Model-based Specification

11

Objectives

- To introduce an approach to formal specification which is based on developing a mathematical model of a software system.
- To present some features of the Z specification language which is used to specify formal state models of a system and operations on that state.
- To illustrate the Z specification process using several small examples.
- To show how incremental specifications can be developed using Z schemas.

Contents

Model-based specification is an approach to formal specification where the system specification is expressed as a system state model. This state model is constructed using well-understood mathematical entities such as sets and functions. System operations are specified by defining how they affect the state of the system model.

The most widely used notations for developing model-based specifications are VDM (Jones, 1980, 1986) and Z (Hayes, 1987; Spivey, 1992). I use Z (pronounced Zed, not Zee) for describing this approach here. This notation is based on typed set theory. Systems are therefore modelled using sets and relations between sets. However, Z has augmented these mathematical concepts with constructs which specifically support formal software specification.

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.

In an introduction to model-based specification, I can only give an overview of how a specification can be developed. It is not even possible to introduce all of Z. 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 (Diller, 1994) and Wordsworth (Wordsworth, 1992). Hayes (Hayes, 1987) describes a number of case studies where Z has been used and a reference manual for the language has been published (Spivey, 1992).

11.1 Z schemas

In Chapter 10, I introduced the notion of incremental specification where formal specifications are constructed from simpler specifications. Z incorporates excellent support for incremental specification. Specifications are built from components called *schemas*. 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 11.1).

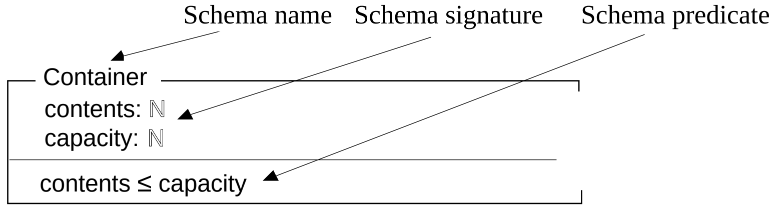


Figure 11.1 A Z schema specifying a container

```
── Indicator ──────────────────────────────
│ light: {off, on}
│ reading: ℕ
│ danger_level: ℕ
├──────────────────────────────────────────
│ light = on ⇔ reading ≤ danger_level
└──────────────────────────────────────────
```

Figure 11.2 The specification of an indicator

The schema is given a meaningful name which is used to refer to it in other parts of the specification. The schema signature declares the names and types of the entities introduced in the schema. In Figure 11.1 the signature introduces two state variables. These are contents and capacity which are modelled as natural numbers (indicated by ℕ). A natural number is an integer that is greater than or equal to zero. These partially define a container which can holds a discrete quantity of something.

The schema predicate defines relationships between the entities in the signature by stating a logical expression which must always be true (an invariant). The predicate states the obvious fact that the contents of the container cannot exceed its capacity. This specification says nothing about the size of the container or what the container is intended to hold. The definition of contents and capacity as natural numbers states that the container must hold a discrete amount of contents.

The specification in Figure 11.1 is a building block which can be used in further specifications. Figure 11.2 shows a specification of another building block that might be associated with a container to provide information about its contents.

The indicator specified in Figure 11.2 introduces three entities namely light (modelled by the values off and on), reading and danger_level (modelled as natural numbers). Both light and reading would have some physical manifestation in the real system (a warning lamp and a dial, perhaps) which provides an operator with information about the system.

The symbol ⇔ in the predicate part can be read as 'if and only if'. The predicate therefore specifies that the light should be on if and only if reading is less than or equal to danger_level. That is, a 'low-contents' warning is signalled. At this stage, danger_level is not defined.

Given the specification of an indicator and a container, they can be combined (Figure 11.3) to define a storage tank with some capacity and an indicator light. The combined specification includes all the state variable declarations and predicates of the included specifications. Thus, Storage_tank combines the signatures of Container and Indicator and their predicates. These are combined with any new signatures and predicates introduced in the specification. Predicates are implicitly anded when schemas are composed so must all hold for the schema invariant to be true.

```
── Storage_tank ───────────────────────────
│ Container
│ Indicator
├──────────────────────────────────────────
│ reading = contents
│ capacity = 5000
│ danger_level = 50
└──────────────────────────────────────────
```

