However, you will note that the set operators introduced in this chapter have been described in English. We could have given precise definitions using logic. For example, the following is a definition of set union. For a given type T

$$\forall x : T ; A, B : \mathbb{P} T \bullet x \in A \cup B \Leftrightarrow x \in A \vee x \in B$$

In other words

$$\forall A, B : \mathbb{P} T \bullet A \cup B = \{ x : T \mid x \in A \vee x \in B \}$$

Equality used as a predicate

Exercises 3.6

1. Give logic expressions to define formally the meaning of:
   - (i) set intersection;
   - (ii) set difference;
   - (iii) generalised union.
2. Describe the following situation using the notation covered in this chapter. Assume that you have the type [*PERSON*], the set of all people.
   - (i) People are either women or men, but not both.
   - (ii) A company employs people in three departments: marketing, personnel and production. Each employee is in precisely one of these departments.
   - (iii) Each department has a maximum of 10 staff.
   - (iv) All the staff in marketing are women.
   - (v) The company employs more men than women.
3. Now assume that each employee in the previous question can be in more than one department. Write down expressions for:
   - (i) The number of women who work in all three departments.
   - (ii) The number of men who work in marketing and personnel but not in production.

CHAPTER 4

The structure of a Z specification

Aims

To apply the material of the previous two chapters to the development of Z specifications, to introduce mechanisms for structuring Z specifications, and to illustrate the above with a simple example.

Learning objectives

When you have completed this chapter, you should be able to:

- represent the state of simple systems using sets and logic;
- specify the effect of operations which change or interrogate the state of a system;
- structure your specifications using schemas;
- introduce appropriate exception handling to totalise your operations using the schema calculus;
- determine the conditions necessary for an operation to take place successfully.

4.1 Introduction

In this chapter, we will develop an example to illustrate the 'flavour' of writing specifications with Z. We will introduce most of the notation for structuring Z specifications that will be used for the rest of the book. The specification is for the student badminton club mentioned in Chapter 3, and could be implemented as a computer system or paper records, to keep track of the whereabouts of the club members, add or remove members from the club, etc.

To write a specification in Z, we create a *model* of the required system. The structure, or state, of the system is represented using sets, and the relationships between the elements of the state are expressed using the language of logic. We then use logic to specify operations to change or make queries about the state of the system.

4.2 The system state

Suppose that the student badminton club has the sole use of a hall with a single badminton court. To use the hall, one must be a member of the club. To ensure that everyone gets enough games, there is a limit of 20 people allowed in the hall at any one time. We will construct a model of this system. We begin by identifying the basic types required in our specification; here there is only one:

$[STUDENT]$ the set of all students

We can represent the limit on the number of people allowed in the hall by an *axiomatic description* which is a Z construct for defining a global variable, which is in scope (may be referred to) throughout the specification.

```
| maxPlayers : ℕ
|---------------
| maxPlayers = 20
```

constraint, not an assignment

The top half is a declaration, and the bottom half is an optional predicate specifying a constraint on the values of the variable declared. The constraint chosen here effectively makes *maxPlayers* a global constant.

We are interested in the whereabouts of the members of the badminton club. We can represent this information using two sets of students: *badminton*, the set of all members of the club, and *hall*, the set of all those who are in the hall. The things which must always be true for any values of these sets are:

1. A person in the hall must be a member of the club.

   $hall \subseteq badminton$

2. The number of people in the hall must not exceed *maxPlayers*.

   $\# hall \leqslant maxPlayers$

like integrity constraints

These predicates are called the *state invariants* of the system. If the sets *hall* and *badminton* are to represent a valid state for the system, then the values which they take must be such that they make the state invariant predicates true.

In Z, we combine the declaration of the sets with the predicates constraining their values using a box called a *schema*. The state schema for our system is as follows:

```
┌──── ClubState ────
│ badminton : ℙ STUDENT
│ hall : ℙ STUDENT
├───────────────────
│ hall ⊆ badminton
│ # hall ≤ maxPlayers
└───────────────────
```

state variables

predicates defining invariant properties of the state

The state variables are declared in the top half of the schema box, above the line. *badminton* and *hall* are variables which take values which are sets of students, so their type is the powerset of the type *STUDENT*. The predicates defining the invariant properties of the state are defined in the bottom half. Note that predicates on separate lines are implicitly conjoined to make one predicate; the above could be rewritten as

$$(hall \subseteq badminton) \wedge (\# hall \leqslant maxPlayers)$$

4.3 Operations

The above state schema defines the set of valid states which our system may assume. The system may move from one valid state to another by *operations* which change the values of one or more of the state variables. For this example, we might be interested in operations to add or remove a member to/from the club, or to add or remove a member to/from the hall. Such operations will involve adding or removing elements to/from the two sets which constitute the state variables. However, before we specify any operations, we must learn some new notation.

