

Concordia University  
Department of Computer Science and Software  
Engineering  
**SOEN 331:**  
**Formal Methods for Software Engineering**

**Assignment 2 Richard Badir 40249566**  
**Valentin Gornostaev 40211600**

March 27, 2024

## Problem 2

1. CL-USER 1 > (defvar \*airports\* '((YUL.Montreal) (LCY.London\_UK) (LHR.London\_UK) (MIL.MILAN) (SFO.SAN\_FRANCISCO) (SD.SAN\_DIEGO))  
\*AIRPORTS\*

CL-USER 2 > (print \*airports\*)

((YUL.MONTREAL) (LCY.LONDON\_UK) (LHR.LONDON\_UK) (MIL.MILAN) (SFO.SAN\_FRANCISCO) (SD.SAN\_DIEGO))  
((YUL.MONTREAL) (LCY.LONDON\_UK) (LHR.LONDON\_UK) (MIL.MILAN) (SFO.SAN\_FRANCISCO) (SD.SAN\_DIEGO))

CL-USER 3 > \*airports\*

((YUL.MONTREAL) (LCY.LONDON\_UK) (LHR.LONDON\_UK) (MIL.MILAN) (SFO.SAN\_FRANCISCO) (SD.SAN\_DIEGO))

CL-USER 4 >

2.  $\text{monitored} = \text{dom}(\text{airports})$

3. *airports* is a variable that holds one to many mappings of objects of type AIRPORT to objects of type CITY.  $\text{dom}(\text{airports}) = \mathbb{P} \text{ AIRPORT}$ .  $\text{codomain}(\text{airports}) = \mathbb{P} \text{ CITY}$ .  
 $\text{airports} = \mathbb{P} \text{ AIRPORT} \mapsto \text{CITY}$

4. This should be an unordered structure, because order doesn't matter in this scenario, nor does it make sense. Looking up in an unordered structure would also be a lot faster than in an ordered one  $O(1)$  vs  $O(n)$ . Assuming a Hashmap or any variation is not permitted, a set containing tuples of type (AIRPORT, CITY) should be used. There should be a restriction on the domain of this data structure, indicating no head of tuple can be a duplicate of another head. Lists of strict length 2 with types  $\langle \text{AIRPORT}, \text{CITY} \rangle$  could also be used to replace the tuples.

5.

<i>AirportSystem</i>	
<i>codes</i> : $\mathbb{P} \text{ AIRPORT}$	
<i>airports</i> : $\text{AIRPORT} \rightarrow \text{CITY}$	--non-injective, surjective
<i>cities</i> : $\mathbb{P} \text{ CITY}$	
<hr/>	
<i>codes</i> = dom <i>airports</i>	
<i>deployed</i> = ran <i>airports</i>	

6.

<i>Success</i>	
$\exists \text{ AirportSystem}$	
<i>response!</i> : <i>MESSAGE</i>	
<hr/>	
<i>response!</i> = 'ok'	

<i>CodeAlreadyInUse</i>	
$\exists \text{ AirportSystem}$	
<i>code?</i> : <i>AIRPORT</i>	
<i>response!</i> : <i>Message</i>	
<hr/>	
<i>code?</i> $\in$ dom <i>airports</i>	
<i>response!</i> = 'Code already in use'	

<i>AddAirportOK</i>	
$\Delta \text{ AirportSystem}$	
<i>code?</i> : <i>AIRPORT</i>	
<i>city?</i> : <i>CITY</i>	
<hr/>	
<i>code?</i> $\notin$ dom <i>airports</i>	
<i>airports'</i> = <i>airports</i> $\cup$ { <i>code?</i> $\mapsto$ <i>city?</i> }	

$$\text{AddAirport} \hat{=} (\text{AddAirportOK} \wedge \text{Success}) \oplus \text{CodeAlreadyInUse}$$

7. CL-USER 1 > (defvar \*airports\*

```

'((YUL . Montreal)
  (LCY .London_UK)
  (LHR . London_UK)
  (MIL . Milan)
  (SFO . San_Francisco)
  (SDQ . Santo_Domingo)))
*AIRPORTS*

```

```

CL-USER 2 : 1 > *airports*

```

```

((YUL . MONTREAL) (LCY .LONDON_UK) (LHR . LONDON_UK) (MIL . MILAN) (SFO . SAN_FRANC

```

```

CL-USER 3 : 1 > (defun addAirport (code city)

```

```

  (if (find code *airports* :key #'car)
      (print "Code already in use.")
      (progn
        (push (cons code city) *airports*)
        (print "ok"))))

```

```

ADDAIRPORT

```

```

CL-USER 4 : 1 >

```

8.

