CHAPTER 7

Examples of Z specification documents

7.1 Introduction

We now re-express the aircraft example of Chapter 3 as a Z specification using schemas. This specification concerns recording the passengers aboard an aircraft. There are no seat numbers; passengers are allowed aboard on a first-come-first-served basis.

7.2 The types

The only *basic type* involved here is the set of all possible persons, *PERSON*:

[PERSON] the set of all possible uniquely identified persons

The aircraft has a fixed capacity:

| capacity: $\mathbb{N}$

7.3 The state

The state of the system is given by the set of persons on board the aircraft. The number of persons on board must never exceed the capacity. This is the state's *invariant* property.

```
┌─Aircraft──────────────────
│ onboard:      ℙPERSON
├───────────────────────────
│ #onboard ≤ capacity
└───────────────────────────
```

The state before and after an operation is described by the schema Δ*Aircraft*, which has its conventional meaning:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\_\_\Delta Aircraft\_\_\_\_\_\_\_\_\_\_\_\_} \\
onboard: & \mathbb{P}PERSON \\
onboard': & \mathbb{P}PERSON \\
\hline
\multicolumn{2}{|l}{\#onboard \le capacity} \\
\multicolumn{2}{|l}{\#onboard' \le capacity} \\
\hline
\end{array}
$$

7.4 Initialisation operation

There must be an initial state for the system. The obvious one is where the aircraft is empty. A suitable initialisation operation sets a new value to the variable:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_Init\_\_\_\_\_\_\_\_\_\_\_\_} \\
Aircraft' \\
\hline
onboard' = \emptyset \\
\hline
\end{array}
$$

The initialised state must satisfy the state's invariant property. This it clearly does, since the size of the empty set is zero, which is less than or equal to all natural numbers and so to all possible values of *capacity*.

7.5 Operations

7.5.1 Boarding

There must be an operation to allow a person *p?* to board the aircraft. A first version of this is called $Board_0$:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\_\_Board_0\_\_\_\_\_\_\_\_\_\_\_\_} \\
\multicolumn{2}{|l}{\Delta Aircraft} \\
p?: & PERSON \\
\hline
\multicolumn{2}{|l}{p? \notin onboard} \\
\multicolumn{2}{|l}{\#onboard < capacity} \\
\multicolumn{2}{|l}{onboard' = onboard \cup \{p?\}} \\
\hline
\end{array}
$$

7.5.2 Disembarking

It is also necessary to have an operation to allow a person *p?* to disembark from the aircraft. A first version of this is $Disembark_0$:

$$
\begin{array}{|l}
\hline
\textit{Disembark}_0 \\
\hline
\Delta Aircraft \\
p?: \quad PERSON \\
\hline
p? \in onboard \\
onboard' = onboard \setminus \{p?\} \\
\hline
\end{array}
$$

7.6 Enquiry operations

These operations leave the state unchanged and therefore use the schema:

$\Xi Aircraft$

which has its (automatic) conventional meaning:

$$
\begin{array}{|l}
\hline
\Xi Aircraft \\
\hline
onboard: \quad \mathbb{P}PERSON \\
onboard': \quad \mathbb{P}PERSON \\
\hline
\#onboard \leq capacity \\
\#onboard' \leq capacity \\
onboard = onboard' \\
\hline
\end{array}
$$

7.6.1 Number on board

In addition to operations which change the state of the system it is necessary to have an operation to discover the number of persons on board:

$$
\begin{array}{|l}
\hline
Number \\
\hline
\Xi Aircraft \\
numOnboard!: \quad \mathbb{N} \\
\hline
numOnboard! = \#onboard \\
\hline
\end{array}
$$

7.6.2 Person on board

Furthermore, a useful enquiry is to discover whether or not a given person *p* ? is on board. The data type *YESORNO* is defined to provide suitable values for the reply and is used in the schema *OnBoard*:

$YESORNO ::= yes \mid no$

```
__OnBoard_______________
| ΞAircraft
| p?:      PERSON
| reply!:  YESORNO
|________________
| (p? ∈ onboard ∧ reply! = yes)
| ∨
| (p? ∉ onboard ∧ reply! = no)
|________________
```

7.7 Dealing with errors

The schemas $Board_0$ and $Disembark_0$ do not state what happens if their preconditions are not satisfied. The schema calculus of Z allows these schemas to be extended. First we define a small schema *OKMessage* to give the reply *OK* in the event of success:

```
RESPONSE ::=
  OK | twoErrors | onBoard | full | notOnBoard
OKMessage == [rep!: RESPONSE | rep! = OK]
```

