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.M

   AIRPORTS

   CL-USER 2 > (print *airports*)

   ((YUL.MONTREAL) (LCY.LONDON_UK) (LHR.LONDON_UK) (MIL.MILAN) (SFO.SAN_FRANCISCO) (SD
   ((YUL.MONTREAL) (LCY.LONDON_UK) (LHR.LONDON_UK) (MIL.MILAN) (SFO.SAN_FRANCISCO) (SD

   CL-USER 3 > *airports*
   ((YUL.MONTREAL) (LCY.LONDON_UK) (LHR.LONDON_UK) (MIL.MILAN) (SFO.SAN_FRANCISCO) (SD

   CL-USER 4 >

2. $monitored$=dom($airports$)

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

4. This should be an unordered structe, 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 $\langle$AIRPORT, CITY$\rangle$ could also be used to replace the tuples.

5.

$$
\begin{array}{l}
\underline{\quad AirportSystem \quad}\\
codes \ : \ \mathbb{P} \ AIRPORT \\
airports : AIRPORT \rightarrow CITY \qquad \texttt{--non-injective, surjective} \\
cities : \ \mathbb{P} \ CITY \\
\hline
codes = \mathrm{dom} \ airports \\
deployed = \mathrm{ran} \ airports \\
\end{array}
$$

6.

$$
\begin{array}{l}
\underline{\quad Success \quad}\\
\Xi AirportSystem \\
response! : MESSAGE \\
\hline
response! \ = \ 'ok' \\
\end{array}
$$

$$
\begin{array}{l}
\underline{\quad CodeAlreadyInUse \quad}\\
\Xi AirportSystem \\
code? : AIRPORT \\
response! : Message \\
\hline
code? \in \mathrm{dom} \ airports \\
response! \ = \ 'Code \ already \ in \ use' \\
\end{array}
$$

$$
\begin{array}{l}
\underline{\quad AddAirportOK \quad}\\
\Delta AirportSystem \\
code? : AIRPORT \\
city? : CITY \\
\hline
code? \notin \mathrm{dom} \ airports \\
airports' = airports \cup \{code? \mapsto city?\} \\
\end{array}
$$

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

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.

$$
\begin{array}{l}
\underline{\quad\textit{UpdateAirportOK}\quad\underline{\qquad\qquad\qquad\qquad\qquad}} \\
\Delta\,AirportSystem \\
code? : AIRPORT \\
city? : CITY \\
\hline
code? \in \mathrm{dom}\ airports \\
airports' = airports \oplus \{code? \mapsto city?\}
\end{array}
$$

4

```
   ┌─ CodeNotInUse ────────────────────────
   │ ΞAirportSystem
   │ code? : AIRPORT
   │ response! : Message
   ├────────────────────────────────────────
   │ code? ∉ dom airports
   │ response! = 'Code not in use'
   └────────────────────────────────────────
```

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

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.

$$
\begin{array}{l}
\rule{3cm}{0.4pt}\ DeleteAirportOK\ \rule{5cm}{0.4pt} \\
\Delta AirportSystem \\
code? : AIRPORT \\
\hline
code? \in \mathrm{dom}\ airports \\
airports' = \{code?\} \lhd airports
\end{array}
$$

```
 __ CodeNotInUse _____
| ΞAirportSystem
| code? : AIRPORT
| response! : Message
|_____
| code? ∉ dom airports
| response! = 'Code not in use'
|_____
```

$$DeleteAirport \;\widehat{=}\;\; (DeleteAirportOK \wedge Success) \oplus CodeNotInUse$$

12.
```
CL-USER 1 > (defvar *airports*
   '((YUL . Montreal)
     (LCY . London_UK)
     (LHR . London_UL)
     (MIL . Milan)
     (SFO . San_Francisco)
     (SDQ . Santo_Domingo)))
 *AIRPORTS*


CL-USER 2 > (defun deleteAirport (code)
  (let ((removed (remove-if (lambda (airport)
                              (string= (car airport) code))
                            *airports*)))
    (if (equal removed *airports*)
        (print "Code not in use")
        (progn
          (setf *airports* removed)
          (print "ok"))))
  *airports*)
 DELETEAIRPORT


CL-USER 3 > (deleteAirport 'YUL)
```

```
"ok"

((LCY . LONDON_UK) (LHR . LONDON_UL) (MIL . MILAN) (SFO . SAN_FRANCISCO) (SDQ . SAN

CL-USER 4 >
```