Inputs and outputs

Operations often require inputs and outputs, and the convention for naming these is that an input identifier is terminated by a ? and an output identifier is terminated by a !.

'Before' and 'after' states

To specify the effect of an operation on the state of the system, we must be able to refer to the state variables both 'before' and 'after' an operation. The convention is that 'before' variables are undecorated, whereas the names of 'after' variables are decorated with primes (dashes). If x is a variable before a given operation, then x' will be the same variable after the operation. For example, the set *badminton* would represent the variable before an operation, and the set *badminton'* would represent the same variable after the operation.

We define the effect of a state-changing operation using:

1. Predicates which state what must be true about the 'before' state of the system and the inputs, if any, in order for the operation to take place. These are known as *preconditions*.
2. Predicates which relate the 'before' state and the inputs, if any, to the 'after' state and the outputs, if any. These are known as *postconditions*, and define the effect of the operation. Note that we specify *what* the result of the operation must be; *how* the operation works is not stated. This is a consideration for those who must implement the specification as a computer program or other system.

Note that the preconditions may be explicitly stated, or they may be implicit, occurring as a consequence of the postconditions and/or the state invariant, in which case we may have to calculate them. See Section 4.13 for more on this.

4.4 Adding a new member

We will now use the above ideas to specify an operation to add a new member to the badminton club. To join the club, a potential member must register with the club secretary, after which the member may go to the hall to play. The potential new member will be an input for the operation, declared as

$$newMember? : STUDENT$$

For this operation, it is important that *newMember?* is not already a member of the club. This leads to the precondition

$$newMember? \notin badminton$$

Note that the state invariant implies that *newMember?* is also not in the hall. The required behaviour for the operation is that *newMember?* be added to the set *badminton* but not to the set *hall*. We can achieve this by placing *newMember?* in a set by itself (a singleton set), and taking the union of this set with the set *badminton*. This leads to the following postconditions:

$$\begin{array}{l} badminton' = badminton \cup \{newMember?\} \\ hall' = hall \end{array}$$

Note that we have explicitly specified that the set *hall* does not change. If we didn't include this condition, we would be underspecifying the operation, that is we would not be stating whether or not *newMember?* is to become an element of *hall*. A person implementing this operation could then make either choice. It is important in general to specify our operations fully; that is, to state what is to happen to all of the state variables.

To complete the definition of the operation in Z, we gather the above declarations and predicates into an *operation schema* which we will call *AddMember*. The declarations part (top half) of the schema must contain the declaration of any input and output variables. As things stand, *AddMember* would also have to contain the 'before' and 'after' versions of the state schema declarations and invariant predicates. This is because all declarations and predicates are local to the schema in which they appear, and are therefore not implicitly available in any other schema. Any object referred to in the predicate part (bottom half) of a schema must either be declared in the top half of that schema or be globally defined.

We could simply copy all the required 'before' and 'after' state declarations and predicates into our *AddMember* schema, as follows:

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\quad AddMember \quad}} \\ badminton : \mathbb{P}\ STUDENT \\ badminton' : \mathbb{P}\ STUDENT \\ hall : \mathbb{P}\ STUDENT \\ hall' : \mathbb{P}\ STUDENT \\ newMember? : STUDENT \\ \hline hall \subseteq badminton \\ \# hall \leqslant maxPlayers \\ hall' \subseteq badminton' \\ \# hall' \leqslant maxPlayers \\ \\ newMember? \notin badminton \\ badminton' = badminton \cup \{newMember?\} \\ hall' = hall \\ \hline \end{array}$$

tedious

However, this could get very tedious, and Z provides the following mechanisms for simplifying the process, thereby making specifications clearer and more succinct.

