Department of Computer Science, CUI Lahore Campus

Formal Methods

By

Farooq Ahmad

Vienna development method (VDM)

Formal specification of software using VDM

Topics to be covered

- VDM specification language
- State specification in VDM
- Operations in VDM

Vienna Development Method (VDM)

- Vienna Development Method (VDM) is also a type of Formal Methods.
- It has well defined syntax.
- It has also tool support.
- It has two types:
  - VDM-SL (support imperative programming)
  - VDM++ (support object oriented programming)

The intrinsic types available in VDM-SL

- N : natural numbers (positive whole numbers)
- N_1 : natural numbers excluding zero
- Z : integers (positive and negative whole numbers)
- R : real numbers (positive and negative numbers that can include a fractional part)
- B : Boolean values (TRUE or FALSE)
- *Char* : the set of alphanumeric characters

The case study: requirements analysis

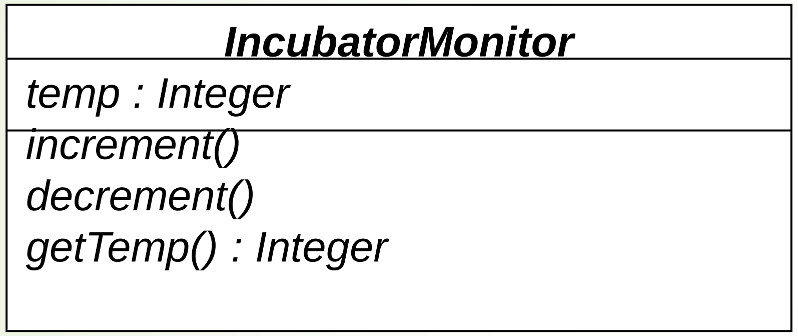
- The example we will use will be that of an incubator;
- The temperature of the incubator needs to be carefully controlled and monitored in order to provide the correct conditions for a particular biological experiment to be undertaken;
- We will specify the software needed to monitor and control the incubator temperature.
- In the first instance we are going to develop an extremely simple version of the system;
- In the initial version, *control* of the hardware lies outside of our system;

Contd.

- The hardware increments or decrements the temperature of the incubator in response to instructions (from someone or something outside of our system);
- Each time a change of one degree has been achieved, the software is informed of the change, which it duly records;
- However, safety requirements dictate that the temperature of the incubator must never be allowed to rise above 10 Celsius, nor fall below -10 Celsius.

The UML specification

We can identify a single class, *IncubatorMonitor*.



Operation increment(), decrement() don't give the output while operation *getTemp()* gives output of type integer.

We have identified one attribute and three methods;

- The single attribute records the temperature of the system and will be of type integer;
- The first two methods do not involve any input or output (since they merely record an increase or decrease of one degree);
- The final method reads the value of the temperature, and therefore will output an integer.

Specifying the state

- The first thing we will consider is known as the **state** of the system;
- In VDM-SL the state refers to the permanent data that must be stored by the system, and which can be accessed by means of operations;
- **It corresponds to the attributes in the class diagram;**
- The state is specified by declaring variables, in a very similar manner to the way that this is done in a programming language;
- The notation is not dissimilar from that used in the UML diagram;
- We specify one or more **variables**, giving each a name, and stating the *type* of data that the variable represents.

State of the System in VDM

- Syntax

state Name **of**

....

end

Specifying the state of the Incubator Monitor System

- The only data item that we need is the current temperature of the incubator; This will be of type integer;
- We shall call it *temp*. We specify the state as follows:

state *IncubatorMonitor* **of**

temp : Z

end

- In the above specification the variable *temp* (to hold the temperature) is an integer and is therefore declared to be of type Z.
- This is the only item of data to record in this case.

Declaring constants

- it is possible in VDM-SL to specify constants.
- It is done by using the keyword **values**, and the declaration would come immediately before the state definition.

values

$MAX : \mathbb{Z} = 10$

$MIN : \mathbb{Z} = -10$

Specifying functions explicitly

$$add \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$add(x, y) \;\underline{\Delta}\; x + y$$

- we can specify a function in VDM-SL **explicitly**.
- The first line is called the function **signature**.
- Its purpose is to state the input types that the function accepts (to the left of the arrow), together with the type of the output (to the right of the arrow).
- The second part is the **definition**, and describes the algorithm that is used for transforming the inputs to the output; this definition is placed on the right of the symbol, $\underline{\Delta}$ which is read ‘is defined as’.

Specifying invariants as explicit functions

$$\textbf{inv mk-}\mathit{IncubatorMonitor}(t) \;\underline{\Delta}\; \mathit{MIN} \leq t \leq \mathit{MAX}$$

- The invariant definition uses the keyword **inv**.
- **inv**: *State* → B
- The function maps a value of the state onto a Boolean – either TRUE or FALSE; and by specifying such a function we are saying that the state variables must be such that the result of the function is TRUE.
- After the keyword **inv**, we have the expression **mk**-*IncubatorMonitor*(*t*), which effectively is the input to the *inv* function.

