modelling at a *low* level of abstraction, akin to modelling an algorithm in a programming language. (See the answer to question 8 of Exercises 8.1 for a suggested alternative, more abstract, representation for the concept of a video copy for this specification.) This type of modelling may lead to a specification which is easier to implement in a programming language, but may also lead to a lack of clarity in the specification, and possibly less flexibility if it later becomes necessary to modify the specification to accommodate new requirements.

In contrast, the project allocation specification has a higher level of abstraction, with some use of non-determinism as already discussed. It may not be quite so obvious how to produce an implementation from such a specification, as it very much specifies *what* the system must do and not *how* it is to do it. However, the greater level of abstraction means that we can study the problem in its own right without having to be concerned about the paraphernalia of how we are eventually going to produce a program. So is it better to write specifications with a low level of abstraction, or a high level of abstraction? As always in the field of science, the answer is 'it depends'! The important thing is that the specification correctly represents the requirements, and is clear and understandable to all those who must read, discuss and use it. There are techniques in Z for writing specifications at a high level of abstraction, and then refining them to a lower level of abstraction, while proving that the properties of the original specification are preserved, thus getting the best of both worlds. However, these techniques are beyond the scope of this book.

CHAPTER 11

Two outline Z specifications

Aims

To apply the preceding material to some more demanding specifications, and to illustrate the use of generic constants.

Learning objectives

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

- apply the Z notation to modelling the state of more complex systems using combinations of mathematical structures;
- specify more challenging operations;
- appreciate the need to validate your specifications to ensure they are a true description of the required system;
- recognise situations in which it is desirable to define a new operator, or to name subexpressions using **let**, to enhance the readability of a specification.

11.1 Introduction

In this chapter, we will outline the construction of two Z specifications. The specifications are incomplete and not very large, but they are a little more challenging than the examples we have encountered so far, and should illustrate the value of Z in expressing complex ideas clearly and precisely.

11.2 A timetabling system

A university requires a timetabling system to keep track of where and when its degree modules are scheduled, and which students are registered for each module. Sometimes a module can be scheduled in several rooms at the same time, perhaps for tutorial sessions in smaller groups, in which case we wish to know which room any given student should be in. We could visualise the final system as an on-line facility by which students can find out where they should be at any given time, register for modules, find out where their friends will be, etc. For simplicity, we will say that a given student may attend any number of modules provided none of their chosen modules clash on the timetable, and we will assume that there is no limit on the capacity of rooms or the number of students permitted to register for a module. For the time being, we will also ignore the role of lecturers and tutorial staff in the specification!

For our preliminary analysis, we identify the basic types (given sets) necessary for the specification. We will assume that for the purpose of timetabling, the week is divided into fixed-size time slots in which modules may be scheduled. This suggests the following basic types:

$$[STUDENT, MODULE, TIMESLOT, ROOM]$$

which are respectively the set of all students, the set of all modules, the set of all time slots and the set of all teaching rooms.

The time slots could be units of one hour, two hours, half a day, or whatever. Making *TIMESLOT* a basic type means that we do not need to concern ourselves with the precise nature of a time slot at our chosen level of abstraction.

We must now describe the abstract state of the system using appropriate mathematical structures. Clearly, there is more than one way of doing this, but the one we will choose is based on the idea that each individual student and each individual module will be associated with their/its own individual schedule. This models the real-world situation, where each student will carry around their personal schedule, and a module lecturer might issue a schedule for his/her module. We can model a schedule as a relation between time slots and rooms. However, for any given time slot, a given student can only be in a maximum of one room (one cannot be in two places at the same time!), so therefore a student's schedule will be a *partial function* from time slots to rooms. Therefore, the type of a student's schedule will be

for a specific student

$$TIMESLOT \nrightarrow ROOM$$

partial function

and the type of a module's schedule will be

module can occupy more than one room for a time slot

$$TIMESLOT \leftrightarrow ROOM$$

binary relation

Now, we need a way to represent the schedules for *all* students and *all* modules on the course. We can do this using two partial functions, one of which maps individual students to their schedules, and the other of which maps individual modules to their schedules. These functions are called *studentTT* and *moduleTT* respectively.

$$studentTT : STUDENT \nrightarrow (TIMESLOT \nrightarrow ROOM)$$
$$moduleTT : MODULE \nrightarrow (TIMESLOT \leftrightarrow ROOM)$$

studentTT s is the schedule for student *s*, and *moduleTT m* is the schedule for module *m*.

$$\mathrm{dom}\ studentTT$$

is the set of all students on the course and

in the English sense! (=program)

$$\mathrm{dom}\ moduleTT$$

= course!

is the set of all modules which may be offered on the course. However, not all students are registered for modules and not all modules are offered at any given time. For a student *s* not registered for any modules,

$$studentTT\ s = \varnothing$$

and for a module *m* not currently offered,

$$moduleTT\ m = \varnothing$$

that is, *s* and *m* are mapped to empty schedules.