$UpdateAirportOK$	_____
$\Delta AirportSystem$	
$code? : AIRPORT$	
$city? : CITY$	
$code? \in \text{dom } airports$	
$airports' = airports \oplus \{code? \mapsto city?\}$	

$\frac{CodeNotInUse}{\begin{array}{l} \exists AirportSystem \\ code? : AIRPORT \\ response! : Message \end{array}}$
$\frac{}{\begin{array}{l} code? \notin \text{dom airports} \\ response! = 'Code not in use' \end{array}}$

$$UpdateAirport \hat{=} (UpdateAirportOK \wedge Success) \oplus CodeNotInUse$$

9.  $airports = \{ YUL \mapsto Dorval,$   
 $LCY \mapsto LondonUK,$   
 $LHR \mapsto LondonUK,$   
 $MIL \mapsto Milan,$   
 $SFO \mapsto SanFrancisco,$   
 $SDQ \mapsto SantoDomingo$   
 $\}$

10. CL-USER 1 > (defvar \*airports\*  
'((YUL . Montreal)  
(LCY . London\_UK)  
(LHR . London\_UK)  
(MIL . Milan)  
(SFO . San\_Francisco)  
(SDQ . Santo\_Domingo)))  
\*AIRPORTS\*

CL-USER 2 > (defun updateAirport (code new-city)  
(let ((existing (assoc code \*airports\* :test #'equal)))  
(if existing  
(progn  
(setf (cdr existing) new-city)

```

        (print "ok"))
      (print "Code not in use."))))
UPDATEAIRPORT-CITY

CL-USER 3 > (defun updateAirport (code new-city)
  (let ((existing (assoc code *airports* :test #'equal)))
    (if existing
      (progn
        (setf (cdr existing) new-city)
        (print "ok"))
      (print "Code not in use."))))
UPDATEAIRPORT

CL-USER 4 > (updateAirport 'YUL 'Dorval)

"ok"
"ok"

CL-USER 5 > (updateAirport 'LL 'GG)

"Code not in use."
"Code not in use."

CL-USER 6 >

```

11.

$\text{DeleteAirportOK}$ $\Delta \text{AirportSystem}$ $code? : \text{AIRPORT}$ <hr/> $code? \in \text{dom } airports$ $airports' = \{code?\} \triangleleft airports$
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>CodeNotInUse</i>
$\exists \text{AirportSystem}$
$code? : \text{AIRPORT}$
$response! : \text{Message}$
$code? \notin \text{dom airports}$
$response! = \text{'Code not in use'}$

$$\text{DeleteAirport} \hat{=} (\text{DeleteAirportOK} \wedge \text{Success}) \oplus \text{CodeNotInUse}$$

12.

**Remove Everything After this line** \_\_\_\_\_

## Part 1: Temperature monitoring system with the Z specification

Consider a system called 'TempMonitor' that keeps a number of sensors, where each sensor is deployed in a separate location in order to read the location's temperature. Before the system is deployed, all locations are marked on a map, and each location will be addressed by a sensor. The formal specification of the system introduces the following three types:

*SENSOR\_TYPE, LOCATION\_TYPE, TEMPERATURE\_TYPE*

We also introduce an enumerated type *MESSAGE* which will assume values that correspond to success and error messages.

Provide a formal specification in Z, with the following operations:

- **DeploySensorOK**: Places a new sensor to a unique location. You may assume that some (default) temperature is also passed as an argument.
- **ReadTemperatureOK**: Obtain the temperature reading from a sensor, given the sensor's location.

Provide appropriate success and error schemata to be combined with the definitions above to produce robust specifications for the following interface:

- **DeploySensor**,
- **ReadTemperature**.



Solution:

<i>TempMonitor</i>
$deployed' : \mathbb{P} \text{ SENSOR\_TYPE}$ $map : \text{SENSOR\_TYPE} \rightarrow \text{LOCATION\_TYPE} \quad \text{--partial bijective}$ $read : \text{SENSOR\_TYPE} \rightarrow \text{TEMPERATURE\_TYPE}$
$deployed = \text{dom } map$ $deployed = \text{dom } read$

<i>DeploySensorOK</i>
$\Delta \text{TempMonitor}$ $sensor? : \text{SENSOR\_TYPE}$ $location? : \text{LOCATION\_TYPE}$ $temperature? : \text{TEMPERATURE\_TYPE}$
$sensor? \notin deployed$ $location? \notin \text{ran } map$ $deployed' = deployed \cup \{sensor?\}$ $map' = map \cup \{sensor? \mapsto location?\}$ $read' = read \cup \{sensor? \mapsto temperature?\}$

<i>ReadTemperatureOK</i>
$\exists \text{TempMonitor}$ $location? : \text{LOCATION\_TYPE}$ $temperature! : \text{TEMPERATURE\_TYPE}$
$location? \in \text{ran } map$ $temperature! = read(map^{-1}(location?))$

<i>Success</i>
$\exists \text{TempMonitor}$ $response! : \text{MESSAGE}$
$response! = 'ok'$