Figure 11.3 The specification of a storage tank

```
── Storage_tank ──────────────────────────
│ contents: ℕ
│ capacity: ℕ
│ reading: ℕ
│ danger_level: ℕ
│ light: {off, on}
├─────────────────────────────────────────
│ contents ≤ capacity
│ light = on ⇔ reading ≤ danger_level
│ reading = contents
│ capacity = 5000
│ danger_level = 50
└─────────────────────────────────────────
```

Figure 11.4 Expanded specification of a storage tank

Storage_tank has three associated predicates which define constraints on the state variables introduced in the schemas Container and Indicator. In Z, writing predicates on separate lines means that they are separated by an implicit 'and'. Thus the predicate can be read as "reading equals contents and capacity equals 5000 and danger_level equals 50". Predicates may also be written on the same line separated by an 'and' symbol ( ).

Including schemas in another schema as shown in Figure 11.3 is equivalent to merging these schemas (Figure 11.4). There is some redundancy here in that reading and contents represent the same thing; this results from the use of generalised schema building blocks. Z includes facilities for variable renaming which could remove this redundancy but I do not cover these constructs here.

The examples of schemas which we have seen so far show the specification of system state and the constraints or state invariants. Operations can be specified using schemas by defining their effect on the system state. That is, you define the state after the operation in terms of the state before the operation and the operation parameters.

Z uses various conventions to identify particular types of schema and state variable used in operation specification:

1. If any variable name, N, is followed by ' e.g. N', this means that it represents the value of the state variable N after the operation. In Z terminology, N is *decorated with* a dash.
2. If a schema name is decorated with ', this introduces the dashed values of all names defined in the specification together with the invariant applying to these values.
3. If a variable name is decorated with !, this means that it is an output e.g. 'message!'.
4. If a variable is decorated with ?, this means that it is an input e.g. 'amount?'.
5. If a schema name is prefixed with the Greek character Xi (Ξ), this means that dashed versions of the variables defined in the named schema are introduced. For all variable names introduced in the schema, the values of corresponding dashed names are the same. That is, the values of state variables are not changed by the operation.

```
── Fill-OK ──────────────────────────────
   Δ Storage_tank
   amount?: ℕ
├────────────────────────────────────
   contents + amount? ≤ capacity
   contents' = contents + amount?
└─────────────────────────────────────────
```

Figure 11.5 A partial specification of the fill operation

6. If a schema name is prefixed with the Greek character Delta (Δ), this implies that values of one or more state variables will be changed by the operation where that schema is introduced. For all variable names introduced in the named schema, corresponding dashed names are also introduced and may be referenced in operations.

Figure 11.5 shows part of the specification of the fill operation which adds an amount to a tank. The schema name is prefixed with Delta, indicating that the operation changes the state. The amount to be added to the tank is an input. The predicate associated with the operation specifies that the state is changed by the operation if there is enough capacity in the tank.

The predicate for Fill-OK specifies that the contents after completion of the operation (referenced as contents') is equal to the sum of the contents before the operation and the amount added to the tank. This is only true if adding the specified amount does not exceed the capacity of the tank. This is precluded because of the predicates defined in Container. If the addition of the specified amount would cause the tank to overflow, the operation is undefined.

A convention in writing Z specifications of operations is that they are specified in parts. The first schema defines the 'correct' operation. Following schemas define what should happen in exceptional situations. These schemas are then combined using a disjunction (or) operator to specify the operation completely.

Figure 11.6 is a specification of what should happen if adding the specified amount exceeds the capacity of the tank. In this situation, nothing is added to the tank and a warning message is output.

Note the use of the Xi schema here indicating that the values of state variables are not changed. The predicate associated with OverFill is true when the capacity of the tank is less than the current contents plus the amount to be added. Nothing is added to the tank if there is not enough room to add all the specified amount. A message 'Insufficient tank capacity - Fill cancelled' is output.

To complete the specification of the fill operation, Fill-OK and OverFill must be combined using a disjunction (or) operator (Figure 11.7). The effect of this

```
── OverFill ─────────────────────────────
   Ξ  Storage-tank
   amount?: ℕ
   r!: seq CHAR
├────────────────────────────────────
   capacity < contents + amount?
   r! = "Insufficient tank capacity - Fill cancelled"
└─────────────────────────────────────────
```

Figure 11.6 Further specification of the fill operation

Figure 11.7 The complete specification of the fill operation

```
─ Fill ─────────────────────────────
  Fill-OK     OverFill
────────────────────────────────────
```

operator is to merge the signatures of Fill-OK and OverFill. These are identical in this case. The predicate parts are independent and are separated by an or operator ( ). Therefore either the predicate in Fill-OK or the predicate in Overfill must be true.