These structures constitute the state declarations for the specification. We must now identify the invariant properties required to define the state predicate:

1. It must not be possible to have two modules scheduled in the same room at the same time; in other words, the relations (schedules) in the range of *moduleTT* must be disjoint.

   $$\forall r, s : \mathrm{ran}\ moduleTT \bullet \mathrm{disjoint}\ \langle r, s \rangle$$

   The predicate implies that no two module schedules have any maplets in common; that is, no two modules are scheduled in the same room at the same time.
2. A student can only be scheduled for a time and place where a module is scheduled. In other words, any maplet found in any of the functions in the range of *studentTT* must also be found in precisely one of the relations in the range of *moduleTT*.

   To express this requirement in Z, we need a convenient way of referring to the set of *all* the maplets in the ranges of *studentTT* and of *moduleTT* functions!

respectively. We will define a generic global constant function *allPairs*, which when applied to objects such as *studentTT* or *moduleTT* returns the distributed union of all the relations in the object's range. (See Section 9.4 for a description of generic constants.)

TT types

$$
\begin{array}{|l}
\hline\hline
\quad [X, Y, Z] \\
\mathit{allPairs} : (X ⇸ (Y \leftrightarrow Z)) \rightarrow (Y \leftrightarrow Z) \\
\hline
\forall f : (X ⇸ (Y \leftrightarrow Z)) \bullet \\
\mathit{allPairs}\ f = \bigcup \{ x : X \mid x \in \mathrm{dom}\ f \bullet f x \} \\
\hline
\end{array}
$$

function!

For example, say *studentTT* has the following value:

$$
\begin{array}{l}
\mathit{studentTT} = \{ \mathit{sally} \mapsto \{ t_1 \mapsto r_4, t_3 \mapsto r_4, t_5 \mapsto r_1 \}, \\
\qquad\qquad\quad \mathit{helen} \mapsto \{ t_2 \mapsto r_1, t_3 \mapsto r_4 \}, \\
\qquad\qquad\quad \mathit{john} \mapsto \{ t_1 \mapsto r_1, t_5 \mapsto r_2 \} \\
\qquad\qquad\quad \}
\end{array}
$$

then

$$\mathit{allPairs}\ \mathit{studentTT} = \{ t_1 \mapsto r_4, t_3 \mapsto r_4, t_5 \mapsto r_1, t_2 \mapsto r_1, t_1 \mapsto r_1, t_5 \mapsto r_2 \}$$

Condition 2 can now be expressed as

$$\mathit{allPairs}\ \mathit{studentTT} \subseteq \mathit{allPairs}\ \mathit{moduleTT}$$

Note that this predicate simply states that every maplet in every student's schedule also occurs in the schedule of *at least* one module. The additional constraint that each student schedule maplet is in the schedule of *precisely one* module is implied by the predicate for condition 1.

No two modules are scheduled in the same room at the same time.

Here is our final condition.

3. Any given student is either scheduled to attend a given module at every available time slot, or not at all. Although this is a valid requirement for our specification, it may not be reflected in the behaviour of real students!

$$
\begin{array}{l}
\forall s : \mathrm{dom}\ \mathit{studentTT};\ m : \mathrm{dom}\ \mathit{moduleTT} \\
\bullet\ (\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) \neq \varnothing \Rightarrow \\
\quad \mathrm{dom}(\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) = \mathrm{dom}(\mathit{moduleTT}\ m)
\end{array}
$$

The expression

$$(\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) \neq \varnothing$$

is true iff student *s* and module *m* are scheduled to be in the same place at the same time on at least one occasion. The expression

$$\mathrm{dom}(\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) = \mathrm{dom}(\mathit{moduleTT}\ m)$$

states that student *s* is scheduled to attend module *m* at all of the times when the module is available.

The state schema for our system is therefore

$$
\begin{array}{|l}
\hline
\quad \mathit{Timetable} \\
\mathit{studentTT} : \mathit{STUDENT} ⇸ (\mathit{TIMESLOT} ⇸ \mathit{ROOM}) \\
\mathit{moduleTT} : \mathit{MODULE} ⇸ (\mathit{TIMESLOT} \leftrightarrow \mathit{ROOM}) \\
\hline
\forall r, s : \mathrm{ran}\ \mathit{moduleT}\overset{T}{\bullet}\ \mathit{disjoint}\langle r, s \rangle \\
\\
\mathit{allPairs}\ \mathit{studentTT} \subseteq \mathit{allPairs}\ \mathit{moduleTT} \\
\\
\forall s : \mathrm{dom}\ \mathit{studentTT};\ m : \mathrm{dom}\ \mathit{moduleTT} \\
\bullet\ (\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) \neq \varnothing \Rightarrow \\
\quad \mathrm{dom}(\mathit{studentTT}\ s \cap \mathit{moduleTT}\ m) = \mathrm{dom}(\mathit{moduleTT}\ m) \\
\hline
\end{array}
$$

For the initial state, we will have no students and no modules in the system.

$$
\begin{array}{|l}
\hline
\quad \mathit{InitTimetable} \\
\mathit{TimeTable}' \\
\hline
\mathit{studentTT}' = \{\ \} \\
\mathit{moduleTT}' = \{\ \} \\
\hline
\end{array}
$$

Inspection of the system state shows that this initial state exists. (Strictly, we are obliged to verify this formally.)