Schema decoration

Given a schema *S*, the notation *S'* stands for *S* with all of its variables decorated with primes throughout the schema. For example,

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\quad ClubState' \quad}} \\ badminton' : \mathbb{P}\ STUDENT \\ hall' : \mathbb{P}\ STUDENT \\ \hline hall' \subseteq badminton' \\ \# hall' \leqslant maxPlayers \\ \hline \end{array}$$

Schema inclusion

All declarations and predicates from a schema S may be included in a schema T by simply placing the name of S in the declaration part of T:

```
┌─── T ──────────
│ S
│ ⟨declarations for T⟩
├────────────────
│ ⟨predicates for T⟩
└────────────────
```

If the same variable name occurs in both S and T, then it must have the same type in both.

The delta convention

For a given schema S, the notation ΔS represents the schema obtained by including S and S' in an otherwise empty schema. The Δ symbol is not an operator, but simply part of a schema name. However, the convention is universally used.

```
┌─ ΔS ──
│ S
│ S′
└───────
```

We can therefore rewrite the *AddMember* schema as follows:

```
┌─── AddMember ──────────────────
│ Δ ClubState
│ newMember? : STUDENT
├────────────────────────────────
│ newMember? ∉ badminton
│ badminton′ = badminton ∪ {newMember?}
│ hall′ = hall
└────────────────────────────────
```

Short

In general, for a state schema S, the inclusion of ΔS in an operation schema indicates that the operation potentially changes the state, because it brings into scope the 'before' and 'after' versions of the state variables and invariant predicates.

Note that by including the 'before' and 'after' versions of the state invariants, we implicitly introduce further pre- and postconditions which must not be violated by the operation.

Exercise 4.1

Write a schema for the operation *RemoveMember*, which removes a member from the club. Hint: To remove an element from a set, place the element in a singleton set and use the set difference operator \.

4.5 Entering the hall

We will now specify an operation for a student to enter the hall. The student will be an input to the operation.

$$enterer? : STUDENT$$

For the student to enter the hall, s/he must be a member of the club.

$$enterer? \in badminton$$

and must not already be in the hall!

$$enterer? \notin hall$$

Additionally, to ensure that the operation will not violate the state invariant, the number of members already in the hall must be less than *maxPlayers*. To improve clarity, it is often better to make such implicit preconditions, which are consequences of the state invariant, explicit in the operation schema.

$$\# \, hall < maxPlayers$$

This will also make it easier later on to identify and handle exceptions, that is to specify what action is to be taken when the preconditions of the operation are not satisfied.

The above properties constitute the preconditions for the *EnterHall* operation. The effect of the operation, captured by the postconditions, is to add the person to the set *hall*. The set *badminton* is not changed by the operation.

$$hall' = hall \cup \{enterer?\}$$
$$badminton' = badminton$$

The required schema is

```
 ____ EnterHall ______
| Δ ClubState
| enterer? : STUDENT
|______________________
| enterer? ∈ badminton
| enterer? ∉ hall
| # hall < maxPlayers
| hall′ = hall ∪ {enterer?}
| badminton′ = badminton
|______________________
```

Exercise 4.2

Write a schema for the operation *LeaveHall*, which removes a member from the hall.

4.6 The xi convention

For a given state schema S, the notation ΞS represents the schema obtained by including ΔS in an otherwise empty schema together with, for every variable x declared in S, the predicate

$$x = x'$$

In other words, including ΞS in an operation schema makes visible the 'before' and 'after' versions of the declarations and predicates of S, together with the assertion that these variables are not changed by the operation. Again, the Ξ symbol is simply part of a schema name, but the convention is universally used.

For the badminton club example, we have

```
 ____ Ξ ClubState ______
| Δ ClubState
|______________________
| badminton′ = badminton
| hall′ = hall
|______________________
```

4.7 Query operations

Sometimes we wish to specify operations which do not change the state, but output some information about it. For example an operation to output the set of all club members not in the hall:

```
 ____ NotInHall ______
| Ξ ClubState
| outside! : ℙ STUDENT
|______________________
| outside! = badminton \ hall
|______________________
```

use output variables for results of a query

Note that the inclusion of Ξ *ClubState* specifies that the operation does not change the state, and the suffix ! in the variable name *outside*! indicates that this is an output. There is no precondition; this operation may be applied to any state.

Exercise 4.3

Specify an operation which inputs a student and outputs a message stating whether s/he is:

(i) in the hall;
(ii) a member but not in the hall;
(iii) not a member.

