Concordia University

Department of Computer Science and Software Engineering

SOEN 331: Formal Methods for Software Engineering

Exercises in Z and Object-Z

Nathan Grenier, Nirav Patel

March 27, 2024

PROBLEM 1 (40 pts)

Consider a system that handles railway connections as it associates (source) cities to all their corresponding destinations. The requirements of the system are as follows:

1. A source city may have connections to possibly multiple destination cities.
2. Multiple source cities may share the same group of destination cities.

We introduce the type $CITY$. A possible state of the system is shown below as it is captured by variable *connections*:

$$
\begin{aligned}
&\mathit{connections} = \\
&\qquad \{ \\
&\qquad\ \ \mathit{Montreal} \mapsto \{\mathit{Ottawa}, \mathit{Kingston}, \mathit{Quebec}, \mathit{Halifax}\}, \\
&\qquad\ \ \mathit{Ottawa} \mapsto \{\mathit{Montreal}, \mathit{Toronto}\}, \\
&\qquad\ \ \mathit{Toronto} \mapsto \{\mathit{Montreal}, \mathit{Ottawa}\}, \\
&\qquad\ \ \mathit{Halifax} \mapsto \{\mathit{Montreal}, \mathit{Quebec}\}, \\
&\qquad\ \ \mathit{Quebec} \mapsto \{\mathit{Montreal}, \mathit{Halifax}\}, \\
&\qquad\ \ \mathit{Kingston} \mapsto \{\mathit{Montreal}\} \\
&\qquad \}
\end{aligned}
$$

1. (2 pts) Is *connections* a binary relation? Explain why, and if so express this formally.

   Solution:

   Yes, the variable *connections* represents a binary relation. A binary relation is a set of ordered pairs where each pair consists of two elements. In this case, the pairs in *connections* are formed by associating a source city with its corresponding set of destination citites.

Formally, a binary relation R from set A to set B is defined as a subset of the cartesian product $A \times B$. In this context, the set of connections can be expressed formally as:

$$connections \subseteq \{(source, destination) \mid source \in CITY, destination \in CITY\}$$

Therefore, the variable *connections* conforms to the definition of a binary relation as it relates source cities to their respective destination cities in pairs.

2. (2 pts) In the expression $connections \in (...)$, what would RHS be?

   The right-hand side would be the set of all possible binary relations on the set of cities, denoted as 'CITY'.

   Formally, the right-hand side can be represented as:

   $$(CITY \times \mathcal{P}(CITY))$$

   Where:

   - 'CITY' is the set of all cities
   - '$\mathcal{P}(CITY)$' is the power set of 'CITY', which represents the set of all possible subsets of 'CITY'.
   - 'CITY$\times\mathcal{P}$(CITY)' is the Cartesian product of 'CITY' and '$\mathcal{P}$(CITY)', which represents the set of all ordered pairs '(*source*, *destinations*)' where '*source*' is a city and '*destinations*' is a set of cities.

   Therefore, the expression '$connections \in (CITY, \mathcal{P}(CITY))$' means that the *connections* variable is an element of the set of all binary relations on the set of cities.

3. (2 pts) Is *connections* a function? If so, define the function formally, and reason about the properties of injectivity, surjectivity, and bijectivity.

   To determine if *connections* is a function, we need to ensure that each source city maps to exactly one set of destination cities.

We see that each source city (e.g., Montreal, Ottawa, Toronto, Halifax, Quebec, Kingston) maps to exactly one set of destination cities, satisfying the definition of a function.

Now, let's define the function formally:

Let f be the function representing the railway connections, where:

- Domain ($\texttt{Dom}(f)$): Set of source cities.
- Codomain ($\texttt{Cod}(f)$): Set of sets of destinations cities.
- $f(x)$: Set of destination cities connected to the source city x.

Formally:

$$f(x) = \text{Set of destination cities connected to source city } x$$

We then analyze whether the function is injective, surjective, and bijective:

(a) Injectivity: The function is injective if each source city maps to a unique set of destination cities. In this case, it's not injective because multiple source cities may share the same group of destination cities.