We should examine our specification to see whether it has any unforeseen undesirable properties. This check could take the form of a walkthrough with colleagues, in which the group tries to find inconsistencies in the specification as a representation of the user's requirements. For example, is it possible for a student to be scheduled for two different modules which clash on the timetable; that is, which take place at the same time? The answer is no, because every student's schedule is a function.

We should also check for any particular desirable properties which we would expect the specification to have. Other questions can give us more insight into our specification. For example, if two students are never in the same place at the same time, must it be that they have no modules in common? A moment's consideration reveals that this is not necessarily true. It may be that a module has concurrent sessions in more than one room at every time slot in which it is scheduled, in which case two students could do the module without ever being in the same room.

Such checks can be made at any stage in the development of the specification, but it is a good idea to find any errors in the state specification before embarking on operation specifications.

The use of implication $\Rightarrow$ is a common source of errors, and it is a good idea to check that the correct meaning has been captured. In condition 3, for example, what have we said about the following situation?

$$(studentTT\, s \cap moduleTT\, m) = \varnothing$$

In fact, dom $\varnothing = \varnothing$ (Spivey 1992), and so the consequent (right-hand side of $\Rightarrow$) will be true iff $moduleTT\, m = \varnothing$, that is module m is not timetabled, and false otherwise; either way the predicate will be true. so why do we care ?

Validation exercises are not guaranteed to reveal all inconsistencies in the specification, but the fact that the specification is expressed in a formal language makes it easier to identify such inconsistencies. Verification by formal proof gives more confidence, but is much more expensive in time and effort.

Exercises 11.1

1. What assumption is implicit in condition 3 of the state invariant?
2. What issues would have to be considered in order to introduce the concept of *lecturers* for the modules? For example, what assumptions must you make about the case where a module is scheduled in more than one room at the same time?

Adding a student to the course

"program"

The student $s?$ must not already be on the course.

$$s? \notin \mathrm{dom}\ studentTT$$

The student joins the course with an initially empty schedule.

$$studentTT' = studentTT \cup \{s? \mapsto \varnothing\}$$

The operation schema is as follows:

AddStudent

$$\begin{array}{|l}
\Delta\ Timetable \\
s? : STUDENT \\
\hline
s? \notin \mathrm{dom}\ studentTT \\
studentTT' = studentTT \cup \{s? \mapsto \varnothing\} \\
moduleTT' = moduleTT
\end{array}$$

Exercises 11.2

1. Write the operation schema *AddModule*, to add a module with an empty schedule to the system.
2. Write the operation schemas *RemoveStudent* and *RemoveModule*.

Scheduling a module

The module $m?$ must be a valid module for the course

$$m? \in \mathrm{dom}\ moduleTT$$

and it must not already be scheduled:

$$moduleTT\, m? = \varnothing$$

The postcondition is non-deterministic in that more than one valid potential schedule for the module may be possible, and in these circumstances, the postcondition does not state which valid schedule is to be selected. This arbitrary decision is left to the implementors of the system. Thus the schema defines more than one possible final state.

$$\begin{array}{l}
\exists\ schedule : TIMESLOT \leftrightarrow ROOM \bullet \\
\quad (allPairs\ moduleTT \cap schedule = \varnothing \\
\quad \wedge\ moduleTT' = moduleTT \oplus \{m? \mapsto schedule\})
\end{array}$$

do not clash with another module
in terms of room-timeslot

The components of this predicate are explained as follows. Any valid schedule for this module must not clash with that of any other module.

$$allPairs\ moduleTT \cap schedule = \varnothing$$

The module is scheduled by overriding the *moduleTT* function.

$$moduleTT' = moduleTT \oplus \{m? \mapsto schedule\}$$

The operation schema is as follows:

$$
\begin{array}{|l}
\hline
\textit{ScheduleModule} \\
\Delta\, \textit{Timetable} \\
m?: \textit{MODULE} \\
\hline
m? \in \mathrm{dom}\ \textit{moduleTT} \\
\textit{moduleTT}\ m? = \varnothing \\
\exists\ \textit{schedule}: \textit{TIMESLOT} \leftrightarrow \textit{ROOM} \bullet \\
\quad (\textit{allPairs moduleTT} \cap \textit{schedule} = \varnothing \\
\quad\ \wedge \textit{moduleTT}' = \textit{moduleTT} \oplus \{m? \mapsto \textit{schedule}\}) \\
\\
\textit{studentTT}' = \textit{studentTT} \\
\hline
\end{array}
$$