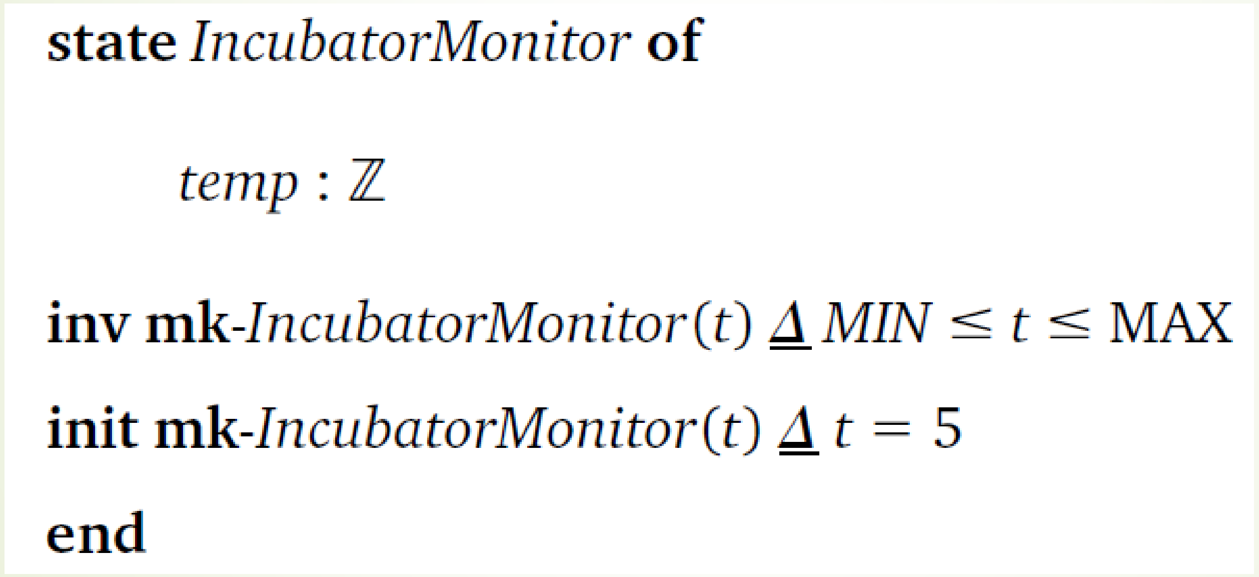
Specifying an Initialization Function

$$\textbf{init mk-}\mathit{IncubatorMonitor}(t) \ \underline{\Delta} \ t = 5$$

This is similar in style to the **invariant function**, and has the same signature;
It is very important to note that the initialization function – as with the operations –must preserve the invariant.

Once again, in our example, we are able to argue that since this function sets the temperature to 5 degrees, which is within the constraints allowed, the invariant is not violated.

State of the incubator monitoring system



state *IncubatorMonitor* **of**

$temp : \mathbb{Z}$

inv mk-$IncubatorMonitor(t) \ \underline{\Delta} \ MIN \leq t \leq \mathrm{MAX}$

init mk-$IncubatorMonitor(t) \ \underline{\Delta} \ t = 5$

end

Specifying the operations

- We need to specify a number of **operations** that the system should be able perform and by which means the data (that is the *state*) can accessed;
- <u>In VDM operations by definition access the state in some way, either by reading or writing the data, or both.</u>
- **There are three operations that we need to consider:**
  - An operation that records an increment in the temperature;
  - An operation that records a decrement in the temperature;
  - An operation that reads the value of the temperature.

Operations in VDM SL

- In VDM-SL an operation consists of four sections:
  - The operation header;
  - The **external** clause;
  - The **precondition**;
  - The **postcondition**.

The external clause

- Introduced by the VDM keyword **ext**;
- Keywords are written in lower case;
- In most texts they are bold and non-italic, whereas variable and type names are plain but italicized;
- The purpose of the external clause is to restrict the access of the operation to only those components of the state that are specified, and **to specify the mode of access,** either **read-only** (indicated by the keyword **rd**) or **read-write** (indicated by the keyword **wr**);
- In this example, there is only one component to the state (*temp*) and in this operation it is necessary to have read-write access to that component, since the operation needs actually to change the temperature.