(b) Surjectivity: The function is surjective if every set of destination cities is mapped to by some source city. Since there can be multiple source cities mapping to the same group of destination cities, it is surjective.

(c) Bijectivity: The function is bijective if it is both injective and surjective. Since it's not injective but surjective, it's not bijective.

In conclusion, the *connections* variable represents a function, but it is not injective or bijective, only surjective.

4. (2 pts) Describe the meaning and evaluate the following expression:

$\{Montreal, Halifax\} \lhd connections$

The symbol $\lhd$ denotes the domain restriction operator. When applied to a function or a relation, it restricts the domain of a function to a specified subset.

In the expression $\{Montreal, Halifax\} \lhd connections$, we are restricting the domain of the function represented by *connections* to the set $\{Montreal, Halifax\}$.

We are essentially interested in the subset of the connections that involve either Montreal or Halifax as the source city.

We evaluate the expression:

$$
\begin{aligned}
Montreal &\mapsto \{Ottawa, Kingston, Quebec, Halifax\}, \\
Halifax &\mapsto \{Montreal, Quebec\}
\end{aligned}
$$

So, after the domain restriction, we only have connections involving Montreal and Halifax as source cities.

In conclusion, the expression $\{Montreal, Halifax\} \lhd connections$ yields the connections from Montreal and Halifax to their respective destination cities.

5. (2 pts) Describe the meaning and evaluate the following expression:

   $connections \rhd \{\{Montreal, Halifax\}\}$

   The symbol $\rhd$ represents the range restriction operator. When applied to a function, it restricts the range of the function to a specified subset.

   In the expression $connections \rhd \{\{Montreal, Halifax\}\}$, we are restricting the range of the function represented by *connections* to the set containing the destinations Montreal and Halifax.

   We evaluate the expression:

$$
Quebec \mapsto \{Montreal, Halifax\}
$$

So, after the range restriction, we only have connections where the destination is either Montreal or Halifax.

In conclusion, the expression $connections \rhd \{\{Montreal, Halifax\}\}$ yields the connections from source cities that map to the destinations Montreal and Halifax.

6. (2 pts) Describe the meaning and evaluate the following expression:

   $\{Montreal, Quebec, Halifax\} \lhd\!\!\!- \ connections$

   The symbol $\lhd\!\!\!-$ represents the domain subtraction or domain anti-restriction operator. When applied to a function or a relation, it removes a specified subset from the domain of the function/relation.

   In the expression $\{Montreal, Quebec, Halifax\} \lhd\!\!\!- \ connections$, we are subtracting the set $\{Montreal, Quebec, Halifax\}$ from the domain of the function $connections$.

   We evaluate the expression:

$$\begin{aligned} Ottawa &\mapsto \{Montreal, Toronto\}, \\ Toronto &\mapsto \{Montreal, Ottawa\}, \\ Kingston &\mapsto \{Montreal\} \end{aligned}$$

   So, after the domain subtraction, we only have connections originating from Ottawa, Toronto, and Kingston.

   In conclusion, the expression $\{Montreal, Quebec, Halifax\} \lhd\!\!\!- \ connections$ yields the connections excluding those originating from Montreal, Quebec, and Halifax.

7. (2 pts) Describe the meaning and evaluate the following expression:

   $connections \rhd\!\!\!- \{\{Ottawa, Kingston, Quebec, Halifax\}, \{Montreal, Ottawa\}, \{Montreal\}\}$
   The symbol $\rhd\!\!\!-$ represents the range subtraction operator. When applied to a function, it removes a specified subset from the range of the function.

In the expression $connections \rhd \{\{Ottawa, Kingston, Quebec, Halifax\}, \{Montreal, Ottawa\}, \{Montreal$ we are subtracting the set containing various sets of destinations from the range of the function represented by $connections$.