Descheduling a module

The module to be descheduled $m?$ must be a valid module for the course.

$$m? \in \mathrm{dom}\ \textit{moduleTT}$$

and the schedule for $m?$ must not already be empty.

$$\textit{moduleTT}\ m? \neq \varnothing$$

To deschedule the module, we override *moduleTT* to map the module to the empty schedule.

$$\textit{moduleTT}' = \textit{moduleTT} \oplus \{m? \mapsto \varnothing\}$$

We must also remove the module from the schedules of all students who are registered for it.

$$\textit{studentTT}' = \bigcup\{s: \mathrm{dom}\ \textit{studentTT} \bullet \{s \mapsto (\textit{studentTT}\ s \setminus \textit{moduleTT}\ m?)\}\}$$

The operation schema is as follows:

$$
\begin{array}{|l}
\hline
\textit{DescheduleModule} \\
\Delta\, \textit{Timetable} \\
m?: \textit{MODULE} \\
\hline
m? \in \mathrm{dom}\ \textit{moduleTT} \\
\textit{moduleTT}\ m? \neq \varnothing \\
\textit{moduleTT}' = \textit{moduleTT} \oplus \{m? \mapsto \varnothing\} \\
\textit{studentTT}' = \bigcup\{s: \mathrm{dom}\ \textit{studentTT} \bullet \{s \mapsto (\textit{studentTT}\ s \setminus \textit{moduleTT}\ m?)\}\} \\
\hline
\end{array}
$$

Note that the precondition $\textit{moduleTT}\ m? \neq \varnothing$ is not necessary for the operation as specified, but it may be important to identify it at implementation, to output an appropriate message if the precondition is not satisfied. This exception would be handled by a separate schema, and the total operation would be specified using schema calculus.

Registering a student for a module

The student $s?$ must be on the course, and the module $m?$ must be a valid module for the course.

$$
\begin{array}{l}
s? \in \mathrm{dom}\ \textit{studentTT} \\
m? \in \mathrm{dom}\ \textit{moduleTT}
\end{array}
$$

The module must be scheduled.

$$\textit{moduleTT}\ m? \neq \varnothing$$

The student must be free at all the times when the module is scheduled.

$$\mathrm{dom}(\textit{studentTT}\ s?) \cap \mathrm{dom}(\textit{moduleTT}\ m?) = \varnothing$$

time-room combinations

We do not want to add all the module's slots to the student's schedule; where the module is scheduled in more than one room at the same time, the student must be allocated only one of these slots. The following postcondition predicate is non-deterministic in this respect, in that it does not state specifically which slot is to be allocated in these circumstances. (In a real system, issues such as room capacities and previous allocations would come into play.) The postcondition is as follows:

$$
\begin{array}{l}
\exists\ \textit{newPairs}: \textit{TIMESLOT} \rightarrowtail \textit{ROOM} \\
\bullet ((\mathrm{dom}\ \textit{newPairs} = \mathrm{dom}\ \textit{moduleTT}\ m?) \\
\quad \wedge (\textit{newPairs} \subseteq \textit{moduleTT}\ m?) \\
\quad \wedge (\textit{studentTT}' = \textit{studentTT} \oplus \{s? \mapsto \textit{studentTT}\ s? \cup \textit{newPairs}\}))
\end{array}
$$

match timeslots with those of the module

newPairs is the set of those pairs from the module's schedule which are to be added to the student's schedule. Note that because students' schedules are functions, *newPairs* must be a function. The constituent parts of this predicate are explained as follows.

The time slots for the pairs to be added to the student's schedule are precisely *all* the time slots when the module is available.

$$\mathrm{dom}\ \textit{newPairs} = \mathrm{dom}\ \textit{moduleTT}\ m?$$

The pairs to be added to the student's schedule are pairs from the module's schedule.

$$newPairs \subseteq moduleTT\ m?$$

The appropriate pairs are added to the student's schedule by overriding the *studentTT* function.

$$studentTT' = studentTT \oplus \{s? \mapsto studentTT\ s? \cup newPairs\}$$

The operation schema is as follows:

RegForModule

$$\begin{array}{l}
\Delta\ Timetable \\
s? : STUDENT \\
m? : MODULE \\
\hline
s? \in \mathrm{dom}\ studentTT \\
m? \in \mathrm{dom}\ moduleTT \\
moduleTT\ m? \neq \varnothing \\
\mathrm{dom}(studentTT\ s?) \cap \mathrm{dom}(moduleTT\ m?) = \varnothing \\
\\
\exists\ newPairs : TIMESLOT \nrightarrow ROOM \\
\bullet ((\mathrm{dom}\ newPairs = \mathrm{dom}\ moduleTT\ m?) \\
\quad \wedge (newPairs \subseteq moduleTT\ m?) \\
\quad \wedge (studentTT' = studentTT \oplus \{s? \mapsto studentTT\ s? \cup newPairs\})) \\
\\
moduleTT' = moduleTT
\end{array}$$

Exercises 11.3

1. Write a schema to 'deregister' a student from a module.
2. How could we modify the state schema to specify that each module can use a maximum of one room in any given time slot?
3. How could we modify the state schema to specify that each module is only allowed one time slot in the schedule?
4. Write a Z expression for the set of all students registered for a module *m*.
5. Write a Z expression for the set of all modules being taken by a student *s*.
6. Write a Z expression for the set of all students in a room *r* at a time *t*.
7. Write a Z expression for the set of all modules which student *p* and student *q* have in common.
8. Write a Z expression for the set of all times when student *p* and student *q* are in the same room.