You may assume the existence of the free type

$$MESSAGE ::= inHall \mid notInHall \mid notMember$$

4.8 Combining schemas with propositional operators

Two schemas S and T can be combined using any of the following propositional operators:

$S \wedge T$
$S \vee T$
$S \Rightarrow T$ or
$S \Leftrightarrow T$

Each of these defines a schema which merges the declarations from S and T, and whose predicate is

$$P_s \; op \; P_t$$

where P_s is the predicate of S, op is the appropriate propositional operator, and P_t is the predicate of T.

Any variable name occurring in both S and T must have the same type in each.

For example, consider the following schemas:

$$\begin{array}{|l} \multicolumn{1}{l}{A} \\ \hline a : \mathbb{Z} \\ \hline a = 42 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{B} \\ \hline a, b : \mathbb{Z} \\ \hline a = b + 2 \\ b < 10 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{C} \\ \hline b : \mathbb{P}\mathbb{Z} \\ \hline 42 \in b \\ \hline \end{array}$$

$AandB \mathrel{\widehat=} A \wedge B$ is the schema

$$\begin{array}{|l} \multicolumn{1}{l}{AandB} \\ \hline a, b : \mathbb{Z} \\ \hline (a = 42) \wedge ((a = b + 2) \wedge (b < 10)) \\ \hline \end{array}$$

$\widehat=$ is the *schema definition symbol*, which is used to associate a name with a schema.

$AimpliesC \mathrel{\widehat=} A \Rightarrow C$ is the schema

$$\begin{array}{|l} \multicolumn{1}{l}{AimpliesC} \\ \hline a : \mathbb{Z} \\ b : \mathbb{P}\mathbb{Z} \\ \hline (a = 42) \Rightarrow (42 \in b) \\ \hline \end{array}$$

The use of propositional operators with schemas enables us to give more structure to complex specifications by breaking them down into simpler units. For example, the *AddMember* schema requires that the new member joins the club whilst outside the hall, and the only people allowed in the hall are members of the badminton club. Let us picture a different scenario, where the hall has other activities going on, and people other than members of the badminton club may be present in the hall. The set *hall* is still the set of all members of the badminton club who are in the hall, but it is possible for a new member to join either outside or inside the hall. We can represent the joining location as a free type, and define two operation schemas, *AddMemberInHall* and *AddMemberOutHall*, to specify the two possible cases as follows:

not anybody in the a [illegible] hall

people who want to join the club may be physically in the hall or outside of it when they join.

LOCATION ::= *inside* | *outside*

$$\begin{array}{|l} \multicolumn{1}{l}{AddMemberInHall} \\ \hline \Delta\, ClubState \\ newMember? : STUDENT \\ where? : LOCATION \\ \hline where? = inside \\ newMember? \notin badminton \\ \#\, hall < maxPlayers \\ badminton' = badminton \cup \{newMember?\} \\ hall' = hall \cup \{newMember?\} \\ \hline \end{array}$$

(physically inside the hall)

LOCATION ::= inside | outside

$$\begin{array}{|l} \multicolumn{1}{l}{AddMemberOutHall} \\ \hline \Delta\, ClubState \\ newMember? : STUDENT \\ where? : LOCATION \\ \hline where? = outside \\ newMember? \notin badminton \\ badminton' = badminton \cup \{newMember?\} \\ hall' = hall \\ \hline \end{array}$$

We can now define a schema which represents the operation of adding a new member either outside or inside the hall using schema disjunction as follows:

$$AddMemberAnywhere \mathrel{\widehat=} AddMemberInHall \vee AddMemberOutHall$$

Where an operation schema has explicit preconditions, we can specify separate schemas for handling the exception situations where the preconditions of the operation are not satisfied, and use the schema calculus to combine these schemas to define a robust, so-called *total* version of the operation which does something meaningful for any combination of the values of its 'before' state and inputs. An example of this is presented in the next section.

Exercise 4.4

1. Given the above definitions of A, B and C, write down the expansion of the following schema expressions:
   (i) $P \mathrel{\widehat=} A \Leftrightarrow B$
   (ii) $Q \mathrel{\widehat=} A \Rightarrow (A \vee C)$
   (iii) $R \mathrel{\widehat=} A \vee (B \wedge C)$

2. What do you notice about the predicate of the schema $A \wedge B$?
3. Give a definition of Δ *ClubState* using a propositional schema operator.

