4. display2! always shows the value of the last dose of insulin delivered and clock! always shows the current clock time.

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 RUN schema, shown in Figure 27.11, models the normal operating condition of the pump.

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. The delta schema (Δ) in the first line in Figure 27.11 illustrates this. The delta means that the state variables defined in INSULIN_PUMP_STATE are in scope as are a set of other variables that represent state values before and after some operation. These are indicated by 'priming' the name defined in INSULIN_PUMP_STATE. Therefore, insulin_available represents the amount of insulin available before some operation and insulin_available′ represents the amount of insulin available after some operation.

Figure 27.12 The SUGAR_OK schema

```
SUGAR_OK
────────────────────────────────────────
    $r2 \geq safemin \wedge r2 \leq safemax$
    // sugar level stable or falling
    $r2 \leq r1 \Rightarrow CompDose = 0$
    $\vee$
    // sugar level increasing but rate of increase falling
    $r2 > r1 \wedge (r2-r1) < (r1-r0) \Rightarrow CompDose = 0$
    $\vee$
    // sugar level increasing and rate of increase increasing compute dose
    // a minimum dose must be delivered if rounded to zero
    $r2 > r1 \wedge (r2-r1) \geq (r1-r0) \wedge (round\ ((r2-r1)/4) = 0) \Rightarrow$
                $CompDose = minimum\_dose$
    $\vee$
    $r2 > r1 \wedge (r2-r1) \geq (r1-r0) \wedge (round\ ((r2-r1)/4) > 0) \Rightarrow$
                      $CompDose = round\ ((r2-r1)/4)$
────────────────────────────────────────
```

Figure 27.11 The RUN schema defines the operation of the system by specifying a set of predicates that are true in normal system use. Of course, these are in addition to the predicates defined in the INSULIN_PUMP_STATE schema that are invariant (always true). This schema also shows the use of a Z feature — schema composition — where the schemas SUGAR_LOW, SUGAR_OK and SUGAR_HIGH are included by giving their names. Notice that these schemas are 'ored' so that there is a schema for each of three possible conditions. The ability to compose schemas means that you can break down a specification into smaller parts in the same way that you can define functions and methods in a program.

I won't go into the details of the RUN schema here but, in essence, it starts by defining predicates that are true for normal operation. For example, it states that normal operation is only possible when the amount of insulin available is greater than the maximum single dose that may be delivered. Three schemas are then 'ored' that represent different blood sugar levels and, as we shall see later, these define a value for the state variable CompDose.

The value of CompDose represents the amount of insulin that has been computed for delivery, based on the blood sugar level. The remainder of the predicates in this schema define various checks to be applied to ensure that the dose actually delivered (dose!) follows safety rules defined for the system. For example, one safety rule is that no single dose of insulin may exceed some defined maximum value.

Finally, the last two predicates define the changes to the value of insulin_available and cumulative_dose. Notice how I have used the 'primed' version of the names here.

The final schema example given in Figure 27.12 defines how the dose of insulin is computed assuming that the level of sugar in the diabetic's blood lies within some safe zone. In these circumstances, insulin is only delivered if the blood sugar level is rising and the rate of change of blood sugar level is increasing. The other schemas SUGAR_LOW and SUGAR_HIGH define the dose to be delivered if the sugar level is outside the safe zone. The predicates in the schema are as follows:

1. The initial predicate defines the safe zone, that is, r2 must lie between safemin and safemax.
2. If the sugar level is stable or falling, indicated by r2 (the later reading) being equal to or less than r1 (an earlier reading) then the dose of insulin to be delivered is zero.
3. If the sugar level is increasing (r2 > r1) but the rate of increase is falling then the dose to be delivered is zero.
4. If the sugar level is increasing and the rate of increase is stable then a minimum dose of insulin is delivered.
5. If the sugar level is increasing and the rate of increase is increasing, then the dose of insulin to be delivered is derived by applying a simple formula to the computed values.

I don't model the temporal behavior 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.

KEY POINTS

- Methods of formal system specification complement informal requirements specification techniques. They may be used with a natural language requirements definition to clarify any areas of potential ambiguity in the specification.
- 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 cost-effective 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 behavioral specification.
- You define the operations in a model-based specification by defining pre- and post-conditions on the system state.

FURTHER READING

'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)

'Correctness by construction: Developing a commercially secure system'. A good description of how formal methods can be used in the development of a security-critical system. (A. Hall and R. Chapman, *IEEE Software*, 19(1), January 2002)

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.

EXERCISES

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

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

27.3 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.

27.4 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.

27.5 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 10.8 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.