9. Write a Z expression for the set of all modules which clash with a module *m* on the timetable, that is which take place at the same time as module *m*.
10. Write a Z expression for the set of all time/room maplets for which one or more modules are scheduled, but no students are scheduled.
11. What would it mean if *allPairs moduleTT* was one-to-one? (See Section 7.4 for a definition of the term one-to-one for functions.)
12. What would it mean if *allPairs studentTT* was one-to-one?
13. How could you extend the specification to include concepts such as room capacities and maximum numbers of students allowed in a module.
14. Why would it be difficult to extend the module registration example in Chapter 6 to include timetabling information?

11.3 A genealogical database

A database is required to keep track of genealogical relationships between people (family trees). It would be possible to represent the required relationships (parent, grandparent, aunt, cousin, etc.) separately, but this would limit the number of relationships, increase the complexity of the specification, and would make it necessary to carry out extensive integrity checks every time the database is updated. We will therefore represent only the minimum information necessary to be able to define operations to output any required relationships. The most fundamental genealogical relationship is that of parent to child, and this, together with the sex of all the individuals in the database, will be enough to enable us to specify all the operations we require. This suggests the following types:

$[PERSON]$	the set of all people
$GENDER ::= male \mid female$	

The parent/child relationship can be represented by the relation

$$parent : PERSON \leftrightarrow PERSON$$

(consider as "has Parent")

where

$$x \mapsto y \in parent$$

represents the information that y is x's parent.

The sex of all the people in the database can be represented by the function

$$sex : PERSON \nrightarrow GENDER$$

where

$$\mathrm{dom}\ sex$$

is the set of all people in the database. A person can be in the database even if they do not occur in *parent*. It may be that information about their parents or children is not available or is incomplete.

$$sex\ p$$

is the sex of person *p*.

We need an invariant predicate to state that the relation *parent* only holds information about people in the database.

$$\mathrm{dom}\ parent \cup \mathrm{ran}\ parent \subseteq \mathrm{dom}\ sex$$

We also need an invariant predicate to capture such requirements as:

names unique?

1. If *y* is *x*'s parent, then *x* cannot be *y*'s parent.
2. If *y* is *x*'s parent, then *y* cannot also be an ancestor of *x* at a depth in the family tree greater than that of parent, that is *y* cannot be *x*'s grandparent, great-grandparent, etc.
3. A person cannot be their own parent.

We can neatly capture all of these requirements in a predicate which states that a person cannot be their own ancestor. (You may have to think about this to convince yourself that it is true.) Now the transitive closure of *parent*

$$parent^+$$

will relate any person to their ancestors (parents, grandparents, great-grandparents, etc.). The required predicate is therefore simply

$$\forall p : PERSON \bullet p \mapsto p \notin parent^+$$

The final restriction is that anyone in the database can have a maximum of two parents, and if they have two, the parents must be of opposite sexes!

$$\forall p, q, r : PERSON \bullet \{p \mapsto q, p \mapsto r\} \subseteq parent \wedge q \neq r \Rightarrow sex\ q \neq sex\ r$$

Note that the condition restricts the number of parents to a maximum of two because the type *GENDER* only has two values.

The state schema is as follows:

GenDB

$$\begin{array}{l}
parent : PERSON \leftrightarrow PERSON \\
sex : PERSON \twoheadrightarrow GENDER \\
\hline
\mathrm{dom}\ parent \cup \mathrm{ran}\ parent \subseteq \mathrm{dom}\ sex \\
\forall p : PERSON \bullet p \mapsto p \notin parent^+ \\
\forall p, q, r : PERSON \bullet \{p \mapsto q, p \mapsto r\} \subseteq parent \wedge q \neq r \Rightarrow sex\ q \neq sex\ r
\end{array}$$

You may notice that there is no requirement for a person's parents to come from the same generation as each other; nor does the database contain any information about whether the people it contains are alive or dead. Thus a person in the database could have Julius Caesar as one parent and Joan of Arc as the other. However, for simplicity, we will quietly ignore this.

For the initial state, we will have an empty database.

InitGenDB

$$\begin{array}{l}
GenDB' \\
\hline
sex' = \varnothing \\
parent' = \varnothing
\end{array}$$

The predicate $parent' = \varnothing$ is not strictly necessary, as it is implied by the state invariant, but increases the clarity of the specification. The initial state satisfies the state invariant.