We evaluate the expression:

$$\begin{aligned} Ottawa &\mapsto \{Montreal, Toronto\}, \\ Halifax &\mapsto \{Montreal, Quebec\}, \\ Quebec &\mapsto \{Montreal, Halifax\} \end{aligned}$$

So, after the range subtraction, we only have connections starting from Ottawa, Halifax and Quebec.

In conclusion, the expression $connections \rhd \{\{Ottawa, Kingston, Quebec, Halifax\}, \{Montreal, Ottawa\}$ yields the connections excluding those leading to the sets defined in the expression.

8. (2 pts) Describe the meaning and evaluate the following expression that forms a post-condition to some operation:

$$\begin{aligned} connections' = connections \oplus \{ \\ & Halifax \mapsto \{Montreal, Charlottetown, Quebec\}, \\ & Charlottetown \mapsto \{Halifax\} \\ \} \end{aligned}$$

The expression given represents a post-condition for an operation involving the variable $connections$. The operator being used here is the relational overriding operated denoted by $\oplus$. This operator combines two relations, where the right-hand side relation takes precedence over the left-hand side in case of conflicting keys.

After evaluating this expression with the overriding operator:

(a) The existing entry for Halifax in *connections* will be replaced with the new set of destinations: $\{Montreal, Charlottetown, Quebec\}$.

(b) A new entry for Charlottetown with destinations $\{Halifax\}$ will be added to *connections*.

The resulting updated *connections* will be:

$$
\begin{aligned}
&connections = \\
&\qquad \{ \\
&\qquad\quad Montreal \mapsto \{Ottawa, Kingston, Quebec, Halifax\}, \\
&\qquad\quad Ottawa \mapsto \{Montreal, Toronto\}, \\
&\qquad\quad Toronto \mapsto \{Montreal, Ottawa\}, \\
&\qquad\quad Halifax \mapsto \{Montreal, Charlottetown, Quebec\}, \\
&\qquad\quad Quebec \mapsto \{Montreal, Halifax\}, \\
&\qquad\quad Kingston \mapsto \{Montreal\} \\
&\qquad\quad Charlottetown \mapsto \{Halifax\} \\
&\qquad \}
\end{aligned}
$$

9. (6 pts) Assume that we need to add a new entry into the database table represented by *connections*. We have decided <u>not</u> to deploy a precondition. What could be the consequences to the system if we deployed a) **set union** and b) **relational overriding**?

(a) Set Union: If set union is deployed without a pre-condition when adding a new entry to the *connections* database table, the consequences could include:

- **Duplication of Data**: Without a precondition, set union may lead to duplicate entries being added to the database. This can result in inefficiencies and potential data inconsistencies.
- **Increased Complexity**: The set union operation may add unnecessary complexity to the system, especially if it allows for redundant or conflicting data entries.
- **Potential Data Integrity Issues**: Inconsistent data entries may arise if set union is used without proper checks, leading to data integrity issues within the system.

(b) Relational Overriding:

- **Loss of Existing Data**: Relational overriding will replace existing data without considering any conditions. This can lead to the loss of valuable information if not handled carefullly.
- **Potential Data Loss**: If relational overriding is applied without caution, it can overwrite critical information unintentionally, resulting in data loss and inconsistencies.
- **Inconsistencies in Data**: Without a precondition, relational overriding may introduce inconsistencies in the database by replacing existing connections with new ones without proper validation.

In both cases, deploying these operations without preconditions can introduce risks to the system's data integrity, consistency, and overall functionality.

10. (2 pts) Consider operation *AddOperation* to **add a new entry** to the table, defined by the following pair of assertions.

$$city? \notin \texttt{dom}\ connections$$

$$connections' = connections \cup \{city? \mapsto destinations?\}$$

What would be the result of the call `AddConnection(Montreal, (Boston, NYC))`, and in the case of failure, whom should we blame and why?

Given the call `AddConnection(Montreal, (Boston, NYC))`, let's evaluate it step by step:

(a) Check if Montreal is not already in the domain of connections. Since Montreal is present in the domain, the first assertion fails.

(b) Since the first assertion fails, the operation cannot proceed to update the connections. Therefore, the call to `AddConnection(Montreal, (Boston, NYC))` fails.

