

2.13 Some More Complete Example Specifications

2.13.1 Hints and Suggestions on Creating Specifications

(1) Types:

- Find out what the sets (types) in the system are and keep them *general*. For example,

(a) [STUDENT] the set of students at this university

Because this declaration fixes the type, it does not allow people to be enrolled or to leave, which is clearly not satisfactory.

For this reason, it would be much better to use the more general type

(b) [PERSON] the set of all persons

and then to define

```
students:PPERSON
```

- Also, ensure that types are *atomic*. For example,

```
class:PPERSON
```

is much better than declaring a type

```
[CLASS]
```

(2) Relationships: What relations are there between types? Are any of these relations functions? Are any such functions total, partial, injective, etc?

Are there any relationships between the functions and relations identified? For example, have they the same domain, or is the range of one equal to the domain of another, etc?

Is there a natural order to some values? If so, consider sequences, though sets are simpler to deal with.

(3) State: The investigation of relationships should lead to a schema representing the state. There may also be some invariants discovered, to be included in this schema.

(4) Initialisation operation: Initial state to be as simple as possible – often empty sets and relations. Must not violate any system invariants, something which should, ideally, be formally proved.

(5) Operations: Start by considering the behaviour of each operation only when it is given “sensible” values, including checks for such values.

Later, define a separate schema to deal with errors by returning a reply value.

Don’t forget to state explicitly what does NOT change.

(6) Enquiry Operations: Use \exists where possible. As state cannot change, invariants cannot be violated.

(7) When in difficulty ...: If specification is becoming too difficult, try viewing it more abstractly. Hide detail for now and take a broader view. Put detail back in later when there is a better understanding.

2.13.2 Aircraft Seat Allocation – Extension to Several Flights

In “Section 2.11.4.3 Aircraft passenger seat allocation” a specification was developed for the case of seat allocation on one flight of one aircraft. We refer to this as **Spec2.11.4.3**.

We now extend **Spec2.11.4.3** to a more general specification dealing with many flights. We retain the types from **Spec2.11.4.3** but will also introduce a new type.

(a) Type of a flight:

In **Spec2.11.4.3** we had

[PERSON] the set of all possible uniquely defined persons
[SEAT] set of all seats on this aircraft

In our extended specification, **we re-define** SEAT as

[SEAT] the set of all seat numbers

because now there are several aircraft involved and the available seat numbers will vary according to the aircraft type.

We now introduce

[FLIGHT] the set of all flight identifications¹

(b) State:

• A function to a function:

In **Spec2.11.4.3** seat bookings on a flight were represented by

Seating

bookedTo: SEAT \leftrightarrow PERSON

In the extended system of seat bookings, the behaviour across a fleet of aircraft is to be the same for each flight, so the new seat booking information can be conveniently

¹ If the flight identification is composed of a date and a flight number it could be declared as
[DATE] the set of all dates
[FLIGHTNO] the set of all flight numbers
FLIGHT == DATE×FLIGHTNO (where == means abbreviation definition).
However, we will not use this in our solution.

represented by a function from flight to seat bookings for that flight. The state schema will use the function *booked*:

$$\text{booked: FLIGHT} \rightarrow (\text{SEAT} \rightarrow \text{PERSON})$$

So, for any flight *f* (that is, *f*: FLIGHT), the function

$$\text{booked } f$$

is a function from SEAT to PERSON (just as “bookedTo” in **Spec2.11.4.3**).

Then, for any seat *s*:SEAT,

$$\text{booked } f \text{ } s$$

is the person to whom seat *s* is booked on flight *f*. (In **Spec2.11.4.3**, “bookedTo *s*” is the person to whom seat *s* is booked).

- Seat numbers allocated to a flight: We need a new relation to discover what seat numbers have been assigned to a flight. (Seat is related to flight rather than aircraft as the same aircraft seat will be booked many times during its lifetime but only once for a given flight). Our relation is named *hasSeat*.
- Hence, the schema representing the system state is

<i>FleetSeatAllocation</i>	
<i>booked</i> :	$\text{FLIGHT} \rightarrow (\text{SEAT} \rightarrow \text{PERSON})$
<i>_hasSeat_</i> :	$\text{FLIGHT} \leftrightarrow \text{SEAT}$
$\text{dom } \text{booked} = \text{dom } \text{hasSeat}$	
$\forall f:\text{FLIGHT} \setminus f \in \text{dom } \text{hasSeat} \bullet \text{dom}(\text{booked } f) \subseteq \text{hasSeat}(\{f\})$	

Only the seat numbers assigned to a flight (i.e. $\text{hasSeat}(\{f\})$) may be booked.