4.9 Totalising operations

When specifying an operation, it is important to state the action to be taken for all possible values of the state variables and inputs. Such an operation is sometimes referred to as *total*. If the operation specification is not total, then the specification is not complete; we are leaving it up to the implementor of the system to decide what to do in the cases not specified.

In the *AddMember* operation, for example, we have only specified what is to happen if the precondition

$$newMember? \notin badminton$$

is satisfied. If the precondition is not satisfied, the successful case of the operation must not happen, the state must not change, and we will want the system to produce an appropriate exception message stating the reason for not doing the operation. A common way of dealing with this in Z is to represent the set of 'outcome' messages as a free type.

$$MESSAGE ::= success \mid isMember$$

We can now write a schema to specify the action for each possible outcome. The situation where a potential new member is *already* a member gives the following schema. Note the inclusion of Ξ *ClubState* to indicate that this schema does not change the state.

```
── IsMember ──────────
 Ξ ClubState
 newMember?: STUDENT
 outcome! : MESSAGE
──────────────────────
 newMember? ∈ badminton
 outcome! = isMember
──────────────────────
```

error case of "AddMember"

For consistency, we may also wish to produce a message when the operation executes successfully. This message can be represented by the schema

```
── SuccessMessage ────
 outcome! : MESSAGE
──────────────────────
 outcome! = success
──────────────────────
```

Schemas may also be written in an equivalent *horizontal form*, where declarations and predicates are enclosed in square brackets and separated by |. If there is more than one declaration or predicate, they are separated by semicolons. The horizontal form is appropriate for small schemas such as *SuccessMessage*:

$$SuccessMessage \mathrel{\widehat{=}} [outcome! : MESSAGE \mid outcome! = success]$$

The complete specification of the *AddMember* operation can now be defined by combining the various schemas using the propositional operators of the schema calculus encountered above.

$$TotalAddMember \mathrel{\widehat{=}} (AddMember \wedge SuccessMessage) \vee IsMember$$

This is a schema which specifies the outcome for any possible values of the 'before' state and inputs, and outputs the appropriate message. Note that some operations have no preconditions and therefore such exception handling is unnecessary. For example, the *NotInHall* operation in Section 4.7 above may be applied to any state and is therefore already a total operation.

For operations with more than one precondition, the neatest way of defining the total operation is to write separate schemas to handle each of the precondition exceptions, and then combine the schemas using schema disjunction as above. However, if more than one error occurs simultaneously, the operation may be non-deterministic in that it does not specify which of the errors are to be reported. This could be overcome by including error messages not just for every individual error condition, but also for all combinations of error conditions. However, this would become extremely tedious and lead to unnecessarily large specifications. A better solution is to document the non-determinism in the specification document, leaving it up to the implementor of the specification to decide how to handle multiple error conditions.

The use of the delta and xi conventions, and the notation for combining schemas using propositional operators, allow us to produce modular specifications which are clearer and more succinct. To demonstrate this, you might try expanding the *TotalAddMember* operation as a single schema!

Exercise 4.5

Extend the specification to totalise the *EnterHall* operation. Note that in this case there are three preconditions to consider, the exception to each of which should be handled by a separate schema.

4.10 The initial state

We have created a model of a system by defining a set of valid states and a set of valid operations, some of which cause the system to move from one state to another. However, we have not specified in which valid state the system must start. An appropriate initial state for this system would be one in which there are no members in the club; that is, in which the sets *badminton* and *hall* are empty.

```
── InitClubState ──
│ ClubState'
├──────────────
│ badminton' = { }
│ hall' = { }
└──────────────
```

Note the inclusion of *ClubState'*; the convention is for initial state variables to be decorated. The initial state is effectively a special 'after' state, without a corresponding 'before' state. We may think of it as the result of an operation to create our system from nothing, or to reset the system from any state.

We are obliged to verify that the proposed initial state is indeed a valid state; that is, the state invariant property is not violated. A brief inspection in this case confirms that this is so, because

$$\{\} \subseteq \{\}$$

p. 33
hall ⊆ badminton
# hall ≤ maxplayers

and

$$\forall n : \mathbb{N} \bullet \#\{\} \leqslant n$$

However, for some specifications, the proof that the proposed initial state is valid is not so straightforward.

