CHAPTER 5

A first specification: The student badminton club

Aims

To describe the process of specification development, to outline a format for presenting specifications, and to present a complete simple specification.

Learning objectives

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

- develop simple Z specifications in a methodical way;
- produce a well-organised specification document.

5.1 Introduction

In the previous chapter, we introduced many of the concepts necessary for writing Z specifications, illustrating them by reference to a simple specification. However, we did not fully describe a *method* to be used when developing a Z specification. Such a method must embrace both mechanisms for structuring a specification, as described in the previous chapter, and a process to be followed in the development of the specification.

5.2 The process of specification development

A methodical approach to the development of Z specifications has evolved from the work of the Programming Research Group at Oxford University, and others. This was referred to as 'The Established Strategy' by Barden *et al.* (1994), and a summary of some of the steps they suggest is given below. The structure of the specification document should also be based on this sequence. The document should contain both the formal Z text and accompanying informal text which provides explanations of the formal text and describes any aspects of the specification not amenable to formal description.

Firstly, *requirements analysis* is carried out, in which the sets and constants necessary to describe the important parts of the problem are identified.

Then the *basic types* of the specification are identified and recorded, together with any *global variables* required.

Next, the *state schema* is developed. For a complex state, several schemas may be used for its constituent parts, and combined using the schema calculus.

The *initial state* is then described, and a *proof* that this initial state exists.

This is followed by schemas describing the *operations*, without taking into consideration the error cases for each operation.

Next, the *preconditions* of the operations are calculated, and each operation schema is checked and, if necessary, modified to ensure that it explicitly contains its precondition predicate. This facilitates the construction of *error handling schemas*, usually one for each possible exception to the operation's preconditions.

For each operation, the successful case schema and the error schema(s) are now combined using schema disjunction to produce *total specifications* for each operation. 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 leave the non-determinism in the formal text and document it in the accompanying informal text, leaving it up to the implementor of the specification to decide how to handle multiple error conditions.

The above is a 'standalone' strategy for developing Z specifications. Various schemes have also been developed for integrating Z into the process of non-formal structured analysis methods such as SSADM and Yourdon, and several object-oriented versions of Z (Stepney *et al.* 1992) have been produced, but these topics are beyond the scope of this book.

5.3 The student badminton club specification

This specification is for a system to manage a student badminton club. The specification could be implemented as a computer system or paper records. The system will keep track of the whereabouts of the club members and add or remove members from the club.

Basic type and global variable

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.

The basic type required is as follows:

$[\mathit{STUDENT}]$ the set of all students

The limit on the number of people allowed in the hall is *maxPlayers*.

$$
\begin{array}{|l}
\mathit{maxPlayers} : \mathbb{N} \\
\hline
\mathit{maxPlayers} = 20
\end{array}
$$

The state schema

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

We are interested in the whereabouts of the members of the badminton club. We 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 invariant properties are:

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

   $\mathit{hall} \subseteq \mathit{badminton}$

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

   $\#\ \mathit{hall} \leqslant \mathit{maxPlayers}$

The initial state

For the initial state, there are no members in the club; that is, the sets *badminton* and *hall* are empty.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitClubState}} \\
\hline
\mathit{ClubState}' \\
\hline
\mathit{badminton}' = \{\} \\
\mathit{hall}' = \{\} \\
\hline
\end{array}
$$

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

$\{\} \subseteq \{\}$ hall ⊆ badminton

and

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

The operations

We now define the successful cases of operations to add or remove a member to/from the club, and to add or remove a member to/from the hall.

Adding a new member

To join the club, a potential member *newMember?* must register with the club secretary, after which the member may go to the hall to play. *newMember?* must not already be a member of the club, and joins the club outside the hall.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddMember}} \\
\hline
\Delta\ \mathit{ClubState} \\
\mathit{newMember?} : \mathit{STUDENT} \\
\hline
\mathit{newMember?} \notin \mathit{badminton} \\
\mathit{badminton}' = \mathit{badminton} \cup \{\mathit{newMember?}\} \\
\mathit{hall}' = \mathit{hall} \\
\hline
\end{array}
$$

Removing a member

This operation removes a member from the club. The operation is non-deterministic in that the member may or may not be inside the hall prior to the operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemoveMember}} \\
\hline
\Delta\ \mathit{ClubState} \\
\mathit{member?} : \mathit{STUDENT} \\
\hline
\mathit{member?} \in \mathit{badminton} \\
\mathit{badminton}' = \mathit{badminton} \setminus \{\mathit{member?}\} \\
\mathit{hall}' = \mathit{hall} \setminus \{\mathit{member?}\} \\
\hline
\end{array}
$$