(c) Initialisation operation:

In **Spec2.11.4.3** the initialisation operation was

<i>Init</i>	
<i>Seating'</i>	
<i>bookedTo'</i>	$= \emptyset$

In the extended system, a possible initial state is where there are no bookings for any seats on any flights:

*Init**FleetSeatAllocation'**hasSeat'*= \emptyset *booked'*= \emptyset

(d) Operations: In the extended system, there are some operations that have counterparts in **Spec2.11.4.3** (Book_0 and Cancel_0) and some (NewFlight_0) that are specific to the extended system.

- Creating a new flight:

NewFlight₀ Δ *FleetSeatAllocation**f?*: *FLIGHT**available?*: $\mathbb{P}SEAT$ *f?* $\notin \text{dom } booked$ *available?* $\neq \emptyset$ *hasSeat'*=*hasSeat* $\cup \{s:SEAT \mid s \in available? \bullet f? \mapsto s\}$ *booked'*=*booked* $\cup \{f? \mapsto \emptyset\}$

The flight must not already be known. The set of seats available to that flight must not be empty. The seat bookings for the new flight are initially empty.

Note: It can be seen that the constraint $\text{dom } booked = \text{dom } hasSeat$ has not been violated.

- Booking a seat on a flight:

In **Spec2.11.4.3** the corresponding operation was

Book₀ Δ *Seating**p?*: *PERSON**s?*: *SEAT**s?* $\notin \text{dom } bookedTo$ *bookedTo'* = *bookedTo* $\cup \{s? \mapsto p?\}$

In the extended system, we have

<i>Book</i> ₀
$\Delta FleetSeatAllocation$ $f?: FLIGHT$ $p?: PERSON$ $s?: SEAT$
$f? \in \text{dom } booked$ $f? \text{ hasSeat } s?$ $s? \notin \text{dom } booked f?$ $booked' = booked \oplus \{f? \mapsto (booked f? \cup \{s? \mapsto p?\})\}$ $hasSeat' = hasSeat$

The flight must be known, the seat must be one that is assigned to that flight, and the seat must not be booked already.

- Cancelling a booking on a flight:

In **Spec2.11.4.3** the corresponding operation was

<i>Cancel</i> ₀
$\Delta Seating$ $p?: PERSON$ $s?: SEAT$
$s? \mapsto p? \in bookedTo$ $bookedTo' = bookedTo \setminus \{s? \mapsto p?\}$

In the extended system, we have

<i>Cancel</i> ₀
$\Delta FleetSeatAllocation$ $f?: FLIGHT$ $p?: PERSON$ $s?: SEAT$
$f? \in \text{dom } booked$ $f? \text{ hasSeat } s?$ $s? \mapsto p? \in \text{dom } booked f?$ $booked' = booked \oplus \{f? \mapsto (booked f? \setminus \{s? \mapsto p?\})\}$ $hasSeat' = hasSeat$

The flight must be known, the seat must be one that is assigned to that flight, and the seat must be booked to the person.

(e) Enquiry: One enquiry is specified, corresponding to “WhoseSeat” in **Spec2.11.4.3**, for which the schema was

<i>WhoseSeat</i>
\exists <i>Seating</i> <i>s?</i> : <i>SEAT</i> <i>taken!</i> : <i>REPLY</i> <i>who!</i> : <i>PERSON</i>
$(s? \in \text{dom } \textit{bookedTo} \wedge \textit{taken!} = \textit{yes} \wedge \textit{who!} = \textit{bookedTo } s?)$ \vee $(s? \notin \text{dom } \textit{bookedTo} \wedge \textit{taken!} = \textit{no})$

where $\textit{REPLY} ::= \textit{yes} | \textit{no}$.

The corresponding schema in the extended specification is

<i>Enquire</i>
\exists <i>FleetSeatAllocation</i> <i>f?</i> : <i>FLIGHT</i> <i>s?</i> : <i>SEAT</i> <i>taken!</i> : <i>REPLY</i> <i>p!</i> : <i>PERSON</i>
$f? \in \text{dom } \textit{booked}$ $f? \textit{ hasSeat } s?$ $((s? \in \text{dom } \textit{booked } f? \wedge \textit{taken!} = \textit{yes} \wedge \textit{who!} = \textit{booked } f? \textit{ } s?)$ \vee $(s? \notin \text{dom } \textit{booked } f? \wedge \textit{taken!} = \textit{no}))$

Thus, the flight must be known and the seat must be assigned in this flight.