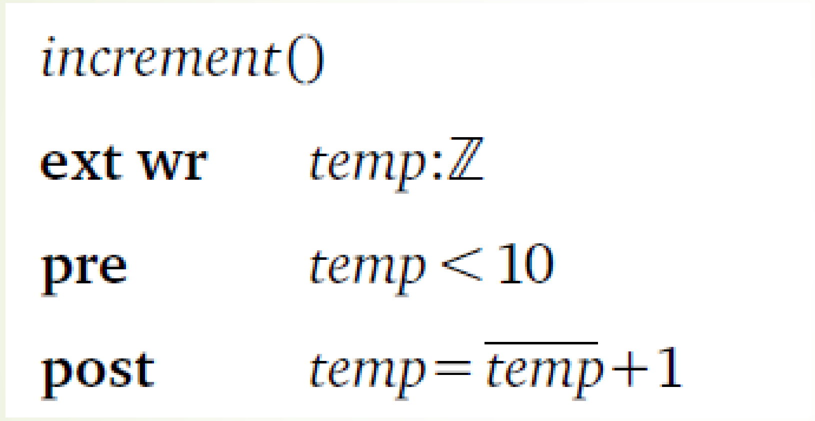
The pre-condition

- The purpose of the precondition is to place any necessary constraints on an operation.
- Pre-condition is introduced by the VDM keyword **pre**.
- In our incubator system, for example, we know that the temperature must be allowed to vary only within the range -10 to 10 degrees.
- If we did not specify a precondition here, we would be allowing the system to record a temperature that was outside of the allowed range – we would be allowing abnormal behavior of the system.

The post-condition

- Introduced by the keyword **post**;
- States the conditions that must be met after the operation has been performed;
- It is a predicate, containing one or more variables, the values of which must be such as to make the whole statement true;
- The only state variables that can be included in the postcondition are those that are referred to in the **ext** clause.

Operation to increase the temperature

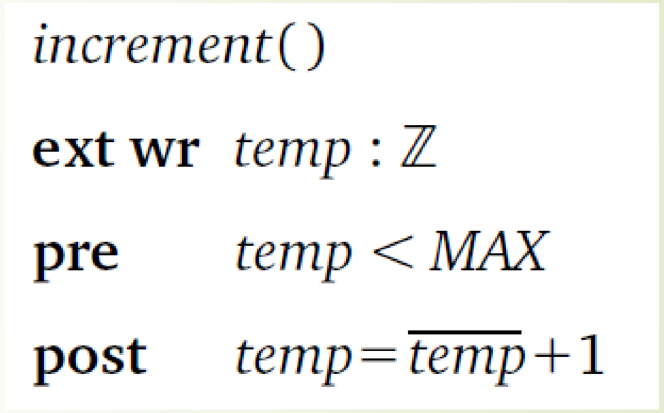


$increment()$

ext wr $temp : \mathbb{Z}$

pre $temp < 10$

post $temp = \overline{temp} + 1$



$increment(\,)$

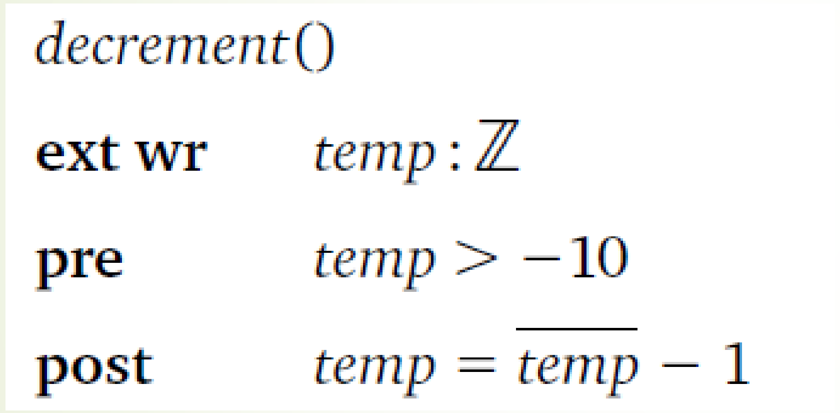
ext wr $temp : \mathbb{Z}$

pre $temp < MAX$

post $temp = \overline{temp} + 1$

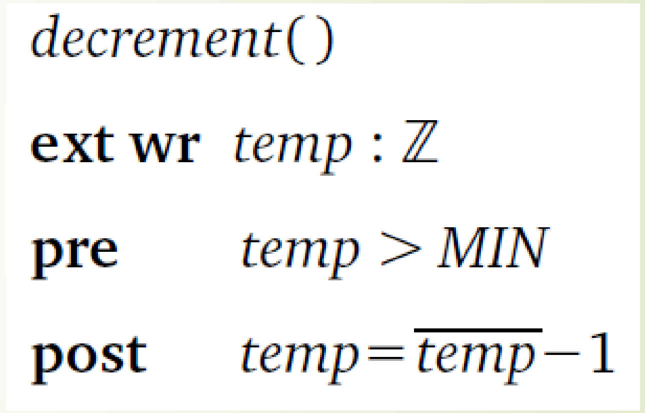
In Z specification language
ext wr == Δ schema
and
temp'=temp- 1