4.11 Renaming

Schema variables may be renamed to produce a new schema, by writing the necessary changes in square brackets after the schema name. In general, for a schema *S*, the expression

$$S[x / a, y / b, z / c]$$

represents *S* with all instances of the name *a* replaced by *x*, *b* by *y* and *c* by *z*.

For example, given the badminton club state schema as before

```
── ClubState ──
│ badminton : ℙ STUDENT
│ hall : ℙ STUDENT
├──────────────
│ hall ⊆ badminton
│ # hall ≤ maxPlayers
└──────────────
```

the schema

$$FootyClub \mathrel{\widehat=} ClubState\ [football / badminton, pitch / hall]$$

is the schema

```
── FootyClub ──
│ football : ℙ STUDENT
│ pitch : ℙ STUDENT
├──────────────
│ pitch ⊆ football
│ # pitch ≤ maxPlayers
└──────────────
```

4.12 Hiding

The schema hiding operator \ takes a schema and a list of variables declared in the schema, and hides the variables in the schema by removing them from the schema declarations and existentially quantifying them in the schema predicates. In general, for a schema *S*, the expression

$$S \setminus (x, y, z)$$

represents *S* with the declarations of variables *x*, *y* and *z* removed and existentially quantified.

Given the schemas *A* and *B* from Section 4.8,

```
── A ──          ── B ──
│ a : ℤ          │ a, b : ℤ
├──────          ├──────
│ a = 42         │ a = b + 2
└──────          │ b < 10
                 └──────
```

the schema $A \setminus (a)$ would be a schema with an empty signature and a predicate which is always true!

The schema $HideB \mathrel{\widehat{=}} B \setminus (b)$ would be as follows: i.e. $\exists a : \mathbb{Z} \bullet a = 42$

$$\begin{array}{|l}
\textit{HideB} \\
\hline
a : \mathbb{Z} \\
\hline
\exists b : \mathbb{Z} \bullet \\
\quad a = b + 2 \\
\quad \land b < 10 \\
\hline
\end{array}$$

In fact, the predicate now simply states that there is a number less than 12 which equals a, so the schema simplifies to

$$\begin{array}{|l}
\textit{HideB} \\
\hline
a : \mathbb{Z} \\
\hline
a < 12 \\
\hline
\end{array}$$

Given the *AddMember* schema as before,

$$\begin{array}{|l}
\textit{AddMember} \\
\hline
\Delta\, \textit{ClubState} \\
\textit{newMember}? : \textit{STUDENT} \\
\hline
\textit{newMember}? \notin \textit{badminton} \\
\textit{badminton}' = \textit{badminton} \cup \{\textit{newMember}?\} \\
\textit{hall}' = \textit{hall} \\
\hline
\end{array}$$

the definition

$$\textit{AddWho} \mathrel{\widehat{=}} \textit{AddMember} \setminus (\textit{newMember}?)$$

is the schema

$$\begin{array}{|l}
\textit{AddWho} \\
\hline
\Delta\, \textit{ClubState} \\
\hline
\exists\, \textit{newMember}? : \textit{STUDENT} \bullet \\
\quad \textit{newMember}? \notin \textit{badminton} \\
\quad \land\, \textit{badminton}' = \textit{badminton} \cup \{\textit{newMember}?\} \\
\quad \land\, \textit{hall}' = \textit{hall} \\
\hline
\end{array}$$

usefulness of this example?

Hiding may also be achieved using *projection*. See Spivey (1992) for further information.

4.13 Operation schema preconditions

When we specify an operation, it is important to know the combinations of the 'before' state and input variables for which the operation may be applied. In other words, the combinations of 'before' state and input variables for which there are values of the 'after' state and output variables which satisfy the operation's predicates. These combinations may be defined from the operation schema by hiding all 'after' state variable and outputs.