(f) Error Conditions: Error handling schemas could be introduced in a straightforward way, as in previous examples.

Note: The extended specification is a good deal more complex than earlier examples. However, its formulation was simplified by the fact that a specification for a single flight had been already worked out. **This is a sound approach in general, that is, specify a simple version of a problem first before taking on its entire complexity.**

2.13.3 Airport gate management

The air-traffic control of an airport keeps a record of

- planes waiting to land
- assignment of planes to gates on the ground

Remark: Effectively, these records constitute the state of the system.

There are operations to

- accept a plane when it arrives in the airport's waiting space
- assign a plane to a gate at the airport
- record that a plane leaves its gate

Types: [PLANE] the set of all possible, uniquely defined planes
 [GATE] the set of all gates at this airport

State:

Airport

waiting: $\mathbb{P}PLANE$

assignment: $GATE \rightsquigarrow PLANE$

$waiting \cap \text{ran } assignment = \emptyset$

Each plane is assigned to at most one gate & each gate has at most one plane assigned to it (partial injective function). The planes waiting are a set of planes; thus, there is no particular order, something that may be unrealistic in practice. No plane is both waiting and assigned to a gate.

Initialisation operation: Initially no planes waiting or at any gate.

Init

Airport'

waiting' = \emptyset

assignment' = \emptyset

Operations:

(a) Arriving plane

$Arrive_0$ $\Delta Airport$ $p?: PLANE$
$p? \notin (waiting \cup ran\ assignment)$ $waiting' = waiting \cup \{p?\}$ $assignment' = assignment$

The plane must be neither waiting nor assigned to a gate. The new arrival is added to those waiting. The assignment of gates to planes is not affected.

(b) Assigning a plane to a gate

$Assign_0$ $\Delta Airport$ $p?: PLANE$ $g?: gate$
$p? \in waiting$ $g? \notin dom\ assignment$ $waiting' = waiting \setminus \{p?\}$ $assignment' = assignment \cup \{g? \mapsto p?\}$

The plane must be waiting and the gate must be free. The pairing of $g?$ and $p?$ must be added to assignment, and the plane is no longer waiting.

(c) A leaving plane

$Leave_0$ $\Delta Airport$ $p?: PLANE$
$p? \in ran\ assignment$ $assignment' = assignment \triangleright p?$ $waiting' = waiting$

The plane must be assigned to a gate. The assignment of a gate to this plane must be removed and the waiting set of planes is unaffected.

Operations for error handling:

Similar to previous examples, we introduce the free type

RESULT ::= OK | full | badAircraft | notWaiting | gateNotFree | notAtgate

and the schema

OKMessage == [reply! : RESULT | reply! = OK]

We also define a global variable

$$\left| \begin{array}{l} \textit{Limit}: \mathbb{N} \end{array} \right.$$

representing the maximum number of waiting planes that can be accommodated. Then we define error handling schemas and complete operations for the three cases considered.

(a) Arriving plane

$\begin{array}{l} \textit{ArriveErr} \\ \hline \exists \textit{Airport} \\ p?: \textit{PLANE} \\ \textit{reply!}: \textit{RESULT} \\ \hline \# \textit{waiting} = \textit{limit} \wedge \textit{reply!} = \textit{full} \\ \vee \\ p? \in (\textit{waiting} \cup \textit{ran assignment}) \wedge \textit{reply!} = \textit{badAircraft} \end{array}$
--

Then the complete operation is, say,

arrive == (Arrive₀ ∧ OKMessage) ∨ ArriveErr

(b) Assigning a plane to a gate

$\begin{array}{l} \textit{AssignErr} \\ \hline \exists \textit{Airport} \\ p?: \textit{PLANE} \\ g?: \textit{gate} \\ \textit{reply!}: \textit{RESULT} \\ \hline p? \notin \textit{waiting} \wedge \textit{reply!} = \textit{notWaiting} \\ \vee \\ g? \in \textit{dom assignment} \wedge \textit{reply!} = \textit{gateNotFree} \end{array}$

Then the complete operation is, say,

Assign == (Assign₀ ∧ OKMessage) ∨ AssignErr

(c) A leaving plane

$LeaveErr$ $\exists Airport$ $p?: PLANE$ $reply!: RESULT$
$p? \notin ran\ assignment \wedge reply! = notAtGate$

Then the complete operation is, say,

$$Leave == (Leave_{e_0} \wedge OKMessage) \vee LeaveErr$$

2.13.4 Library

The basic system is that a library has a stock of books which may be taken out by its registered members. We first present a specification of this basic system.