27.6 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.

27.7 Modify the insulin pump schema, shown in Figure 27.10, to add a further safety condition that the ManualDeliveryButton? can only have a non-zero value if the pump switch is in the manual position.

27.8 Write a Z schema called SELF_TEST that tests the hardware components of the insulin pump and sets the value of the state variable HardwareTest?. Then modify the RUN schema to check that the hardware is operating successfully before any insulin is delivered. If not, the dose delivered should be zero and an error should be indicated on the insulin pump display.

27.9 Z supports the notion of sequences where a sequence is like an array. For example, for a sequence S, you can refer to its elements as S[1], S[2], and so on. It also allows you to determine the number of elements in a sequence using the # operator. That is, if a sequence S is [a, b, c, d] then #S is 4. You can add an element to the end of a sequence S by writing S + a, and to the beginning of the sequence by writing a + S. Using these constructs, write a Z specification of the LIST that is specified algebraically in Figure 27.7.

27.10 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 manager rejects your suggestion. You think his reasons are weak and based on prejudice. Is it ethical to develop the system using methods that you think are inadequate?

REFERENCES

Bolognesi, T. and Brinksma, E. (1987). 'Introduction to the ISO specification language LOTOS'. *Computer Networks,* **14** (1), 25-59.

Cohen, B., Harwood, W. T. and M.I., J. (1986). *The Specification of Complex Systems*. Wokingham: Addison-Wesley.

Dehbonei, B. and Mejia, F. (1995). 'Formal development of safety-critical software systems in railway signalling'. In *Applications of Formal Methods*. Hinchey, M. and Bowen, J. P. (ed.). London: Prentice-Hall. 227-52.

Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y. and Hamilton, D. (1998). 'Experiences using Lightweight Formal Methods for Requirements Modeling'. *IEEE Trans. on Software Eng.,* **24** (1), 4-14.

Futatsugi, K., Goguen, J. A., Jouannaud, J. P. and Meseguer, J. (1985). 'Principles of OBJ2'. *12th ACM Symp. on Principles of Programming Languages*, New Orleans: ACM Press. 52-66.

Guttag, J. (1977). 'Abstract Data Types and the Development of Data Structures'. *Comm. ACM,* **20** (6), 396-405.

Guttag, J., Horning, J., Garland, S., Jones, K., Modet, A. and Wing, J. (1993). *Larch: Languages and Tools for Formal Specification*. Heidleberg: Springer-Verlag.

Hall, A. (1990). 'Seven Myths of Formal Methods'. *IEEE Software,* **7** (5), 11-20.

Hall, A. (1996). 'Using Formal methods to Develop an ATC Information System'. *IEEE Software,* **13** (2), 66-76.

Hall, A. and Chapman, R. (2002). 'Correctness by Construction: Developing a Commercially Secure System'. *IEEE Software,* **19** (1), 18-25.

Hayes, I. (1987). *Specification Case Studies*. London: Prentice-Hall.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. London: Prentice-Hall.

Jacky, J. (1995). 'Specifying a safety-critical control system'. *IEEE Trans. on Software Eng.,* **21** (2), 99-106.

Jacky, J. (1997). *The Way of Z: Practical Programming with Formal methods*. Cambridge: Cambridge University Press.

Jacky, J., Unger, J., Patrick, M. and Resler, R. (1997). 'Experience with Z: Developing a Control program for a Radiation Therapy Machine'. *Proc. ZUM'97*, Reading: Springer.

Jones, C. B. (1980). *Software Development - A Rigorous Approach*. London: Prentice-Hall.

Jones, C. B. (1986). *Systematic Software Development using VDM*. London: Prentice-Hall.

Liskov, B. and Guttag, J. (1986). *Abstraction and Specification in Program Development*. Cambridge, Mass.: MIT Press.

Neil, M., Ostrolenk, G., Tobin, M. and Southworth, M. (1998). 'Lessons from Using Z to Specify a Software Tool'. *IEEE Trans. on Software Eng.,* **24** (1), 15-23.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. New York: McGraw-Hill.

Potter, B., Sinclair, J. and Till, D. (1996). *An Introduction to Formal Specification and Z*. London: Prentice Hall.

Prowell, S. J., Trammell, C. J., Linger, R. C. and Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.

Spivey, J. M. (1990). 'Specifying a Real-Time Kernel'. *IEEE Software,* **7** (5), 21-8.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual, 2nd edition*. London: Prentice-Hall.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.

Wordsworth, J. B. (1991). 'The CICS application programming interface definition'. *Z User Workshop*, Oxford: Berlin: Springer.