13.

```
┌─ GetAllAirportsOK ──────────────────────────────────
│ ΞAirportSystem
│ city? : CITY
│ airports! : ℙ AIPORT
├──────────────────────────────────────────────────────
│ city? ∈ ran airports
│ airports! = airports⁻¹ city?
└──────────────────────────────────────────────────────
```

$$\begin{array}{l} GetAllAirportsOK \\ \Xi AirportSystem \\ city? : CITY \\ airports! : \mathbb{P}\, AIPORT \\ \hline city? \in \mathrm{ran}\ airports \\ airports! = airports^{-1} city? \end{array}$$

```
┌─ CityNotFound ──────────────────────────────────────
│ ΞAirportSystem
│ city? : CITY
│ response! : MESSAGE
├──────────────────────────────────────────────────────
│ city? ∉ ran airports
│ response! = "Citynotfound"
└──────────────────────────────────────────────────────
```

$$\begin{array}{l} CityNotFound \\ \Xi AirportSystem \\ city? : CITY \\ response! : MESSAGE \\ \hline city? \notin \mathrm{ran}\ airports \\ response! = "Citynotfound" \end{array}$$

$$GetAllAirports \;\hat{=}\;\; (GetAllAirportsOK \wedge Success) \oplus CityNotFound$$

14. CL-USER 1 > (defvar *airports*

```
    '((YUL . Montreal)

      (LCY . London_UK)

      (LHR . London_UK)

      (MIL . Milan)

      (SFO . San_Francisco)

      (SDQ . Santo_Domingo)))

  *AIRPORTS*
```

8

```
CL-USER 2 > (defun getAllAirports (city)
(mapcar #'car (remove-if-not (lambda (airport)
(string= (cdr airport) city))
                               *airports*)))
GETALLAIRPORTS


CL-USER 3 > (getAllAirports 'LondonUK)
NIL


CL-USER 4 > (getAllAirports 'London_UK)
(LCY LHR)


CL-USER 5 >(defun getAllAirports (city)
  (let ((airports (mapcar #'car (remove-if-not (lambda (airport)
                                      (string= (cdr airport) city))
                                 *airports*))))
    (if (null airports)
        (print "City not found")
        (progn
          (print "ok")
          airports))))
GETALLAIRPORTS


CL-USER 6 > (getAllAirports 'London_UK)


"ok"
(LCY LHR)


CL-USER 7 > (getAllAirports 'Jamaica)
```

9

```
"City not found"
"City not found"


CL-USER 8 >
```

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:

```
┌─ TempMonitor ──────────────────────────────────────────────
│ deployed' : ℙ SENSOR_TYPE
│ map : SENSOR_TYPE ⇸ LOCATION_TYPE       --partial bijective
│ read : SENSOR_TYPE ⇸ TEMPERATURE_TYPE
├────────────────────────────────────────────────────────────
│ deployed = dom map
│ deployed = dom read
└────────────────────────────────────────────────────────────
```

```
┌─ DeploySensorOK ───────────────────────────────────────────
│ ΔTempMonitor
│ sensor? : SENSOR_TYPE
│ location? : LOCATION_TYPE
│ temperature? : TEMPERATURE_TYPE
├────────────────────────────────────────────────────────────
│ sensor? ∉ deployed
│ location? ∉ ran map
│ deployed' = deployed ∪ {sensor?}
│ map' = map ∪ {sensor? ↦ location?}
│ read' = read ∪ {sensor? ↦ temperature?}
└────────────────────────────────────────────────────────────
```

```
┌─ ReadTemperatureOK ────────────────────────────────────────
│ ΞTempMonitor
│ location? : LOCATION_TYPE
│ temperature! : TEMPERATURE_TYPE
├────────────────────────────────────────────────────────────
│ location? ∈ ran map
│ temperature! = read(map⁻¹(location?))
└────────────────────────────────────────────────────────────
```