Entering the hall

We now specify an operation for a student, *enterer?*, to enter the hall. The student must be a member of the club and must not already be in the hall, and the number of members already in the hall must be less than *maxPlayers*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\_\_\ EnterHall\ \_\_\_\_}\\
\Delta\ ClubState \\
enterer? : STUDENT \\
\hline
enterer? \in badminton \\
enterer? \notin hall \\
\#\ hall < maxPlayers \\
hall' = hall \cup \{enterer?\} \\
badminton' = badminton \\
\hline
\end{array}
$$

Leaving the hall

We now specify an operation which removes a member from the hall. The member must be in the hall prior to the operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\_\_\ LeaveHall\ \_\_\_\_}\\
\Delta\ ClubState \\
leaver? : STUDENT \\
\hline
leaver? \in hall \\
hall' = hall \setminus \{leaver?\} \\
badminton' = badminton \\
\hline
\end{array}
$$

Error handling schemas

enumerated

The following free type represents the set of output messages required to construct total versions of the above operations, and for reports from the query operations which are defined below.

$$MESSAGE ::= success \mid isMember \mid notMember \mid hallFull \mid inHall \mid notInHall$$

The *success* message is used to indicate that an operation has been successfully completed, using the following schema:

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

The precondition for the *AddMember* operation is

$$newMember? \notin badminton$$

The exception to this operation occurs if *newMember?* is already a member of the club. In this case, the state does not change and the message *isMember* is produced. This is specified by the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\_\_\ IsMember\ \_\_\_\_}\\
\Xi\ ClubState \\
newMember? : STUDENT \\
outcome! : MESSAGE \\
\hline
newMember? \in badminton \\
outcome! = isMember \\
\hline
\end{array}
$$

together with addMember

The precondition for the *RemoveMember* operation is

$$member? \in badminton$$

The exception to this operation occurs if *member?* is not a member of the club. In this case, the state does not change and the message *notMember* is produced. This is specified by the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\_\_\ NotMember\ \_\_\_\_}\\
\Xi\ ClubState \\
member? : STUDENT \\
outcome! : MESSAGE \\
\hline
member? \notin badminton \\
outcome! = notMember \\
\hline
\end{array}
$$

Exception for Remove Member

There are three preconditions for the *EnterHall* operation: (p. 58)

$enterer? \in badminton$
$enterer? \notin hall$
$\#\ hall < maxPlayers$

There are therefore three corresponding exceptions for this operation, namely

The person is not a member of the club
The person is already in the hall
The hall is already full to its designated capacity

For the first of these, we can use the *NotMember* schema defined above, with the appropriate renaming of the input:

NotMember2 $\widehat{=}$

$NotMember\ [enterer?\ /\ member?]$ new name for the op! (exception)

The two other exceptions are specified by the following schemas:

AlreadyInHall

$\Xi\ ClubState$
$enterer? : STUDENT$
$outcome! : MESSAGE$

$enterer? \in hall$
$outcome! = inHall$

HallFull

$\Xi\ ClubState$
$outcome! : MESSAGE$

$\#\ hall = maxPlayers$
$outcome! = hallFull$

The exception to the *LeaveHall* schema occurs when the person is not in the hall. This is handled by the following schema. Note that we have chosen not to report the error condition where the person is not only not in the hall, but also not in the club. The corresponding precondition

$$leaver? \in badminton$$

is implicit in the operation schema. LeaveHall

NotInHall

$\Xi\ ClubState$
$leaver? : STUDENT$
$outcome! : MESSAGE$

$leaver? \notin hall$
$outcome! = notInHall$

exception to LeaveHall

The total versions of the operation schemas are now defined using the above exception handling schemas.

$TotalAddMember \mathrel{\widehat{=}} (AddMember \land SuccessMessage)$
$\quad \lor IsMember$

$TotalRemoveMember \mathrel{\widehat{=}} (RemoveMember \land SuccessMessage)$
$\quad \lor NotMember$

$TotalEnterHall \mathrel{\widehat{=}} (EnterHall \land SuccessMessage)$
$\quad \lor NotMember\ [enterer?\ /\ member?]$
$\quad \lor AlreadyInHall$
$\quad \lor HallFull$

$TotalLeaveHall \mathrel{\widehat{=}} (LeaveHall \land SuccessMessage)$
$\quad \lor NotInHall$

Query operations

(queries do not change state)

The following schema specifies an operation to output the set of all club members not in the hall:

OutsideHall

$\Xi\ ClubState$
$outside! : \mathbb{P}\ STUDENT$

$outside! = badminton \setminus hall$

There is no precondition; this operation may be applied to any state, and therefore no exception handling is necessary.

The following schema specifies an operation which inputs a student and outputs a message stating whether s/he is:

1. in the hall;
2. a member but not in the hall;
3. not a member.

Location

$\Xi\ ClubState$
$s? : STUDENT$
$report! : MESSAGE$

$s? \in hall \Rightarrow report! = inHall$
$s? \in badminton \land s? \notin hall \Rightarrow report! = notInHall$
$s? \notin badminton \Rightarrow report! = notMember$

note ⇒ !

Again, there is no precondition, and therefore no exception handling is necessary.