The schema precondition operator, denoted by pre, is used to calculate the precondition of an operation schema. For an operation schema S, the expression

$$\text{pre } S$$

is the precondition schema of S, which is S with all 'after' state variables and output variables hidden. For example, consider the schema *SuccessAddMember* specifying the successful addition of a member:

$$\textit{SuccessAddMember} \mathrel{\widehat{=}} \textit{AddMember} \land \textit{SuccessMessage}$$

If we expand this schema, making all 'after' state declarations and predicates explicit, we get the following:

$$\begin{array}{|l}
\textit{SuccessAddMember} \\
\hline
\textit{ClubState} \\
\textit{badminton}' : \mathbb{P}\ \textit{STUDENT} \\
\textit{hall}' : \mathbb{P}\ \textit{STUDENT} \\
\textit{newMember}? : \textit{STUDENT} \\
\textit{outcome}! : \textit{MESSAGE} \\
\hline
\textit{hall}' \subseteq \textit{badminton}' \\
\#\, \textit{hall}' \leqslant \textit{maxPlayers} \\
\\
\textit{newMember}? \notin \textit{badminton} \\
\textit{badminton}' = \textit{badminton} \cup \{\textit{newMember}?\} \\
\textit{hall}' = \textit{hall} \\
\textit{outcome}! = \textit{success} \\
\hline
\end{array}$$

ClubState has been included instead of explicitly writing all the 'before' state declarations and predicates, to make the schema a little more succinct. Note the difference between this and $\Delta\, \textit{ClubState}$ or $\Xi\, \textit{ClubState}$.

The precondition schema pre *SuccessAddMember* is the schema

$$\textit{SuccessAddMember} \setminus (\textit{badminton}', \textit{hall}', \textit{outcome}!)$$

defined as follows:

pre *SuccessAddMember*

$ClubState$
$newMember? : STUDENT$

$\exists\, badminton', hall' : \mathbb{P}\ STUDENT;\ outcome! : MESSAGE \bullet$
$\quad hall' \subseteq badminton'$
$\quad \land \#\, hall' \leqslant maxPlayers$
$\quad \land newMember? \notin badminton$
$\quad \land badminton' = badminton \cup \{newMember?\}$
$\quad \land hall' = hall$
$\quad \land outcome! = success$

This can be simplified as follows:

$hall' = hall$

implies that we can remove the quantification of *hall'* and replace all references to *hall'* with *hall*. Furthermore

$\exists\, outcome! : MESSAGE \bullet outcome! = success$

is trivially true, and so can be removed, to give the following:

pre *SuccessAddMember*

$ClubState$
$newMember? : STUDENT$

$\exists\, badminton' : \mathbb{P}\ STUDENT \bullet$
$\quad hall \subseteq badminton'$
$\quad \land \#\, hall \leqslant maxPlayers$
$\quad \land newMember? \notin badminton$
$\quad \land badminton' = badminton \cup \{newMember?\}$
$\quad \land hall = hall$

$hall = hall$

is trivially true, and can be removed.

$\#\, hall \leqslant maxPlayers$

is simply repeating the 'before' state invariant which we already have included with *ClubState*.

Now

$badminton' = badminton \cup \{newMember?\}$

and

$hall \subseteq badminton$ (from *ClubState*)

implies

$hall \subseteq badminton'$

Therefore the latter is unnecessary.

$\exists\, badminton' : \mathbb{P}\ STUDENT \bullet badminton' = badminton \cup \{newMember?\}$

is also trivially true, so the schema simplifies to

pre *SuccessAddMember*

$ClubState$
$newMember? : STUDENT$

$newMember? \notin badminton$

We have arrived at the original precondition specified in the operation schema. This seems like a lot of work for no gain, but the preconditions of an operation are not always obvious from first inspection of the operation schema. The above precondition was explicitly stated in the operation schema, but it is possible to specify preconditions implicitly, in which case the above process would make them explicit. For example, if we are not interested in producing exception messages for the *AddMember* operation, we could leave out the predicate

$newMember? \notin badminton$

altogether.

AddMember

$\Delta\ ClubState$
$newMember? : STUDENT$

$badminton' = badminton \cup \{newMember?\}$
$hall' = hall$

The operation has no effect when *newMember*? $\notin$ *badminton*, and therefore does not violate the state invariant. However,

newMember? $\notin$ *badminton*

is still implicitly the precondition for successfully adding a new member. Relying on implicit preconditions, which are implied by the operation's postconditions, or which arise as a consequence of the state invariant, is not necessarily wrong. You should always try to write specifications using whatever style is appropriate to make the specification clear, precise and understandable. However, it is still important to establish the conditions in which each operation is applicable, and that it has been correctly specified, and this often means explicitly calculating preconditions.