If this operation fails, the blame would typically fall on the caller of the function, who invoked the `AddConnection` function with a city that already exists in the domain of connections. It suggests that the caller did not properly check whether the city being added already exists in the system, leading to a violation of the precondition specified by the first assertion. Therefore, the caller should be responsible for handling such cases or ensuring that only valid data is passed to the `AddConnection` function.

11. (2 pts) Consider the following modification to the postcondition of *AddOperation*:

$$connections' = connections \oplus \{city? \mapsto destinations?\}$$

What would be the result of the call `AddConnection(Kingston, (Boston,NYC)`? In the absence of a precondition, can relational overriding unintentionally capture the intent of the operation?

The modified postcondition for the *AddOperation* utilizes relational overriding ($\oplus$) to update the connections. It specifies that the new entry $city? \mapsto destinations?$) should be added to the existing connections, overriding any existing mapping for the same city if it exists.

Now, let's consider the call `AddConnection(Kingston, (Boston, NYC))`:

(a) Since Kingston is not in the domain of connections, the operation can proceed without any conflict.

(b) The postcondition specifies that the connections should be updated using relational overriding, meaning that if Kingston already exists in the connections, its destinations will be replaced with the new destinations (Boston, NYC). If Kingston does not exist in the connections, a new entry will be added for it.

In this case, since Kingston is not already in the connections, the postcondition simply adds a new entry for Kingston mapped to the destinations (Boston, NYC).

In the absence of a precondition, relational overriding can unintentionally capture the intent of the operation if there are conflicting entries. For example, if Kingston already existed in the connections with different destinations, the use of relational overriding would replace those destinations with the new ones (Boston, NYC), potentially overriding the existing connections unintentionally.

Therefore, careful consideration should be given to the potential consequences of using relational overriding without a precondition to ensure that it aligns with the intended behavior of the operation.

12. (3 pts) Consider the following state schema in the Z Specification Language:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RailwayManagement}} \\
\hline
\textit{cities} : \mathcal{P}\,\textit{CITY} \\
\textit{connections} : \textit{CITY} \nrightarrow \mathcal{P}\,\textit{CITY} \\
\hline
\textit{cities} = \mathtt{dom}\ \textit{connections} \\
\hline
\end{array}
$$

Define the schema for operation *GetDestinations* which returns all destinations for one given city.

To define the schema for the operation *GetDestinationsGetDestinations*, which returns all destinations for one given city, we can use the Z notation. The schema will take a city as input and return its corresponding set of destinations. Here's how we can define it:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GetDestinations}} \\
\hline
\Delta \textit{RailwayManagement} \\
\textit{city}? : \textit{CITY} \\
\textit{destinations}! : \mathcal{P}\,\textit{CITY} \\
\hline
\textit{city}? \in \textit{cities} \\
\textit{destinations}! = \textit{connections}(\textit{city}?) \\
\hline
\end{array}$$

In this schema:

- *GetDestinationsGetDestinations* represents the operation schema.
- △ RailwayManagement denotes that this operation may update the state of the system represented RailwayManagement.
- *city*? is an input variable representing the city for which destinations are requested
- *destinations*! is an output variable representing the set of destinations corresponding to the input city.
- The where clause ensures that the input city (*city*?) is a valid city in the system (i.e., it exists in the set cities).
- The assignment $\textit{destinations}! = \textit{connections}(\textit{city}?)$ retrieves the set of destinations for the given city from the connections mapping.

This schema ensures that the operation *GetDestinations* operates correctly within the context of the *RailwayManagement* state schema, returning the set of destinations for a given city.