Operations to change the database

The users of the database will be able to modify the information it contains using operations to add a person to the database, remove a person from the database, add or remove a parent/offspring relationship to/from the database, change the name of a person in the database, and change the sex of a person in the database.

Adding a person to the database

The inputs are a person and their sex.

$$\begin{array}{l}
name? : PERSON \\
morf? : GENDER
\end{array}$$

The person *name*? must not already be in the database.

$$name? \notin \mathrm{dom}\ sex$$

We add them to the *sex* function.

$$sex' = sex \cup \{name? \mapsto morf?\}$$

The operation schema is as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddPerson}}\\ \hline
\Delta\ GenDB\\
name? : PERSON\\
morf? : GENDER\\ \hline
name? \notin \mathrm{dom}\ sex\\
sex' = sex \cup \{name? \mapsto morf?\}\\
parent' = parent\\ \hline
\end{array}
$$

Exercise 11.4

Write a schema for an operation to remove a person from the database.

Adding a parent/offspring relationship to the database

The inputs are a potential offspring and a potential parent.

$$off?, par? : PERSON$$

The potential offspring and parent must be in the database

$$\{off?, par?\} \subseteq \mathrm{dom}\ sex$$

and must not already be a maplet in *parent*, in either order.

$$
\begin{array}{l}
off? \mapsto par? \notin parent\\
par? \mapsto off? \notin parent
\end{array}
$$

There must not be more than one parent already in the database for the potential offspring

$$\#(\{off?\} \lhd parent) \leqslant 1$$

and if there is one such parent, their sex must not be the same as the new potential parent.

$$\forall x : PERSON \bullet off? \mapsto x \in parent \Rightarrow sex\ x \neq sex\ par?$$

For the postcondition, we simply add the new maplet to the relation *parent*.

$$parent' = parent \cup \{off? \mapsto par?\}$$

The operation schema is as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddRel}}\\ \hline
\Delta\ GenDB\\
off?, par? : PERSON\\ \hline
\{off?, par?\} \subseteq \mathrm{dom}\ sex\\
off? \mapsto par? \notin parent\\
par? \mapsto off? \notin parent\\
\#(\{off?\} \lhd parent) \leqslant 1\\
\forall x : PERSON \bullet off? \mapsto x \in parent \Rightarrow sex\ x \neq sex\ par?\\
parent' = parent \cup \{off? \mapsto par?\}\\
sex' = sex\\ \hline
\end{array}
$$

Exercise 11.5

Write a schema for an operation to remove a parent/offspring relationship from the database.

Changing the name of a person in the database

This is a rather artificial example, as we will now have to describe the type *PERSON* as the set of all people's names, which implies that all names are unique. In a real system, *PERSON* would have to be implemented as a set of unique identifiers of some sort, and for simplicity, when we refer to a 'name', we will take it to mean one of these identifiers. The inputs are the old name and the new name.

$$old?, new? : PERSON$$

The old name must be in the database, and the new one must not.

$$
\begin{array}{l}
old? \in \mathrm{dom}\ sex\\
new? \notin \mathrm{dom}\ sex
\end{array}
$$

The old name must be replaced by the new one in the *sex* function

$$sex' = (\{old?\} \ntriangleleft sex) \cup \{new? \mapsto \mathrm{sex}\ old?\}$$

and all relationships involving the old name must be modified to use the new name.

functional notation used for relations!

$$
\begin{array}{l}
parent' = (\{old?\} \mathbin{⩤} parent \mathbin{⩥} \{old?\}) \\
\qquad \cup \{x : PERSON \mid x \in parent\ (\!| \{old?\} |\!) \bullet new? \mapsto x\} \\
\qquad \cup \{x : PERSON \mid x \in parent^{-1}\ (\!| \{old?\} |\!) \bullet x \mapsto new?\}
\end{array}
$$

The operation schema is as follows:

ChangeName

$$
\begin{array}{|l}
\Delta\, GenDB \\
old?, new? : PERSON \\
\hline
old? \in \mathrm{dom}\ sex \\
new? \notin \mathrm{dom}\ sex \\
sex' = (\{old?\} \mathbin{⩤} sex) \cup \{new? \mapsto \mathrm{sex}\ old?\} \\
parent' = (\{old?\} \mathbin{⩤} parent \mathbin{⩥} \{old?\}) \\
\qquad \cup \{x : PERSON \mid x \in parent\ (\!| \{old?\} |\!) \bullet new? \mapsto x\} \\
\qquad \cup \{x : PERSON \mid x \in parent^{-1}\ (\!| \{old?\} |\!) \bullet x \mapsto new?\}
\end{array}
$$

Changing the sex of a person in the database

The person $p?$ must be in the database.