Next, we present three extensions of the basic system:

1. Limit on the number of books that a member may take out
2. Books out for a limited period
3. Reservations

2.13.4.0 Basic system (0)

Types:

- [Book] the set of all possible uniquely identified books
 [PERSON] the set of all possible persons

State:

Lib_0 $stock: \mathbb{P}BOOK$ $members: \mathbb{P}PERSON$ $outTo: BOOK \leftrightarrow PERSON$
$dom\ outTo \subseteq stock$ $ran\ outTo \subseteq members$

Only books that are in the library's stock can be loaned out and only registered members can take a book out.

Initialisation operation: Initially, no members, stock or loans.

<i>SchemaName</i>
<i>Lib</i> ₀ '
<i>stock</i> '= \emptyset
<i>members</i> '= \emptyset
<i>outTo</i> '= \emptyset

Operations:

- Acquire book:

<i>Acquire</i> ₀
Δ <i>Lib</i> ₀
<i>b</i> ? : <i>BOOK</i>
<i>b</i> ? \notin <i>stock</i>
<i>stock</i> '= <i>stock</i> \cup { <i>b</i> ?}
<i>members</i> '= <i>members</i>
<i>outTo</i> '= <i>outTo</i>

Book is not already in stock - it is added; membership and loans unchanged.

- Register member:

<i>Register</i> ₀
Δ <i>Lib</i> ₀
<i>p</i> ? : <i>PERSON</i>
<i>p</i> ? \notin <i>members</i>
<i>stock</i> '= <i>stock</i>
<i>members</i> '= <i>members</i> \cup { <i>p</i> ?}
<i>outTo</i> '= <i>outTo</i>

Person is not already a member - is added; stock and loans unchanged.

- Take a book out on loan:

<i>TakeOut₀</i>
ΔLib_0 $p?: PERSON$ $b?: BOOK$
$p? \in members$ $b? \in stock \setminus dom\ outTo$ $stock' = stock$ $members' = members$ $outTo' = outTo \cup \{b? \mapsto p?\}$

Person must be a member and the book must be in stock and not already on loan.

Loan is recorded, and stock and membership are unchanged.

- Bring back (return) a book:

<i>BringBack₀</i>
ΔLib_0 $b?: BOOK$
$b? \in dom\ outTo$ $stock' = stock$ $members' = members$ $outTo' = \{b?\} \triangleleft outTo$

Book must be on loan and is now recorded as back. Stock and membership are unchanged.

- Dispose of a book:

<i>Dispose₀</i>
ΔLib_0 $b?: BOOK$
$b? \in stock \setminus dom\ outTo$ $stock' = stock \setminus \{b?\}$ $members' = members$ $outTo' = outTo$

Book must be part of stock but not on loan. It is removed from the stock, and membership and loan records are unchanged.

- De-register member:

<i>Deregister</i> ₀
ΔLib_0
$p?: PERSON$
$p? \in members \setminus ran\ outTo$
$stock' = stock$
$members' = members \setminus \{p?\}$
$outTo' = outTo$

Person is a member and does not have books on loan. Person is removed from membership, and stock and loan record are unchanged.

2.13.4.1 Extension 1: Limit on number of books out

Note: Recall that it can sometimes be useful to re-use a schema in a different to its original context. As a reminder, we previously had an aircraft example,

<i>Aircraft</i>
$onboard: \mathbb{P}Person$
$\#onboard \leq capacity$

and then

$Ship \equiv Aircraft[passengers/onboard]$

which is equivalent to

<i>Ship</i>
$passengers: \mathbb{P}Person$
$\#passengers \leq capacity$

We use re-naming in the following.

Global variable:

First, we define the limit as a global variable, the same for all members, using the axiomatic definition

$limit: \mathbb{N}$
$limit \in 1..10$

The limit lies between 1 and 10 (chosen arbitrarily).

State: The state is the same as Lib_0 except that the limit constraint must be specified:

$$\begin{array}{l} \overline{Lib_1} \\ \overline{Lib_0} \\ \hline \forall p:PERSON | p \in members \bullet \#(outTo \triangleright \{p\}) \leq limit \end{array}$$

Initialisation operation: This is the same as before except that it applies now to Lib_1 .

This can be specified using schema re-naming:

$$Init_1 == Init_0[Lib_1'/Lib_0']$$

Note that the initial state satisfies the new state invariant.