<i>SensorAlreadyDeployed</i>
$\exists TempMonitor$
$sensor? : SENSOR\_TYPE$
$response! : Message$
$sensor? \in deployed$
$response! = 'Sensor\ deployed'$

<i>LocationAlreadyCovered</i>
$\exists TempMonitor$
$location? : LOCATION\_TYPE$
$response! : Message$
$location? \in \text{ran } map$
$response! = 'Location\ already\ covered'$

<i>LocationUnknown</i>
$\exists TempMonitor$
$location? : LOCATION\_TYPE$
$response! : Message$
$location? \notin \text{ran } map$
$response! = 'Location\ not\ covered'$

$$DeploySensor \hat{=} (DeploySensorOK \wedge Success) \oplus (SensorAlreadyDeployed \vee LocationAlreadyCovered)$$

$$ReadTemperature \hat{=} (ReadTemperatureOK \wedge Success) \oplus LocationUnknown$$

## Part 2: A booking system with the Object Z specification

We introduce the basic types  $[Person, SeatType]$ . We also introduce an enumerated type *Message* which will assume values (feel free to define your own) that correspond to success and error messages. Consider a system to book seats for a theater play. A customer can book a single seat, and a seat can only accommodate a single customer. The booking system keeps a log of the customers that have booked a seat. The system publishes a plan of the theater and it allows customers to access it online and make a booking or cancel a booking.

### Class Booking

Define a formal specification in Object-Z for class *Bookingt* to support the following operations:

- **BookOK**: Reserves a seat for a given customer.
- **CancelOK**: Frees a seat for a given customer.

You will also need to provide appropriate success and error schemata to be combined with the definitions above to produce *robust specifications* for the following interface:

- **Book**, and
- **Cancel**.

### Class Booking2

Subclassify *Booking* to introduce class **Booking2** that behaves exactly like *Booking*, while introducing the following operations:

- **GetNumberOfCustomers** returns the total number of customers who have made a booking.

- `ModifyBookingOK` assigns an existing customer to a different seat. Provide any additional schema(ta) in order to extend the interface to include a robust operation `ModifyBooking`.

The extended interface will now include operations

- `GetNumberOfCustomers`, and
- `ModifyBooking`.

Solution:

The main functionality for class **Booking** together with the success and error schemata are given below:

<i>Booking</i>
$\uparrow (Book, Cancel)$
$booked : \mathbb{P} Person$ $booking : Person \leftrightarrow SeatType$ $capacity : \mathbb{N}$ $count : \mathbb{N}$
$booked = dom\ booking$ $capacity > 0$ $count \geq 0$
<i>INIT</i>
$booked = \emptyset$ $capacity = 100$ $count = 0$
<i>BookOK</i>
$\Delta(booking, count)$ $customer? : Person$ $seat? : SeatType$
$customer? \notin booked$ $seat? \notin ran\ SeatType$ $count < capacity$ $booking' = booking \cup \{customer? \mapsto seat?\}$ $count' = count + 1$
<i>CancelOK</i>
$\Delta(booking, count)$ $customer? : Person$
$customer? \in booked$ $count > 0$ $booking' = \{customer?\} \triangleleft booking$ $count' = count - 1$
...

*Booking/cont.*

...

*Success*

$response! : Message$

$response! = 'ok'$

*CustomerExists*

$customer? : Person$

$response! : Message$

$customer? \in booked$

$response! = 'Customer\ already\ exists'$

*CustomerUnknown*

$customer? : Person$

$response! : Message$

$customer? \notin booked$

$response! = 'Customer\ unknown'$

*SeatTaken*

$seat? : SeatType$

$response! : Message$

$seat? \in ran\ booking$

$response! = 'Seat\ taken'$

*TheaterFull*

$response! : Message$

$count = capacity$

$response! = 'Theater\ full'$

*TheaterEmpty*

$response! : Message$

$count = 0$

$response! = 'Theater\ empty'$

$Book \triangleq (BookOK \wedge Success) \oplus CustomerExists \oplus (SeatTaken \vee TheaterFull)$

$Cancel \triangleq (CancelOK \wedge Success) \oplus (CustomerUnknown \vee TheaterEmpty)$

<i>Booking2</i>	
$\uparrow (Book, Cancel, GetNumberOfCustomers)$	
<i>Booking</i>	
<i>GetNumberOfCustomers</i>	
$result! : \mathbb{N}$	
$result! = count$	
<i>ModifyBookingOK</i>	
$\Delta(booking)$	
$customer? : Person$	
$seat? : SeatType$	
$customer? \in booked$	
$seat? \notin ran\ booking$	
$booking' = booking \oplus \{customer? \mapsto seat?\}$	
$ModifyBooking \hat{=} (ModifyBookingOK \wedge Success) \oplus (CustomerUnknown \vee SeatTaken)$	