13. (1 pt) (**PROGRAMMING**) Define global variables *connections* in Common LISP and populate it with some sample data. Demonstrate that the variable indeed contains the ordered pairs as shown above.

```
(defvar *connections* '())

(defun add-connection (source destinations)
  (push (cons source destinations) *connections*))

```

```
(defun initialize-connections ()
  (setq *connections*
        '((Montreal . (Ottawa Kingston Quebec Halifax))
          (Ottawa . (Montreal Toronto))
          (Toronto . (Montreal Ottawa))
          (Halifax . (Montreal Quebec))
          (Quebec . (Montreal Halifax))
          (Kingston . (Montreal)))))

;; Initialize connections
(initialize-connections)

;; Function to display connections with arrows
(defun display-connections-with-arrows ()
  (format t "Connections:~%")
  (dolist (connection *connections*)
    (let ((source (car connection))
          (destinations (cdr connection)))
      (format t "~a -> " source)
      (dolist (destination destinations)
        (format t "~a, " destination))
      (terpri))))

;; Display the contents of connections with arrows
(display-connections-with-arrows)
```

We explain a basic rundown of the code:

(a) The code begins by initializing the global variable *connections* to an empty list using defvar. This variable will store the railway connections data.

(b) The add-connection function is defined to add a new connection to the *connections* variable. It takes two arguments: source, representing the source city, and destinations, representing a list of destination cities. It uses push to add a new cons cell representing the connection to the *connections* list.

(c) The initialize-connections function is defined to populate the *connections* variable with sample data. It sets *connections* to a list of cons cells, each representing a connection from a source city to its corresponding destinations.

(d) The display-connections-with-arrows function is defined to visually display the connections stored in the *connections* variable. It iterates over each connection in *connections*, extracting the source city and its destinations, and prints them with arrows separating the source city from its destinations.

(e) Finally, the code calls the initialize-connections function to populate the *connections* variable with sample data and then calls the display-connections-with-arrows function to display the contents of *connections* in a visually formatted way with arrows.

```
Absolute running time: 0.16 sec, cpu time: 0.02 sec, memory peak: 9 Mb, absolute service time: 0,28 sec


Connections:
MONTREAL -> OTTAWA, KINGSTON, QUEBEC, HALIFAX,
OTTAWA -> MONTREAL, TORONTO,
TORONTO -> MONTREAL, OTTAWA,
HALIFAX -> MONTREAL, QUEBEC,
QUEBEC -> MONTREAL, HALIFAX,
KINGSTON -> MONTREAL,
```

Figure 1: Environment Interaction for Question 1 - Part 13

14. (3 pts) Describe how you would validate variable *connections* , i.e. how to show that it holds a function.

    To validate that the variable *connections* holds a function, we need to verify that it satisfies the properties of a mathematical function. In mathematical terms, a function is a relation between a set of inputs (the domain) and a set of possible outputs (the codomain), such that each input is related to exactly one output.

    Here's how we can validate variable *connections* to ensure it holds a function:

    (a) **Domain Check**: Ensure that each source city in the domain of connections is associated with at least one destination city. This ensures that every input (source city) has a corresponding output (destination city).

(b) **Unique Outputs**: Ensure that each source city is associated with a unique set of destination cities. In other words, there are no duplicate entries for the same source city. This ensures that each input is related to exactly one output.

(c) **Consistency**: Ensure that the relation is consistent, meaning that if a source city s is associated with a destination city dd, then dd must be present in the domain of *connections* with s as one of its destinations. This ensures that the relation does not contain contradictory information.

(d) **No Extra Entries**: Ensure that there are no extra entries in *connections* that do not conform to the function's definition. This means ensuring that every key-value pair in *connections* adheres to the function's mapping rules.