Operations: All but 1 of the operations are unaffected by the new requirement:

$$Acquire_1 == Acquire_0[\Delta Lib_1/\Delta Lib_0]$$

$$Register_1 == Register_0[\Delta Lib_1/\Delta Lib_0]$$

$$BringBack_1 == BringBack_0[\Delta Lib_1/\Delta Lib_0]$$

$$Dispose_1 == Dispose_0[\Delta Lib_1/\Delta Lib_0]$$

$$Deregister_1 == Deregister_0[\Delta Lib_1/\Delta Lib_0]$$

However, the “taking out” operation does change, as it must be checked that the limit is not exceeded. This can be expressed succinctly as follows:

$$\begin{array}{l} \overline{TakeOut_1} \\ \overline{TakeOut_0[\Delta Lib_1/\Delta Lib_0]} \\ \hline \#(outTo \triangleright \{p?\}) < limit \end{array}$$

Thus, the number of books that are out to the person must not yet have reached the limit.

2.13.4.2 Extension 2: Books out for a limited period

Next, we consider a library where not only is there a limit on the number of books that can be borrowed by a member but also there is a limit on how long a book can be borrowed for. There is no charge for taking out books but a fine is payable for late return.

Types & Global variables & invariants: A new type DATE is needed and money is represented as whole numbers of currency units, possibly negative:

[DATE] the set of all possible dates

MONEY == \mathbb{Z}

All members may take out books for the same (maximum) period of days:

$period: \mathbb{N}$

Books returned late incur the same fine each day:

$fine: MONEY$
$fine \geq 0$

The State: The date that each book is taken out on must be recorded, as must be the amount (possibly zero) owed by each member:

Lib_2
Lib_1
$dateOut: Book \rightarrow DATE$
$owes: PERSON \rightarrow MONEY$
$dom\ dateOut = dom\ outTo$
$dom\ owes = members$

Notice the invariant constraints on the two functions introduced.

Initialisation operation: This is the same as for Lib_1 , with the addition that initially no information is recorded about when books were taken out or about money owing:

Note that the initial state satisfies the new state invariant.

$Init_2$
$Init_1[Lib_2'/Lib_1']$
$dateOut' = \emptyset$
$owes' = \emptyset$

Operations: Two operations are unaffected by the new requirements:

$Acquire_2 == Acquire_1[\Delta Lib_2 / \Delta Lib_1]$

$Dispose_2 == Dispose_1[\Delta Lib_2 / \Delta Lib_1]$

However, the other four operations are changed to some extent, as follows.

(a) Register member: A new member does not owe anything,

$Register_2$
$Register_1[\Delta Lib_2 / \Delta Lib_1]$
$owes' = owes \cup \{p? \mapsto 0\}$
$dateOut' = DateOut$

(b) De-register a member: Specify that to be de-registered a member must owe nothing,

<i>Deregister₂</i>
<i>Deregister₁</i> [$\Delta Lib_2/\Delta Lib_1$]
<i>owes</i> $p?$ = 0
<i>dateOut'</i> = <i>DateOut</i>

(c) Taking out a book: The date of taking out must be recorded,

<i>TakeOut₂</i>
<i>TakeOut₁</i> [$\Delta Lib_2/\Delta Lib_1$]
<i>d?</i> : DATE
<i>owes</i> = <i>owes'</i>
<i>dateOut'</i> = <i>DateOut</i> \cup { <i>b?</i> \mapsto <i>d?</i> }

(d) Bringing back a book: This requires the most change, as would be expected. First, we specify a global function, through an axiomatic definition,

$$\left| \text{Diff}: \text{DATE} \times \text{DATE} \rightarrow \mathbb{Z} \right.$$

to be such that $\text{Diff}(d_1, d_2)$ returns the number of working days that d_2 is later than d_1 . In the new operation BringBack_2 there is no change in what the member owes if the book is returned within the period. However, if the book is returned late then the member's debt, calculated only when the book is returned, is increased by a fixed fine for each day late.

<i>BringBack₂</i>
<i>BringBack₁</i> [$\Delta Lib_2/\Delta Lib_1$]
<i>today?</i> : DATE
<i>dateOut'</i> = { <i>b?</i> } \Leftarrow <i>DateOut</i> \wedge ($\text{Diff}(\text{dateOut } b?, \text{today}?) \leq \text{period} \wedge \text{owes}' = \text{owes}$)
\vee ($\text{Diff}(\text{dateOut } b?, \text{today}?) > \text{period} \wedge$ $\text{owes}' = \text{owes} \oplus \{\text{outTo } b? \mapsto \text{owes outTo } b? + (\text{Diff}(\text{dateOut } b?, \text{today}?) - \text{period}) * \text{fine}\}$)

The date that the book was taken out is removed from the record.