Operation to decrease the temperature



decrement()

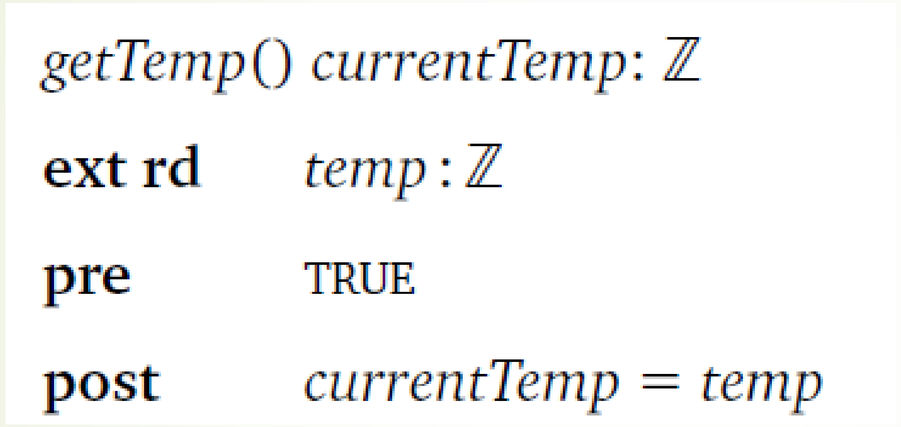
ext wr	$temp : \mathbb{Z}$
pre	$temp > -10$
post	$temp = \overline{temp} - 1$



decrement( )

ext wr	$temp : \mathbb{Z}$
pre	$temp > MIN$
post	$temp = \overline{temp} - 1$

Operation to Read the Temperature



$getTemp()\ currentTemp: \mathbb{Z}$

ext rd	$temp : \mathbb{Z}$
pre	TRUE
post	$currentTemp = temp$

In Z specification language:
ext rd == Ξ
and
*currentTemp***!** = *temp*

VDM Specification of Incubator Monitor

values

$MAX : \mathbb{Z} = 10$
$MIN : \mathbb{Z} = -10$

state *IncubatorMonitor* **of**

$temp : \mathbb{Z}$

inv **mk-**$IncubatorMonitor(t) \triangleq MIN \leq t \leq MAX$

init **mk-**$IncubatorMonitor(t) \triangleq t = 5$

end

operations

increment( )

ext wr $temp : \mathbb{Z}$

pre $temp < MAX$

post $temp = \overline{temp} + 1$

decrement( )