At first sight, this operation would appear to require a simple overriding of the *sex* function. However, the tricky part in specifying the operation is that the person may be recorded as a *parent* in the database, that is they may be a member of ran *parent*. This means that the sex recorded for other people may have to change in order to maintain the integrity of the database, that is so that the database will not contain children with two mothers or two fathers.

An implementation of the operation would probably ask the user whether s/he wished to proceed in these circumstances, as it is unlikely that s/he would wish to make such changes simply to maintain the integrity of the database. It is much more likely that the proposed change was not correct in the first place. However, we will specify a rather contrived operation which simply makes the necessary changes. Essentially, any people with whom our person has had children must change their sex, as must anyone who has had children with those people, and so on, to ensure that anyone with two parents in the database has one of each sex.

So marriage is not an issue!

Now the relation

$$parent^{-1}\,;parent$$

composition

relates together people who have had children with each other. The transitive closure of this relation

$$(parent^{-1}\,;parent)^{+}$$

is the relation such that

$$p \mapsto q \in (parent^{-1}\,;parent)^{+}$$

iff p has had children with q, or p has had children with someone who has had children with q, or p has had children with someone who has had children with someone who has had children with $q, \ldots$ well, you get the idea. The relational image in this relation of the set consisting solely of our person $p?$ gives the set of all people who must have their sex changed by this operation.

$$(parent^{-1}\,;parent)^{+}\ (\!| \{p?\} |\!)$$

We can now use this to specify a function mapping these people to the opposite sex from that given them by the *sex* function, and finally use this function to override the original *sex* function. The resulting postcondition is as follows:

"enumerated" genders!

$$
\begin{array}{l}
sex' = sex \oplus \\
\{q : PERSON; s : GENDER \mid (q \in (parent^{-1}\,;parent)^{+}\ (\!| \{p?\} |\!)) \\
\qquad \wedge (s \neq sex\ q) \bullet q \mapsto s\}
\end{array}
$$

Note that the relation $parent^{-1}\,;parent$ also relates every parent to themselves, which enables the postcondition to specify the sex change for $p?$ him/herself.

The operation schema is as follows:

ChangeSex

$$
\begin{array}{|l}
\Delta\, GenDB \\
p? : PERSON \\
\hline
p? \in \mathrm{dom}\ sex \\
sex' = sex \oplus \\
\{q : PERSON; s : GENDER \mid (q \in (parent^{-1}\,;parent)^{+}\ (\!| \{p?\} |\!)) \\
\qquad \wedge (s \neq sex\ q) \bullet q \mapsto s\} \\
\\
parent' = parent
\end{array}
$$

We also require query operations to interrogate the database for information about various relationships. The next set of exercises gives you the opportunity to specify some, after which we conclude with an operation to find the common ancestors of two given people, and an operation to find the set of all cousins of specified type and removedness for a given person.

Exercises 11.6

1. Specify an operation to return the set of all people who have any one of the following relationships to a given person $x?$. The name of the required relationship should be an input to the operation.

   (i) The parents of person $x?$
   (ii) The grandparents of person $x?$
   (iii) The grandchildren of person $x?$
   (iv) The descendants of person $x?$
   (v) The siblings of person $x?$
   (vi) The aunts of person $x?$

2. Give a Z expression for the set of all people in the database who have no relatives in the database.
3. Give a Z expression for the set of all people in the database who have no siblings in the database.
4. Specify an operation to output the number of generations through which a given individual p can trace his/her family history in the database.

The common ancestors of two given people

This operation must return the set *cas*! containing the common ancestors of two people, say $p?$ and $q?$. We will further stipulate that the set contains only the common ancestors of minimum degree, where the degree refers to the number of steps in the path up and/or down the family tree between the two people. For example, if $p?$ and $q?$ are siblings, the degree is 2. If $p?$ and $q?$ are first cousins, the degree is 4. If $p?$ is the grandparent of $q?$, the degree is 2.

The precondition is that all the people involved are in the database.

$$\{p?, q?\} \cup cas! \subseteq \mathrm{dom}\ sex$$

But cas! is the computed result!

The postcondition characterises elements of the set of common ancestors of minimum degree as people for whom there are multiple compositions of *parent* which map both $p?$ and $q?$ to them, and furthermore there are no other common ancestors with degree smaller than that of members of this set.

$$\begin{array}{l} cas! = \{ca : PERSON \mid \exists m, n : \mathbb{N} \bullet \\ \qquad ((p? \mapsto ca \in parent^n \wedge q? \mapsto ca \in parent^m) \\ \qquad \wedge \neg \exists r : PERSON; x, y : \mathbb{N} \bullet \\ \qquad\qquad ((x + y < m + n) \wedge p? \mapsto r \in parent^x \wedge q? \mapsto r \in parent^y))\} \end{array}$$

Somerel^0 : identity relation p. 74

The operation schema is as follows:

CommonAncestors

$$\begin{array}{l} \Xi\ GenDB \\ p?, q? : PERSON \\ cas! : \mathbb{P}\ PERSON \\ \hline \{p?, q?\} \cup cas! \subseteq \mathrm{dom}\ sex \\ \\ cas! = \{ca : PERSON \mid \exists m, n : \mathbb{N} \bullet \\ \qquad ((p? \mapsto ca \in parent^n \wedge q? \mapsto ca \in parent^m) \\ \qquad \wedge \neg \exists r : PERSON; x, y : \mathbb{N} \bullet \\ \qquad\qquad ((x + y < m + n) \wedge p? \mapsto r \in parent^x \wedge q? \mapsto r \in parent^y))\} \end{array}$$

Note that the specification allows for the special case where one of the two people is a direct descendant of the other, in which case the common ancestor is the person higher in the family tree. m and n are natural numbers, and it is therefore possible for either or both to be zero, yielding the identity relation on *PERSON*. The only debate might be as to whether a person can be their own ancestor!

The cousins of a given person

This operation returns the set *cousins*! containing all cousins of a given type and removedness for a given person, say $p?$. Cousins are people who have a common ancestor who is more distant than a parent, and who are not siblings. The type of cousinship is determined by the shortest path from the common ancestor to either one of the cousins. For example, for first cousins the common ancestor would be a grandparent of one cousin, for second cousins a great-grandparent, and so on. 'Removed' refers to the difference in the number of steps in the path from each cousin to the common ancestor. 'Once removed' means a difference of one step, 'twice removed' means a difference of two steps, etc. For example, my first cousins' children are my first cousins once removed, and their children are my first cousins twice removed.

no need to say this!

This operation has an input $nth? : \mathbb{N}_1$, to represent the type of cousinship, and an input $rem? : \mathbb{N}$, to represent how far removed the relationship is. $nth?$ has type $\mathbb{N}_1$ because the minimum type of cousinship is first cousins, represented by the number 1.

The precondition simply states that everyone involved is in the database:

$$\{p?\} \cup cousins! \subseteq \mathrm{dom}\ sex$$

For the postcondition we define a relation which maps a cousin to his/her cousins at the same or a lower level in the family tree. Remember that $nth?$ is defined with respect to the cousin with the shortest path to the common ancestor, and *rem*? represents the difference in length between the two paths. The required relation is

$$(parent^{nth?+1}\,;(parent^{-1})^{nth?+1+rem?})\setminus(\underbrace{parent\,;parent^{-1}}_{\text{sibling relation, including a person with himself as a sibling}})$$

The set subtraction removes from the relation all pairs of people who have the same parents. Therefore if *rem*? is zero, the relation does not relate anyone to themselves or their siblings.

The image of our person in this relation will yield all the relevant cousins at the same or a lower level in the family tree as our person. However, if the value of *rem*? is non-zero, our person may have some cousins with the same relationship at a higher level in the tree. We therefore also require the image of our person in the *inverse* of the above relation. To avoid repeating the expression for this relation, we use a Z construct called a *let predicate* which provides a method for naming subexpressions in complex predicates.

let $n == e \bullet p$

stands for the predicate *p*, but wherever the name *n* occurs in *p*, it represents the value *e*. We use a let predicate to give the name *cosrel* to the above relation for use as a subexpression in specifying the set of all *nth*? cousins *rem*? removed, both above and below person *p*? in the family tree.

The operation schema is as follows:

```
──── Cousins ──────────────────────────────────────
│ Ξ GenDB
│ p? : PERSON
│ nth? : ℕ₁
│ rem? : ℕ
│ cousins! : ℙ PERSON
├───────────────────────────────────────────────────
│ {p?} ∪ cousins! ⊆ dom sex
│
│ let cosrel == (parent^(nth?+1) ; (parent⁻¹)^(nth?+1+rem?)) \ (parent ; parent⁻¹) •
│     cousins! = cosrel({p?}) ∪ cosrel⁻¹({p?})
└───────────────────────────────────────────────────
```

11.4 In conclusion

In this chapter, we have looked at two outline specifications which are somewhat more complex than those we have met previously. However, we have not introduced much new notation; we have simply applied our previous knowledge to some harder problems. Hopefully these examples will serve to demonstrate the power of the Z notation to model complex situations succinctly, clearly and precisely. If you are having difficulty understanding them, it may help to draw pictures of typical values of the state variables, to enable you to visualise what is going on. If you understand the examples at first viewing that's fine, but if not, don't be disheartened! Complex problems require a lot of study, and if it was easy it wouldn't be rewarding. When you can understand, modify and extend the above examples, you will be very well prepared for further study of the Z language.

Farklı yapılabilirdi

$temp = (parent^{nth?+1}\,;(parent^{-1})^{nth+1+rem?}) - (parent\,;parent^{-1})$

$cousins! = \{cousin : PERSON \mid p? \to cousin \in temp \bullet cousin\}$
$\{cousin : PERSON \mid cousin \to p? \in temp \bullet cousin\}$