If the difference between to-day's date and the date borrowed is within period, the amount owed by the member is unaffected.

However, if the difference between to-day's date and the date borrowed exceeds the allowed period, then the amount owed is increased by the fine amount for each day over the period.

(e) Paying in: It makes sense, in this library, to provide an operation such that a member can pay to offset current or future money owed.

$PayIn_2$ ΔLib_2 ΞLib_1 $p?: PERSON$ $amount?: MONEY$
$p? \in members$ $amount? > 0$ $owes' = owes \oplus \{p? \mapsto owes p? - amount?\}$ $dateOut' = DateOut$

Notice the use of Ξ to specify that everything else is unchanged.

2.13.4.3 Extension 3: Reservations

Finally, we extend the previous specification to allow members to reserve titles. There is a basic distinction to be made between an individual (physical) book and its title in that there may be several, presumably identical, physical copies of the same title. For the purpose of reservation, it is the title that is important.

Types: We have a new type

[TITLE] the set of all possible book titles

The State: The title of every book in stock is known. Reservations are allowed for every title for which there is a book in stock. Only members may reserve titles and they can only reserve each title once. A book which belongs to the library and which is not out may be held for a member who has previously reserved it (and will collect it later).

A number of new functions are introduced to capture this information:

Lib_3 <hr/> Lib_2 <i>title</i> : $Book \rightarrow TITLE$ <i>reserved</i> : $TITLE \rightarrow \text{iseq } PERSON$ <i>heldFor</i> : $BOOK \rightarrow PERSON$ <hr/> dom <i>title</i> = <i>stock</i> dom <i>reserved</i> = ran <i>title</i> $(\forall t: TITLE t \in \text{dom } reserved \bullet \text{ran}(reserved\ t) \subseteq \text{members})$ dom <i>heldFor</i> $\subseteq stock \setminus \text{dom } OutTo$ ran <i>heldFor</i> $\subseteq \text{members}$
--

Initialisation operation: There are no titles recorded, no titles reserved, and no books held for members.

$Init_3$ <hr/> $Init_2[Lib_3'/Lib_2']$ <hr/> <i>title</i> ' = \emptyset <i>reserved</i> ' = \emptyset <i>heldFor</i> ' = \emptyset
--

Operations: All but 1 of the existing operations are unaffected by the new requirement:

$Register_3 == Register_2[\Delta Lib_3 / \Delta Lib_2]$

$Deregister_3 == Deregister_2[\Delta Lib_3 / \Delta Lib_2]$

$TakeOut_3 == TakeOut_2[\Delta Lib_3 / \Delta Lib_2]$

$BringBack_3 == BringBack_2[\Delta Lib_3 / \Delta Lib_2]$

$Dispose_3 == Dispose_2[\Delta Lib_3 / \Delta Lib_2]$

$PayIn_3 == PayIn_2[\Delta Lib_3 / \Delta Lib_2]$

(a) However, the acquisition operation must change in that the title of a book must be recorded when it is acquired. Moreover, if the title is a new one then an empty reservation list must be set up for it.

$Acquire_3$ <hr/> $Acquire_2[\Delta Lib_3/\Delta Lib_2]$ $f?:title$ <hr/> $title'=title \cup \{b? \mapsto t?\}$ $(f? \notin dom\ reserved \wedge reserved' = reserved \cup \{f? \mapsto \diamond\})$ \vee $(f? \in dom\ reserved \wedge reserved' = reserved)$ $heldFor'=heldFor$
--

(b) In addition, a new operation must be specified for a person to make a reservation. For this, the person must be a member and must not have reserved the title already. The library must have a book of this title in stock.

$Reserve_3$ <hr/> ΔLib_3 $\exists Lib_2$ $p?: PERSON$ $t?: TITLE$ <hr/> $p? \in members$ $p? \notin ran(reserved\ t?)$ $t? \in ran\ title$ $reserved' = reserved \oplus \{t? \mapsto reserved\ t? \wedge \langle p? \rangle\}$ $title'=title$ $heldFor'=heldFor$
--

(c) Finally, an operation must be specified to hold a book for a member who has previously reserved it. The book must not be out (borrowed). (see **Problem Sheet 4**).