ext wr $temp : \mathbb{Z}$

pre $temp > MIN$

post $temp = \overline{temp} - 1$

getTemp( ) $currentTemp : \mathbb{Z}$

ext rd $temp : \mathbb{Z}$

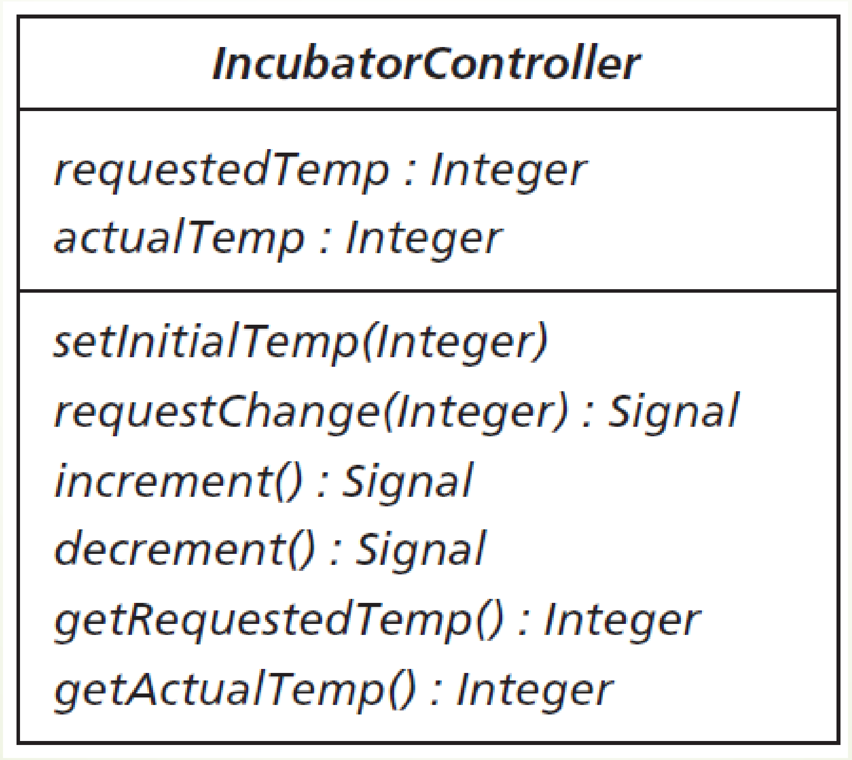
pre TRUE

post $currentTemp = temp$

Improving the incubator system

- In our enhanced system, the software will not only record the current temperature of the system, but will also control the hardware. The system will be able to respond to a request from the user to change the temperature, and subsequently to signal the hardware to increase or decrease the temperature accordingly.
- The hardware itself will still operate in such a way as to either increment or decrement the temperature of the incubator, and to signal the software each time that a change of one degree has been effected. When the software receives such a signal it must, in addition to recording the new temperature, send back a response, telling the hardware whether or not further changes are required to achieve the temperature that has been requested.

The UML specification of the *IncubatorController*

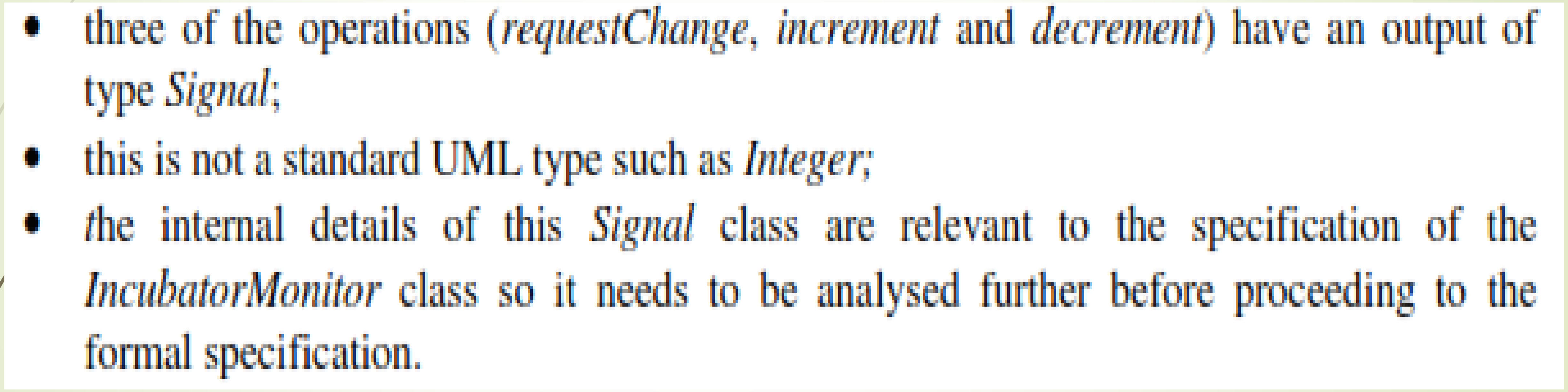


IncubatorController

requestedTemp : Integer
actualTemp : Integer

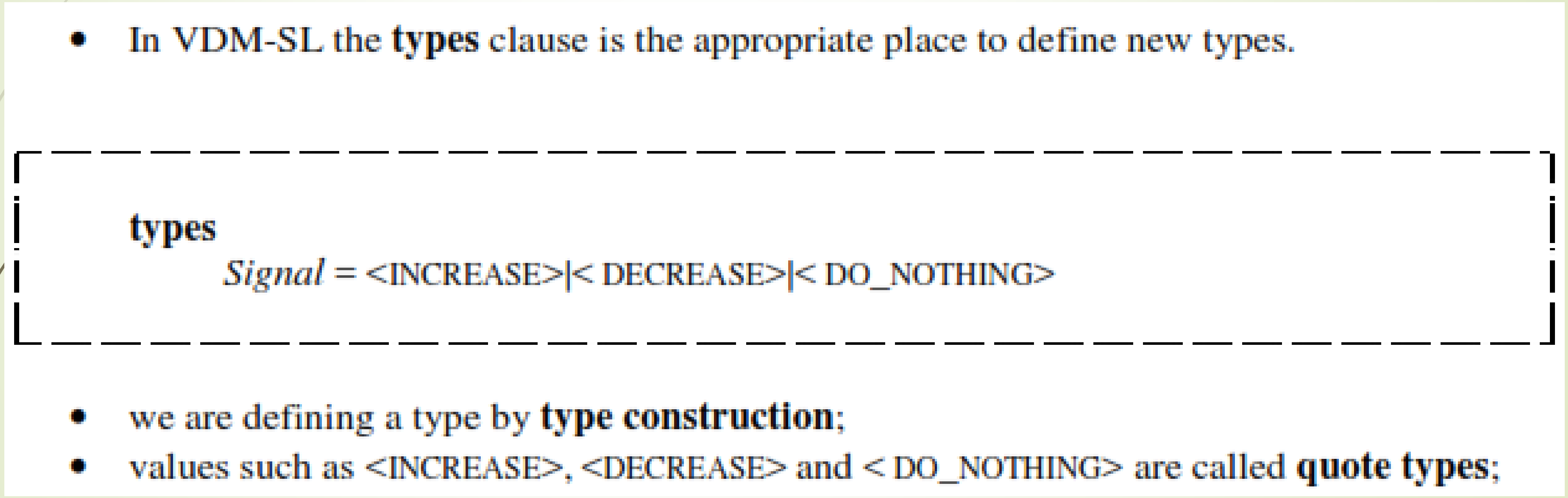
setInitialTemp(Integer)
requestChange(Integer) : Signal
increment() : Signal
decrement() : Signal
getRequestedTemp() : Integer
getActualTemp() : Integer