When schemas are very short, as in Figure 11.7, they may be written as text without the normal graphical highlighting. I use this form of schema where appropriate to save space. The complete specification of the fill operation could have been written:

Fill : Fill-OK   Overfill

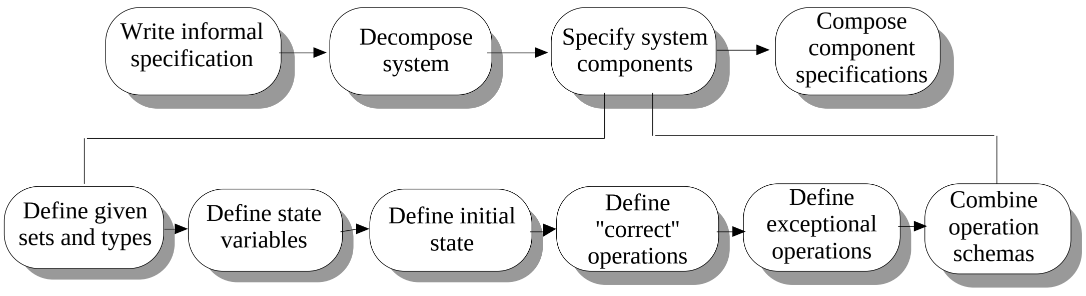
11.2 The Z specification process

There are obviously many different processes which can be used to construct a formal model of a system. The formal specification of a non-trivial system or a sub-system interface is large and complex. An incremental approach must be used where individual system components are specified. These specification fragments are then composed to form the complete specification.

Figure 11.8 illustrates this process and shows the steps involved in constructing the specification for each component.
I illustrate this process by developing part of the specification for a data dictionary like that discussed in Chapter 6. In this example, data dictionaries were used to hold information about the components of semantic data models. Data dictionaries are information systems which maintain details of the names used in some design.

The data dictionary described in Chapter 6 has four fields:

1. *Item name* Holds the name of the entry.



Write informal specification
Decompose system
Specify system components
Compose component specifications
Define given sets and types
Define state variables
Define initial state
Define "correct" operations
Define exceptional operations
Combine operation schemas

Figure 11.8 A process to develop a Z specification

2. *Description* Holds a description of the entry.
3. *Type* Holds the type of the entry. For this example, I assume that only types used in semantic data models will be allowed.
4. Date Holds the date of creation of the entry.

The operations which I define are Add, which adds an entry to the dictionary, Delete, which removes an entry, Lookup, which returns the information associated with a given entry and Replace, which replaces one entry with another.

After constructing the informal specification, the next stage of the process is to define the types used. Types in Z are defined as sets. These sets may be defined explicitly or may be specified as *given sets*.

Given sets are incomplete type definitions where only the type name need be defined. Its details are not included in the specification. This facility is included in Z because it is sometimes appropriate to delay some details until system design or implementation.

Given sets are introduced by enclosing the type name in square brackets. In this case, the given sets are:

[NAME, DATE]

The type field in the data dictionary is restricted to types used in a semantic data model. These are defined by enumerating the values of the set members:

Sem_model_types = { relation, entity, attribute }

Schemas in Z may also be used as a basis for type definition. They are like a record type in Pascal or Ada or a structure in C. The state variables defined in the schema are referenced using dot notation in the same way as the fields of a record. Therefore, to access the name capacity defined in the schema Container, you write Container.capacity.

This type of schema is used to define a type representing an entry in the data dictionary (Figure 11.9). In this type definition, the description associated with a name is defined as a sequence of characters. A sequence is a collection of elements where each element is referenced by its position in the sequence. In this case, the predicate states that there is an arbitrary limit of 2000 characters on the length of the description. Sequences are described in Section 11.3.

```
┌─ DataDictionaryEntry ──────────────────┐
│ entry: NAME
│ desc: seq char
│ type: Sem_model_types
│ creation_date: DATE
├────────────────────────────────────────
│ #description ≤ 2000
└────────────────────────────────────────┘
```

Figure 11.9 A schema defining a data dictionary entry

The next stage of the specification process is to define the data dictionary and its initial state. In this case, functions are used to specify the data dictionary.