Exercise 4.6

For the *EnterHall* schema

```
┌── EnterHall ──────────
│ Δ ClubState
│ enterer? : STUDENT
├───────────────────────
│ enterer? ∈ badminton
│ enterer? ∉ hall
│ # hall < maxPlayers
│ hall′ = hall ∪ {enterer?}
│ badminton′ = badminton
└───────────────────────
```

Part 1 Part 2

which of the explicitly stated preconditions could be made implicit? Write down the precondition schema for *EnterHall*.

Calculating preconditions can also reveal whether an operation is under-specified. For example, there may be combinations of 'before' state and inputs for which the operation does not specify anything. Take the *TotalAddMember* schema, for example.

$$\begin{aligned} \mathit{TotalAddMember} &\mathrel{\widehat=} (\mathit{AddMember} \land \mathit{SuccessMessage}) \lor \mathit{IsMember} \\ &\mathrel{\widehat=} \mathit{SuccessAddMember} \lor \mathit{IsMember} \end{aligned}$$

We can take advantage of the fact that pre distributes through disjunction, that is

$$\text{pre } \mathit{TotalAddMember} = \text{pre } \mathit{SuccessAddMember} \lor \text{pre } \mathit{IsMember}$$

We have already calculated the pre *SuccessAddMember* schema above.

Expanding the *IsMember* schema, making all 'after' state declarations and predicates explicit, we get the following:

```
┌── IsMember ───────────
│ ClubState
│ badminton′ : ℙ STUDENT
│ hall′ : ℙ STUDENT
│ newMember? : STUDENT
│ outcome! : MESSAGE
├───────────────────────
│ hall′ ⊆ badminton′
│ # hall′ ≤ maxPlayers
│
│ newMember? ∈ badminton
│ outcome! = isMember
│ badminton′ = badminton
│ hall′ = hall
└───────────────────────
```

Again, *ClubState* has been included instead of explicitly writing all the 'before' state declarations and predicates.

The precondition schema pre *IsMember* is the schema

$\mathit{IsMember} \setminus (\mathit{badminton}', \mathit{hall}', \mathit{outcome}!)$

defined as follows:

```
┌── pre IsMember ───────────────────────────────
│ ClubState
│ newMember? : STUDENT
├───────────────────────────────────────────────
│ ∃ badminton′, hall′ : ℙ STUDENT; outcome! : MESSAGE •
│   hall′ ⊆ badminton′
│   ∧ # hall′ ≤ maxPlayers
│   ∧ newMember? ∈ badminton
│   ∧ outcome! = isMember
│   ∧ badminton′ = badminton
│   ∧ hall′ = hall
└───────────────────────────────────────────────
```

By a similar process to that used for pre *SuccessAddMember*, this schema simplifies to

```
┌── pre IsMember ──────────
│ ClubState
│ newMember? : STUDENT
├──────────────────────────
│ newMember? ∈ badminton
└──────────────────────────
```

which means that pre *TotalAddMember* is the schema

```
┌── pre TotalAddMember ─────
│ ClubState
│ newMember? : STUDENT
├───────────────────────────
│ newMember? ∉ badminton
│   ∨ newMember? ∈ badminton
└───────────────────────────
```

Clearly, the predicate simplifies to *true*, indicating that the operation is applicable for any combination of 'before' state and inputs. This confirms that the operation is indeed total.

4.14 In conclusion

In this chapter, we have met many of the fundamental techniques and notations used in writing Z specifications. In the rest of the book, we will introduce further mathematical structures and associated operators required to construct more sophisticated specifications, but the basic principles introduced here of modelling state and operations, and of structuring by means of the schema calculus, apply to most Z specifications.

Exercises 4.7

A simple computer game is to be based on the following description;

> The system consists of a pond which may contain any number of fish up to and including a given maximum. Conceptually, the user is fishing in the pond with a rod and line. The user has a net suspended in the pond into which s/he must place any fish which s/he catches.

For a Z specification of this system, we require the following basic type:

$[FISH]$ the set of all fish

and the following global description:

$maxFish : \mathbb{N}$ the maximum number of fish which the pond can contain.

1. Write a schema to describe the state of this system. Hint: At this level of abstraction, we are interested only in the pond, the net and the relationship between them.
2. Write a schema for an operation whereby the user catches a fish and places it in the net.
3. Write a schema for an operation whereby the user returns one or more fish to freedom in the pond.
4. Write a schema for an operation whereby a number of new fish are added to the pond.
5. Write a schema for an operation which outputs the number of fish which are currently free in the pond.