Specifying enumerated type



- three of the operations (*requestChange*, *increment* and *decrement*) have an output of type *Signal*;
- this is not a standard UML type such as *Integer*;
- the internal details of this *Signal* class are relevant to the specification of the *IncubatorMonitor* class so it needs to be analysed further before proceeding to the formal specification.

Types in VDM



- In VDM-SL the **types** clause is the appropriate place to define new types.

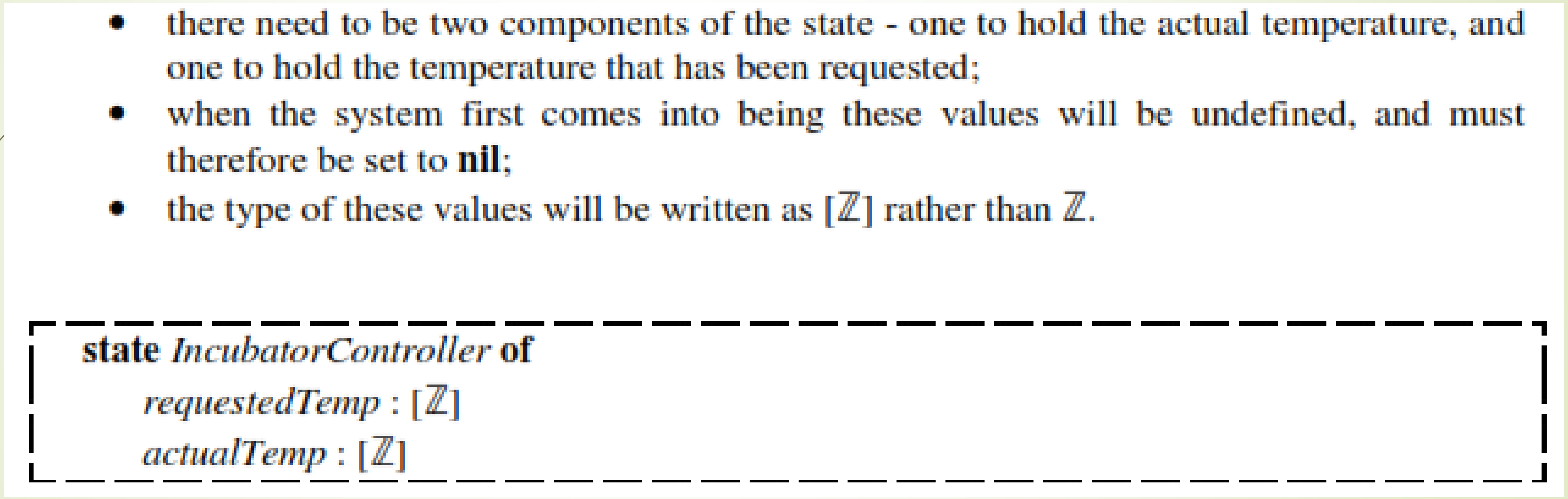
```
types
    Signal = <INCREASE>|< DECREASE>|< DO_NOTHING>
```

- we are defining a type by **type construction**;
- values such as <INCREASE>, <DECREASE> and < DO_NOTHING> are called **quote types**;

Nil value

- It was common in the programming world for a value to be undefined.
- VDM-SL allows for this concept by including the possibility of a term or expression having the value **nil**, meaning that it is undefined.
- Of course, if we want to allow for this possibility, then we need to slightly modify the type of the variable.
- We do that by placing square brackets around the type name – for example [Z] or [N] – meaning that a variable of that type can take the value of **nil**.