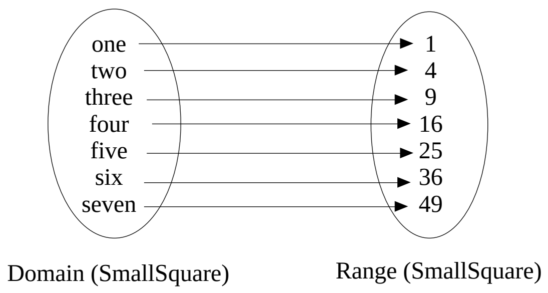


Figure 11.10 The function SmallSquare

Functions or mappings are one of the most commonly used constructs in model-based specifications. In programming languages, a function is an abstraction over an expression. When provided with an input, it computes an output value based on the value of the input. Z functions are similar in effect. However, it is usually more convenient to think of them as a set of relationships between 2 sets called the *domain* and the *range* of the function. Figure 11.10 illustrates this for a simple function called SmallSquare. It relates the identifiers one to seven to members of a set of natural numbers. These are the squares of the numbers represented by the identifiers.

In Z, functions can be defined by listing the set of mappings from values in the domain to values in the range:

SmallSquare = {one 2 1, two 2 4, three 2 9, four 2 16, five 2 25,
six 2 36, seven 2 49 }

The domain of a function (written dom f in Z, where f is the function name) is the set of inputs over which the function has a defined result. The range of a function (written rng f in Z) is the set of results which the function can produce. If an input i is in the domain of some function f (i dom f), the associated result may be specified as f (i), that is, f (i) rng f. For example, in the function SmallSquare, SmallSquare (two) = 4, SmallSquare (five) = 25 and so on.

A function is a partial function if its input is a member of some set T but its domain (those inputs which produce a result) is a subset of T. For example, a partial function f may accept any number between 1 and 50 as an input but may only produce a result if the input is a multiple of 7. The domain of f is therefore (7, 14, 21, 28, 35, 42, 49).

Functions allow one value to be mapped onto another. They can therefore be used to define a data dictionary where a name is mapped onto a data dictionary entry. A schema defining the data dictionary is shown in Figure 11.11. A partial function (indicated by the tagged arrow) is used in this example; not all sequences of characters have associated entries in the data dictionary. Enclosing the schema name DataDictionaryEntry in braces means that the range of DataDictionary is a set of values of type DataDictionaryEntry.

```
┌ DataDictionary ──────────────────────
│
│ DataDictionaryEntry
│ ddict:  NAME ⇸ { DataDictionaryEntry }
└──────────────────────────────────────
```

Figure 11.11 A schema defining a data dictionary as a partial function

The schema DataDictionary defines ddict to be a partial function from NAME to DataDictionaryEntry. Given a name, the associated information such as desc, type and creation_date can be discovered. Sets may not have duplicate members so this definition ensures that names in the data dictionary are unique. However, it does not specify that the data dictionary must be ordered.

The next stage in the specification process is to define the initial state of the data dictionary. We do this by defining a schema that incorporates the schema DataDictionary'. This means that we are interested in the state of the data dictionary after some initialisation has been applied. Its state before this is of no interest. Figure 11.12 shows that the initial value of data dictionary is where this symbol represents the empty set. Initially, there are no entries in the data dictionary.

Now consider the informal definition of operations on the data dictionary:

1. Add This operation takes a name and a data dictionary entry as parameters. If the name is not in the data dictionary, an entry is made in the dictionary.

2. Delete Given a name, the entry associated with that name is deleted from the data dictionary.

3. Lookup Given a name, this value of this operation is the data dictionary entry associated with that name.

4. Replace Given a name and a data dictionary entry, the operation replaces the existing entry with the given name with the new entry.

For the Add operation, an error message is output if the name has already been entered in the data dictionary. For other operations, an error message is output if the name is not in the data dictionary and the data dictionary is unchanged.