15. (3 pts) (**PROGRAMMING**) Define a predicate function, `isfunctionp`, in Common Lisp that reads a variable like *connections* and indicates if the variable corresponds to a function or not.

```lisp
(defun isfunctionp (variable)
  "Check if the given variable corresponds to a function."
  (and (listp variable)                   ; Check if variable is a list
       (every #'consp variable)          ; Check if all elements are cons cells
       (every (lambda (pair)             ; Check if each cons cell is a valid key-value pair
                (and (consp pair)
                     (symbolp (car pair))
                     (listp (cdr pair))))
              variable)))

;; Example 1: Variable corresponds to a function
(setq *connections-function*
      '((Montreal . (Ottawa Kingston Quebec Halifax))
        (Ottawa . (Montreal Toronto))
        (Toronto . (Montreal Ottawa))
        (Halifax . (Montreal Quebec))
```

```lisp
        (Quebec . (Montreal Halifax))
        (Kingston . (Montreal))))

;; Test the function for *connections-function*
(if (isfunctionp *connections-function*)
    (format t "The variable *connections-function* corresponds to a function.~%")
    (format t "The variable *connections-function* does not correspond to a function.~%"))

;; Example 2: Variable does not correspond to a function
(setq *connections-nonfunction* '(Montreal Ottawa Toronto))

;; Test the function for *connections-nonfunction*
(if (isfunctionp *connections-nonfunction*)
    (format t "The variable *connections-nonfunction* corresponds to a function.~%")
    (format t "The variable *connections-nonfunction* does not correspond to a function.~%"))
```

We describe a basic run-down of the code:

(a) The `isFunction` function checks whether a given variable corresponds to a function.

(b) It checks if the variable is a list (`listp`), ensuring it has the structure of a function.

(c) It verifies if every element of the list is a cons cell (`consp`), ensuring each element is a key-value pair.

(d) It checks if each cons cell consists of a symbol as the key and a list as the value, as expected for a function.

(e) Example 1 demonstrates a variable ($*connections - function*$) that corresponds to a function, with a list of key-value pairs.

(f) Example 2 demonstrates a variable ($*connections - nonfunction*$) that does not correspond to a function, being just a list of symbols without the key-value pair structure.

```
Absolute running time: 0.16 sec, cpu time: 0.02 sec, memory peak: 9 Mb, absolute service time: 0,25 sec


The variable *connections-function* corresponds to a function.
The variable *connections-nonfunction* does not correspond to a function.
```

Figure 2: Environment Interaction for Question 1 - Part 15

16. (2 pts) (**PROGRAMMING**) Define function *getDestinations* in Common LISP.

```lisp
(defun getDestinations (city connections)
  "Retrieve the destinations for a given city from the connections."
  (cdr (assoc city connections)))

;; Example usage:
(defvar *connections*
  '((Montreal . (Ottawa Kingston Quebec Halifax))
    (Ottawa . (Montreal Toronto))
    (Toronto . (Montreal Ottawa))
    (Halifax . (Montreal Quebec))
    (Quebec . (Montreal Halifax))
    (Kingston . (Montreal))))

;; Get destinations for Montreal
(format t "Destinations for Montreal: ~a~%" (getDestinations '
    Montreal *connections*))

;; Get destinations for Ottawa
(format t "Destinations for Ottawa: ~a~%" (getDestinations 'Ottawa *
    connections*))
```

We explain a basic rundown of the code:

(a) The *getDestinations* function takes two arguments: *city*, the city for which destinations are to be retrieved, and *connections*, the list of connections.

(b) Inside the function, it uses the *assoc* function to search for the *city* in the *connections* list and retrieve the corresponding cons cell.

(c) Then, it uses *cdr* to extract the destinations from the cons cell.

(d) Finally, it returns the list of destinations for the given city.

(e) Example usage demonstrates how to use the *getDestinations* function with the $*connections*$ variable to retrieve destinations for Montreal and Ottawa.

```
Absolute running time: 0.16 sec, cpu time: 0.02 sec, memory peak: 9 Mb, absolute service time: 0,32 sec


Destinations for Montreal: (OTTAWA KINGSTON QUEBEC HALIFAX)
Destinations for Ottawa: (MONTREAL TORONTO)
```

Figure 3: Environment Interaction for Question 1 - Part 16

PROBLEM 2 (10 pts)

Consider the temperature monitoring system from our lecture notes. In this part, we will extract the specification with a new requirement.

Operation ReplaceSensor Sensors are physical devices, subjected to deterioration, or other type of damage. We need to provide the ability of the system to replace a sensor with another. For simplicity, let us not plan (”fix”, and) reuse them, but permenantly remove them from the system.