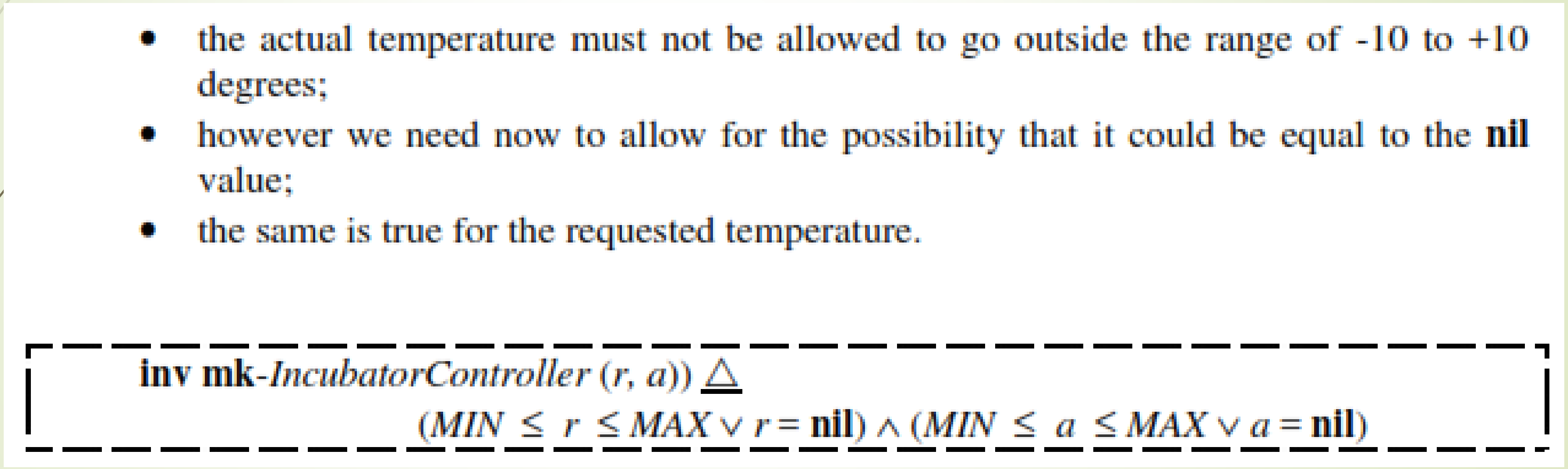
Specifying the state of the *IncubatorController*



- there need to be two components of the state - one to hold the actual temperature, and one to hold the temperature that has been requested;
- when the system first comes into being these values will be undefined, and must therefore be set to **nil**;
- the type of these values will be written as $[\mathbb{Z}]$ rather than $\mathbb{Z}$.

```
state IncubatorController of
    requestedTemp : [ℤ]
    actualTemp : [ℤ]
```