The specification of Add and Lookup uses Z constructs which have already been discussed. Figure 11.13 shows the partial specification of the 'correct' operations. In the case of Add, the operation is defined when the name is not in the data dictionary, that is, when the name is not a member of the domain of ddict. Lookup is defined when the name is in the data dictionary.

```
┌ Init-DataDictionary ─────────────────
│ DataDictionary'
├──────────────────────────────────────
│ ddict' =
└──────────────────────────────────────
```

Figure 11.12 The initial state of the data dictionary

```
── Add ──────────────────────────────────────┐
  Δ DataDictionary
  name?: NAME
  entry?: DataDictionaryEntry
─────────────────────────────────────────────
  name?    dom ddict
  ddict' = ddict    { name? 2  entry?}
─────────────────────────────────────────────┘
```

```
── Lookup ───────────────────────────────────┐
  Ξ DataDictionary
  name?: NAME
  entry!: DataDictionaryEntry
─────────────────────────────────────────────
  name?    dom ddict
  entry! = ddict (name?)
─────────────────────────────────────────────┘
```

Figure 11.13 A partial specification of Add and Lookup operations

Note the use of the Delta schema in Add indicating that the operation causes a state change and the Xi schema in Lookup, indicating that no state change occurs. For the Add operation, the state of ddict after the operation is the union of its state before the operation and a mapping from the input name to a data dictionary entry. For the lookup operation, the result (entry!) is the data dictionary entry associated with name? in ddict.

Once the normal mode of operation has been specified, the next stage is to define operations where some exception occurs (Figure 11.14). In the Add operation, an error state occurs when the name has already been added to the data dictionary. In the Lookup operation, the error state occurs when the given name is not in the data dictionary.

Finally the schemas are combined to complete the definition of the Add and Lookup operations. To save space, I use the linear form of a schema definition here.

```
── Add_Error ────────────────────────────────┐
  Ξ  DataDictionary
  name?: NAME
  error!: seq char
─────────────────────────────────────────────
  name?    dom ddict
  error! = "Name already in dictionary"
─────────────────────────────────────────────┘
```

```
── Lookup_Error ─────────────────────────────┐
  Ξ DataDictionary
  name?: NAME
  error!: seq char
─────────────────────────────────────────────
  name?    dom ddict
  error! = "Name not in dictionary"
─────────────────────────────────────────────┘
```

Figure 11.14 Add and Lookup specifications with input errors

```
── Replace-OK ──────────────────────────────
 Δ DataDictionary
 name?: NAME
 entry?: DataDictionaryEntry
────────────────────────────────────────────
 name?    dom ddict
 ddict' I  { name? ↦    entry?}
────────────────────────────────────────────
```

Figure 11.15 Partial specification of the Replace operation

Add ≙ Add_OK   Add_Error

Lookup ≙ Lookup_OK   Lookup_Error

The specification of the Replace operation relies on the use of a Z operator called the function over-riding operator. The function overriding operator is, perhaps, best illustrated with a very simple example. Say we have a function which maps names to telephone numbers.

phone = { Ian ↦ 3390, Ray ↦ 3392, Steve ↦ 3427}

Now assume we have another function newphone defined as follows.

newphone = {Steve ↦ 3386, Ron ↦ 3427}

The operation phone   newphone results in the following function:

phone   newphone =
{ Ian ↦ 3390, Ray ↦ 3392, Steve ↦ 3386, Ron ↦ 3427}

The function overriding operator acts like a set union operator if the name is not in the set. If the name is in the set, the name and its associated value are replaced. In the above example, an entry for Ron has been added and the number associated with Steve has been changed.

Figure 11.15 shows the schema defining the Replace operation when the name is in the dictionary.

The complete specification of the Replace operation can be constructed by combining the specification in Figure 11.15 with Lookup_Error. If the name is not in the data dictionary, the result is the same for this operation as it is for Lookup.

Replace ≙ Replace_OK   Lookup_Error

The operation to delete an item from the data dictionary makes use of a special operator acting on functions. This is called the domain subtraction operator, which is written ⩤. Using the above example of telephone numbers, if Ian is to be removed from the domain of phone, this would be written

{Ian} ⩤ phone

The resulting function is:

{Ray ↦ 3392, Steve ↦ 3427}

```
── Delete ──────────────────────────────
 Δ DataDictionary
 name?: NAME
────────────────────────────────────────
 name?    dom ddict
 ddict' =  {name?} ⩤  ddict
────────────────────────────────────────
```

Figure 11.16 Partial specification of the Delete operation

The specification of the delete operation for the data dictionary is shown in Figure 11.16.