7.7.1 Boarding

A schema to handle errors *BoardError* is defined. It causes no change to the value of *onboard*, so the schema *ΞAircraft* is used:

```
__BoardError_____________
| ΞAircraft
| p?:      PERSON
| rep!:    RESPONSE
|________________
| (p? ∈ onboard ∧
| #onboard = capacity ∧
| rep! = twoErrors)
| ∨
| (p? ∈ onboard ∧
| #onboard < capacity ∧
| rep! = onBoard)
| ∨
| (p? ∉ onboard
| ∧
| #onboard = capacity ∧
| rep! = full)
|________________
```

Finally, *Board* can be defined:

$Board == (Board_0 \wedge OKMessage) \vee BoardError$

7.7.2 Disembark

```
__DisembarkError ________________
| ΞAircraft
| p?:      PERSON
| rep!:    RESPONSE
|________________________________
| p? ∉ onboard ∧ rep! = notOnBoard
|________________________________
```

Finally *Disembark* can be defined:

$Disembark == (Disembark_0 \wedge OKMessage) \vee DisembarkError$

7.8 Example of schemas: Student Programme of Modules

7.8.1 Introduction

This specification concerns a student on a modular course. The student chooses modules from those offered and constructs a *programme* by *adding* and *deleting modules*. The programme is *viable* if it fulfils certain conditions. At least one viable programme must exist.

7.8.2 Types

[MODULE] the set of all possible modules (module identifications)

7.8.3 Sets

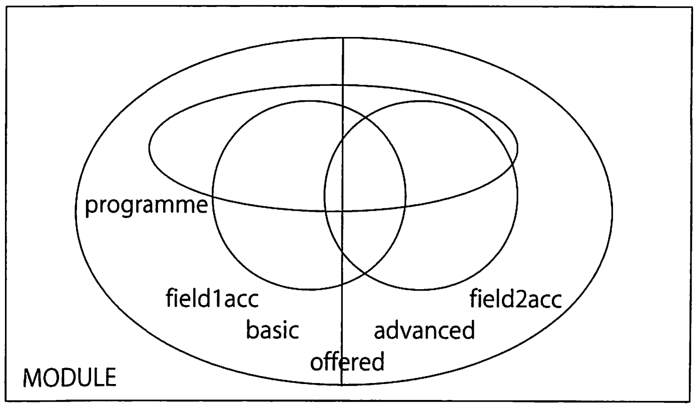
```
| offered, advanced, basic,
| field1acc, field2acc: ℙMODULE
|________________________________
| advanced ∩ basic = ∅
| advanced ∪ basic = offered
| field1acc ⊆ offered
| field2acc ⊆ offered
| #offered ≥ 18
| #(field1acc ∩ advanced) ≥7
| #(field2acc ∩ advanced) ≥ 7
| #((field1acc ∪ field2acc) ∩
| advanced) ≥ 16
```

Certain modules are *offered*. An offered module is either *basic* or *advanced* (not both). Certain offered modules are deemed to be *acceptable* to *field1* and certain to *field2*. A module may be acceptable to more than one field or to none.

For there to be at least one *viable* programme of modules, there must be at least 18 offered modules, at least seven offered that are advanced and acceptable to *field1*, at least seven offered that are advanced and acceptable to *field2*, and at least 16 offered that are advanced and acceptable to the field combination.

The following Venn diagram shows the relationships between the sets in this specification:

Figure 7.1



7.8.4 State

The schema *Student* keeps information on the modules in one student's programme. These may only ever be modules that are *offered*.

Student
$programme : \mathbb{P}MODULE$

$programme \subseteq offered$

7.8.5 Initialisation operation

Initially the student has no modules in their programme.

Init
$Student'$

$programme' = \varnothing$

7.8.6 Operations

Adding a module

```
Add0
ΔStudent
m?: MODULE
─────────────
m? ∈ offered
m? ∉ programme
programme' = programme ∪ {m?}
```

The student may only add a module that is offered. The module should not already be in the student's programme. The module is added to the student's programme.