$temperature! = read(map^{-1}(location?))$

```
┌─ Success ──────────────────────────────────────────────────
│ ΞTempMonitor
│ response! : MESSAGE
├────────────────────────────────────────────────────────────
│ response! = 'ok'
└────────────────────────────────────────────────────────────
```

Ξ *TempMonitor*

```
┌─ SensorAlreadyDeployed ─────────────────────────────────────────
│ Ξ TempMonitor
│ sensor? : SENSOR_TYPE
│ response! : Message
├─────────────────
│ sensor? ∈ deployed
│ response! = ′Sensor deployed′
└─────────────────────────────────────────────────────────────────
```

```
┌─ LocationAlreadyCovered ────────────────────────────────────────
│ Ξ TempMonitor
│ location? : LOCATION_TYPE
│ response! : Message
├─────────────────
│ location? ∈ ran map
│ response! = ′Location already covered′
└─────────────────────────────────────────────────────────────────
```

```
┌─ LocationUnknown ───────────────────────────────────────────────
│ Ξ TempMonitor
│ location? : LOCATION_TYPE
│ response! : Message
├─────────────────
│ location? ∉ ran map
│ response! = ′Location not covered′
└─────────────────────────────────────────────────────────────────
```

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

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

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:

Booking
$\upharpoonright (Book, Cancel)$

      $booked : \mathbb{P}\ Person$
      $booking : Person \rightarrowtail SeatType$
      $capacity : \mathbb{N}$
      $count : \mathbb{N}$

      $booked =\ dom\ booking$
      $capacity > 0$
      $count \geq 0$

    I_{NIT}
      $booked = \varnothing$
      $capacity = 100$
      $count = 0$

    BookOK
      $\Delta(booking, count)$
      $customer? : Person$
      $seat? : SeatType$

      $customer? \notin booked$
      $seat? \notin \mathrm{ran}\ SeatType$
      $count < capacity$
      $booking' = booking \cup \{customer? \mapsto seat?\}$
      $count' = count + 1$

    CancelOK
      $\Delta(booking, count)$
      $customer? : Person$

      $customer? \in booked$
      $count > 0$
      $booking' = \{customer?\} \lhd booking$
      $count' = count - 1$

  ...

```
Booking/cont.
...

  Success
    response! : Message

    response! = 'ok'


  CustomerExists
    customer? : Person
    response! : Message

    customer? ∈ booked
    response! = 'Customer already exists'


  CustomerUnknown
    customer? : Person
    response! : Message

    customer? ∉ booked
    response! = 'Customer unknown'


  SeatTaken
    seat? : SeatType
    response! : Message

    seat? ∈ ran booking
    response! = 'Seat taken'

  TheaterFull
    response! : Message

    count = capacity
    response! = 'Theater full'

  TheaterEmpty
    response! : Message

    count = 0
    response! = 'Theater empty'


  Book ≙ (BookOK ∧ Success) ⊕ CustomerExists ⊕ (SeatTaken ∨ TheaterFull)

  Cancel ≙ (CancelOK ∧ Success) ⊕ (CustomerUnknown ∨ TheaterEmpty)
```

$\underline{\quad Booking2 \quad}$

$\upharpoonright (Book,\ Cancel,\ GetNumberOfCustomers)$

$Booking$

$\underline{\quad GetNumberOfCustomers \quad}$
$result! : \mathbb{N}$

$result! = count$

$\underline{\quad ModifyBookingOK \quad}$
$\Delta(booking)$
$customer? : Person$
$seat? : SeatType$

$customer? \in booked$
$seat? \notin ran\ booking$
$booking' = booking \oplus \{customer? \mapsto seat?\}$

$ModifyBooking \mathrel{\widehat{=}} (ModifyBookingOK \wedge Success) \oplus (CustomerUnknown \vee SeatTaken)$