The domain subtraction operator removes a member from the domain of a function. In this case, the domain subtraction operation removes the data dictionary entry name from the domain of DataDictionary. Its related data dictionary entry in the range of the function is thus inaccessible. The effect of Delete is therefore to remove the mapping between name? and its associated data dictionary entry.

The partial specification of Delete can be combined with Lookup-error to construct the complete specification:

Delete ≙ Delete_OK   Lookup_Error

This specification of the data dictionary is based on sets which are unordered entities. In practice, a data dictionary must be displayed and is usually stored in alphabetic order. In the next section, I discuss how Z sequences may be used to model ordered collections of items.

11.3 Specifying ordered collections

Ordered collections of items in Z may be specified using a built-in construct called a sequence. I have already used this when modelling a message as a sequence of characters.

Informally, a sequence is a collection where the elements are referenced by their position in the collection. Thus if a sequence is named S, S (1) references the first element in the sequence, S (5) references the fifth element and so on. A sequence can be thought of as a special kind of function where the domain of the function consists of consecutive natural numbers. This is illustrated in Figure 11.17 which illustrates a Z sequence called SqSeq. This is a sequence of the squares of the first seven natural numbers.

The domain of a sequence is naturally ordered. This means that it is straightforward to write a predicate which specifies that the range is also ordered. One way of doing this for a sequence s whose elements are of type T is:

$$i, j: \mathrm{dom}\ s \ ' \ (i < j) \Rightarrow s(i) <_T s(j)$$

The large dot in this predicate can be read as 'it is always the case that'. The entire predicate can therefore be read as "for all values of i and j in the domain of s, it is always the case that if i is less than j then s(i) is less than s (j)". The symbol "$<_T$ " is the less than operation over entities of type T.

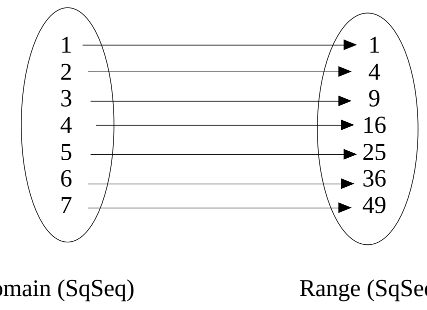


Figure 11.17 A Z sequence

Assume now that a further operation (Extract) on the data dictionary is to be specified. The model of this Extract operation is shown in Figure 11.18. Informally, it may be defined as follows:

> The Extract operation extracts from the data dictionary all entries whose type is the same as the type input to the operation. The extracted list of entries is output in alphabetical order.

The predicate in the Extract operation makes four assertions:

1. For all entries in the data dictionary whose type is in_type?, there is an entry in the output sequence.
2. The type of all members of the output sequence is in_type?. The notation #rep! means the number of members of the sequence called rep!.
3. All members of the output sequence are members of the range of ddict.
4. The output sequence is ordered.

The conjunction of 1 and 2 specifies that there are no members of the output sequence which are not in the data dictionary and that all members of the data dictionary with the given type also appear in the output sequence.

Finally, the complete specification of the data dictionary can be composed from all the above schemas (Figure 11.19).

```
── Extract ──────────────────────────────────────
  DataDictionary
  rep!:  seq {DataDictionaryEntry}
  in_type?: Sem_model_types
─────────────────────────────────────────────────
   n : dom ddict '  ddict(n).type = in_type?     ddict (n)    rng rep!
   i : 1 ≤  i ≤ #rep! '   rep! (i).type = in_type?
   i : 1 ≤  i ≤ #rep! '   rep! (i)   rng ddict
    i, j : dom rep! '   (i < j)       rep.name(i) < NAME rep.name (j)
─────────────────────────────────────────────────
```

Figure 11.18 The specification of the Extract operation

```
┌─ The_Data_Dictionary ──────────────────┐
│
│  DataDictionary
│  Init-DataDictionary
│  Add
│  Lookup
│  Delete
│  Replace
│  Extract
└────────────────────────────────────────┘
```

Figure 11.19 The complete specification of the data dictionary

This brief introduction to model-based specification has only scratched the surface of the technique. There are many other defined language operations. Examples of Z specifications of real systems are given by Earl *et al.* (Earl et al., 1986) who specified part of a software engineering environment. Wordsworth (Wordsworth, 1990) specified a large part of IBM's CICS system and Spivey (Spivey, 1990) has specified a kernel for a real-time system.