7.8.7 Deleting a module

The module must be in the student's programme. It is deleted from the programme.

```
Delete0
ΔStudent
m?: MODULE
─────────────
m? ∈ programme
programme' = programme \ {m?}
```

7.8.8 Enquiries

YESORNO ::= yes | no

Viable programme

```
Viable
ΞStudent
reply!: YESORNO
─────────────
(#programme ≥18 ∧
#(programme ∩ field1acc ∩ advanced) ≥ 7 ∧
#(programme ∩ field2acc ∩ advanced) ≥ 7 ∧
#(programme ∩ (field1acc ∪ field2acc) ∩ advanced) ≥ 16 ∧
reply! = yes)
∨
(¬(#programme ≥ 18 ∧
   #(programme ∩ field1acc ∩ advanced) ≥ 7 ∧
   #(programme ∩ field2acc ∩ advanced) ≥ 7 ∧
   #(programme ∩ (field1acc ∪ field2acc) ∩ advanced) ≥ 16) ) ∧
  reply! = no)
```

To be viable, the student's programme must have at least 18 offered modules, at least seven offered that are advanced and acceptable to *field1*, at least seven offered that are advanced and acceptable to *field2*, and at least 16 offered that are advanced and acceptable to the field combination.

7.8.9 Error operations

RESPONSE ::= OK | noSuchModule | alreadyRegistered | notRegistered

7.8.10 Error in adding

```
__AddError__________________________
| ΞStudent
| m?: MODULE
| resp!: RESPONSE
|------------------------------------
| (m? ∉ offered ∧ resp! = noSuchModule)
| ∨
| (m? ∈ programme ∧ resp! = alreadyRegistered)
|____________________________________
```

If the module is not offered, then the message *noSuchModule* is issued. If the module is already in the programme, then the message *alreadyRegistered* is given. In either case the state remains unchanged.

7.8.11 Error in deleting

```
__DeleteError_______________________
| ΞStudent
| m?: MODULE
| resp!: RESPONSE
|------------------------------------
| m? ∉ programme ∧ resp! = notRegistered
|____________________________________
```

If the module to be deleted is not in the student's programme, then the message *notRegistered* is given and the state remains unchanged.

7.8.12 Final versions of operations

OKMessage ::= [resp!: RESPONSE | resp! = OK]

Add == (Add_0 ∧ OKMessage) ∨ AddError

Delete == ($Delete_0$ ∧ OKMessage) ∨ DeleteError

7.8.13 Note

An alternative way of dealing with viability is to construct the set of all *viable programmes*. This will be a set of sets of modules. To say that there must be at least one viable programme it is enough to say that the set of viable programmes is not empty. The section *Sets* can be rewritten:

$$
\begin{array}{|l}
\text{offered, advanced, basic,} \\
\text{field1acc, field2acc: } \mathbb{P}\text{MODULE} \\
\text{viableProgrammes} : \mathbb{P}\mathbb{P}\text{MODULE} \\
\hline
\text{advanced} \cap \text{basic} = \varnothing \\
\text{advanced} \cup \text{basic} = \text{offered} \\
\text{field1acc} \subseteq \text{offered} \\
\text{field2acc} \subseteq \text{offered} \\
\text{viableProgrammes} = \{\text{viaProg}: \mathbb{P}\text{MODULE} \mid \\
\quad \#\text{viaProg} \geq 18 \land \\
\quad \#(\text{viaProg} \cap \text{field1acc} \cap \text{advanced}) \geq 7 \land \\
\quad \#(\text{viaProg} \cap \text{field2acc} \cap \text{advanced}) \geq 7 \land \\
\quad \#(\text{viaProg} \cap (\text{field1acc} \cup \text{field2acc}) \cap \text{advanced}) \geq 16 \bullet \text{viaProg}\} \\
\text{viableProgrammes} \neq \varnothing
\end{array}
$$

Now the enquiry *Viable* can just test whether the student's programme is a member of the set of viable programmes:

$$
\begin{array}{|l}
\text{Viable} \\
\hline
\Xi\text{Student} \\
\text{reply!: YESORNO} \\
\hline
(\text{programme} \in \text{viableProgrammes} \land \text{reply!} = \text{yes}) \\
\lor \\
(\text{programme} \notin \text{viableProgrammes} \land \text{reply!} = \text{no}) \\
\hline
\end{array}
$$

EXERCISES

Using the style of this chapter, create the following components of a formal specification for the computer example of Question 1, Chapter 2, and later.

1. The types and the schema for the state.
2. The operation to add a user.
3. The operation to remove a user.
4. The operation to log in.
5. The operation to log out.