ReplaceSensorOK

$$
\begin{array}{l}
\Delta\, TempMonitor \\
oldSensor?, newSensor? : SENSOR\_TYPE \\
\hline
oldSensor? \in deployed \\
newSensor? \notin deployed \\
newSensor? \neq oldSensor? \\
newSensor? \notin \operatorname{dom} map \\
deployed' = (\{oldSensor?\} \lhd deployed) \cup \{newSensor?\} \\
map' = \{oldSensor?\} \lhd (map \oplus \{newSensor? \mapsto map(oldSensor?)\}) \\
read' = \{oldSensor?\} \lhd (read \oplus \{newSensor? \mapsto read(oldSensor?)\})
\end{array}
$$

This schema represents the successful replacement of an old sensor with a new sensor. It includes preconditions and postconditions for the operation.

Preconditions:

- $oldSensor?$ must be currently deployed ($oldSensor? \in deployed$).
- $newSensor?$ must not be currently deployed ($newSensor? \notin deployed$).
- $newSensor?$ must be different from oldSensor? ($newSensor? \neq oldSensor?$).
- The new sensor ($newSensor?$) must not be mapped to any location ($newSensor? \notin dommap$).

Postconditions:

- The old sensor ($oldSensor?$) is removed from the deployed set, and the new sensor ($newSensor?$) is added ($deployed' = (\{oldSensor?\} \lhd deployed) \cup \{newSensor?\}$).
- The mapping for the old sensor in the map relation is replaced by the mapping for the new sensor ($map' = \{oldSensor?\} \lhd (map \oplus \{newSensor? \mapsto map(oldSensor?)\})$).
- The temperature reading for the old sensor is replaced by the temperature reading for the new sensor in the read relation ($read' = \{oldSensor?\} \lhd (read \oplus \{newSensor? \mapsto read(oldSensor?)\})$).

SensorNotFound

$\Xi TempMonitor$
$sensor? : SENSOR\_TYPE$
$response! : MESSAGE$

$sensor? \notin deployed$
$response! = 'Sensor\ not\ found'$

This schema handles the case where the old sensor to be replaced is not found in the system.

$$
\begin{array}{|l}
\hline
\textit{SensorAlreadyMapped} \\
\Xi \textit{TempMonitor} \\
\textit{sensor}? : \textit{SENSOR\_TYPE} \\
\textit{response}! : \textit{MESSAGE} \\
\hline
\textit{newSensor}? \in \operatorname{dom} \textit{map} \\
\textit{response}! = \textit{'Sensor already mapped'} \\
\hline
\end{array}
$$

This schema handles the case where the new sensor is already mapped to a location in the system.

$$
\begin{array}{|l}
\hline
\textit{TargetAndNewSensorIdentical} \\
\Xi \textit{TempMonitor} \\
\textit{oldSensor}?, \textit{newSensor}? : \textit{SENSOR\_TYPE} \\
\textit{response}! : \textit{MESSAGE} \\
\hline
\textit{oldSensor}? = \textit{newSensor}? \\
\textit{response}! = \textit{'Old and new sensor are identical'} \\
\hline
\end{array}
$$

This schema handles the case where the old sensor and the new sensor are the same.

$$
\begin{array}{l}
\textit{ReplaceSensor} \mathrel{\widehat=} \\
\quad (\textit{ReplaceSensorOK} \land \textit{Success}) \\
\qquad \oplus\ \textit{SensorAlreadyDeployed} \\
\qquad \oplus\ \textit{SensorNotFound} \\
\qquad \oplus\ \textit{SensorAlreadyMapped} \\
\qquad \oplus\ \textit{TargetAndNewSensorIdentical}
\end{array}
$$

This Z operation combines the success schema *ReplaceSensorOK* with the error schemata *SensorAlreadyDeployed*, *SensorNotFound*, *SensorAlreadyMapped*, and *TargetAndNewSensorIdentical*. It provides a robust specification for the *ReplaceSensor* operation, ensuring proper handling of various error conditions and successful execution of the replacement process when all preconditions are met.