Z has been fairly extensively used (mostly in the UK) for formally specifying moderately large systems. As Hall (Hall, 1990) describes, it can be used for specifying object-oriented systems by adopting various conventions of use. Alternatively, object-oriented variants of Z have been proposed. These, along with Hall's approach are summarised by Stepney *et al.* (Stepney et al., 1992). However, a problem with Z is its lack of scoping and structuring facilities. Schemas are useful for defining specification fragments but they are concerned with low-level structuring rather than organising the architecture of a specification.

KEY POINTS

- Model-based specification relies on building a model of the system using mathematical entities such as sets which have a formal semantics. The Z specification language is based on typed sets.
- Z specifications consist of a mathematical model of the system state and a definition of operations on that state.
- A Z specification is presented as a number of schemas where a schema introduces some typed names and defines predicates over these names. Schemas in Z may be distinguished from surrounding text by graphical highlighting.
- Schemas are building blocks which may be combined and used in other schemas. The effect of including a schema A in schema B is that schema B inherits the names and predicates of schema A.
- Operations may be specified in Z by defining their effect on the system state. It is normal to specify operations incrementally and then combine the specification fragments to produce the complete specification.

- Z functions are sets of pairs where the domain of the function is the set of valid inputs. The range is the set of associated outputs. If ordering is important (sets are unordered), sequences can be used as a specification mechanism.

FURTHER READING

There are many books on Z which cover the notation and its use. Those below are good examples of Z texts.

Z: An Introduction to Formal Methods (2nd ed.) This is a good introduction to Z with many examples and a Z reference manual. It covers the required mathematical background, the use of Z in formal verification and presents a number of case studies. (A. Diller, 1994, John Wiley and Sons)

Software Development with Z As well as introducing Z, this book also describes a development process from formal specification to an implementation. (J.B. Wordsworth, 1992, Addison-Wesley)

'Specifying a Real-Time Kernel' This is a good introductory paper on Z for readers who are familiar with real-time systems architecture. It includes a concise summary of Z notation. (J.M. Spivey, *IEEE Software*, 7 (5), September 1990)

EXERCISES

11.1 Explain how the Z's schema combination mechanism may be used to construct complex specifications.

11.2 Modify the specification of a storage tank (Figure 11.4) by adding a fill warning light which indicates when the tank is close to capacity. This should be switched on when the contents is some high percentage of the capacity.

11.3 Write a specification for an operation called Dispense which dispenses a given number of units from a tank.

11.4 Modify the specification of the Fill operation (Figure 11.7) so that it either adds the given amount or fills the tank completely.

11.5 What do you understand by the term *domain* and *range* of a function. Explain how functions may be used in defining keyed data structures such as tables.

11.6 Modify the specification of DataDictionaryEntry (Figure 11.9) so that the length of attribute descriptions is restricted to 500 characters, the description of relations to 1000 characters and the description of entities to 1500 characters.

11.7 Modify the specification of the Extract operation (Figure 11.18) so that it also takes a date as a parameter and only extracts entries from the data description where were entered after that date. You may assume that greater than and less than operators are defined for the type DATE.

11.8 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

```
┌─ Library ──────────────────────
│ stock: ℙ Book
│ onLoan: Book ⇸ Borrower
├────────────────────────────────
│ dom onLoan ⊆ Stock
└────────────────────────────────
```

Figure 11.20 The state space of a library

also derive account information from a central database and update that database on completion of a transaction. Using your personal knowledge of the operation of such machines, write Z schemas defining the state of the system, card validation (where the user's identifier is checked) and cash withdrawal.

11.9 The Z schema shown in Figure 11.20 defines the state space of a lending library. Define an operation called Borrow which takes inputs called reader and book and which defines the effect on the state of a book being borrowed. The notation ℙ S means Powerset S. That is, the type of stock is defined as the set of all sets of books. The notation ⊆ can be read as 'is a subset of'.

11.10 Define two further operations on Library namely New which adds a new book to the current stock and Return which returns a book which has been on loan to the library .