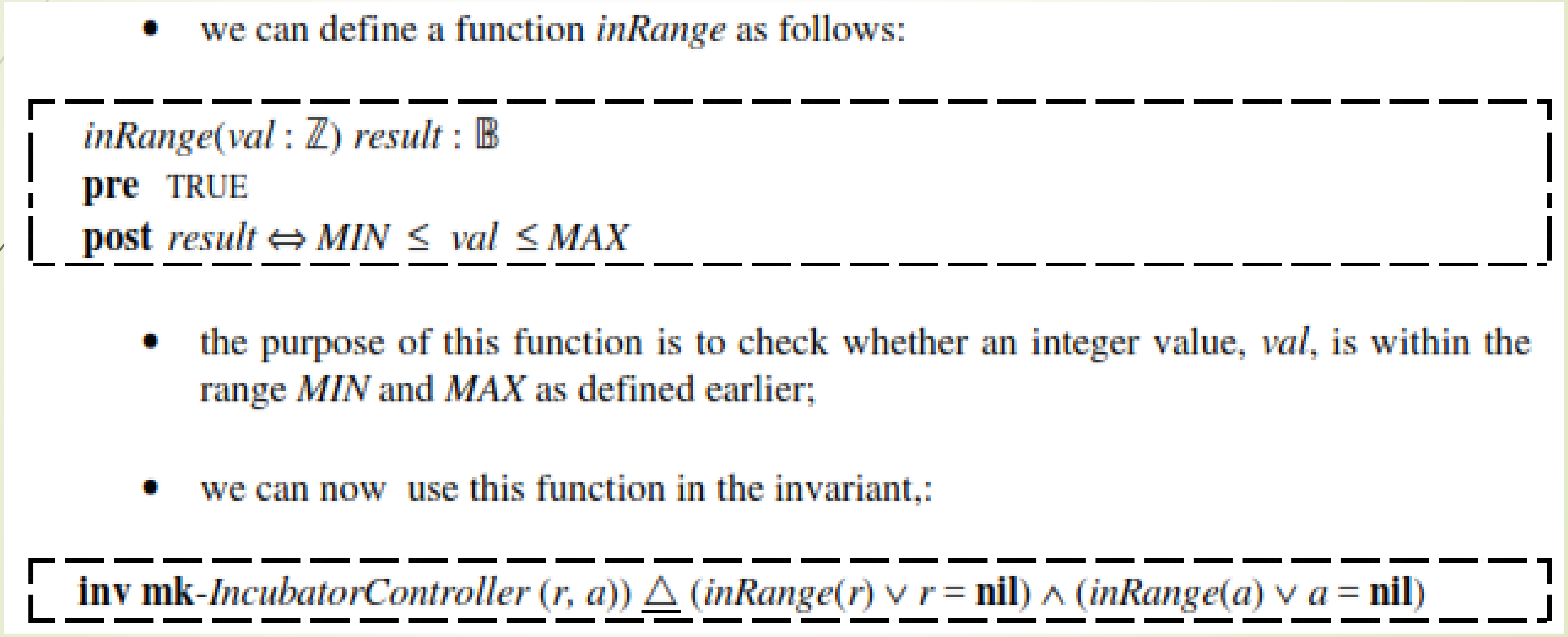
Invariant function



- the actual temperature must not be allowed to go outside the range of -10 to +10 degrees;
- however we need now to allow for the possibility that it could be equal to the **nil** value;
- the same is true for the requested temperature.

$$\textbf{inv mk-}\mathit{IncubatorController}\,(r, a) \triangleq (\mathit{MIN} \leq r \leq \mathit{MAX} \vee r = \textbf{nil}) \wedge (\mathit{MIN} \leq a \leq \mathit{MAX} \vee a = \textbf{nil})$$

Improving the readability of the spec by using a function



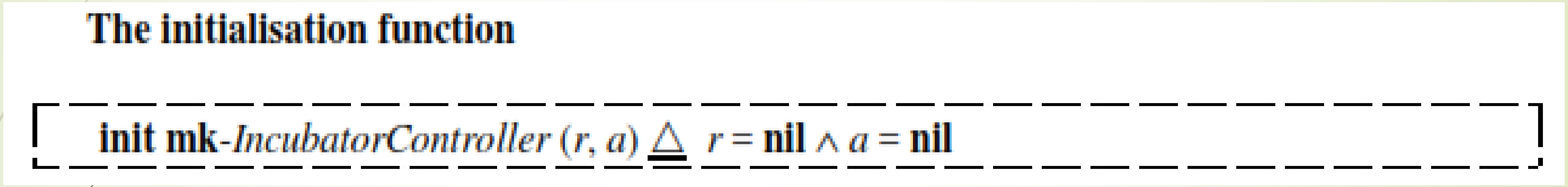
- we can define a function *inRange* as follows:

$$\begin{aligned}&\mathit{inRange}(\mathit{val} : \mathbb{Z})\ \mathit{result} : \mathbb{B}\\ &\mathbf{pre}\ \ \text{TRUE}\\ &\mathbf{post}\ \mathit{result} \Leftrightarrow \mathit{MIN} \leq \mathit{val} \leq \mathit{MAX}\end{aligned}$$

- the purpose of this function is to check whether an integer value, *val*, is within the range *MIN* and *MAX* as defined earlier;

- we can now use this function in the invariant,:

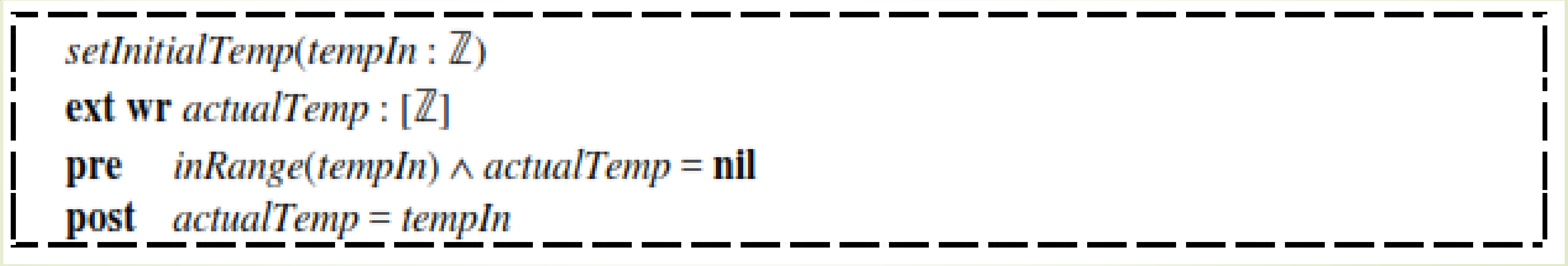
$$\mathbf{inv}\ \mathbf{mk}\text{-}\mathit{IncubatorController}\,(r, a))\ \underline{\triangle}\ (\mathit{inRange}(r) \lor r = \mathbf{nil}) \land (\mathit{inRange}(a) \lor a = \mathbf{nil})$$



The initialisation function

init mk-*IncubatorController* $(r, a) \triangleq r = \textbf{nil} \wedge a = \textbf{nil}$

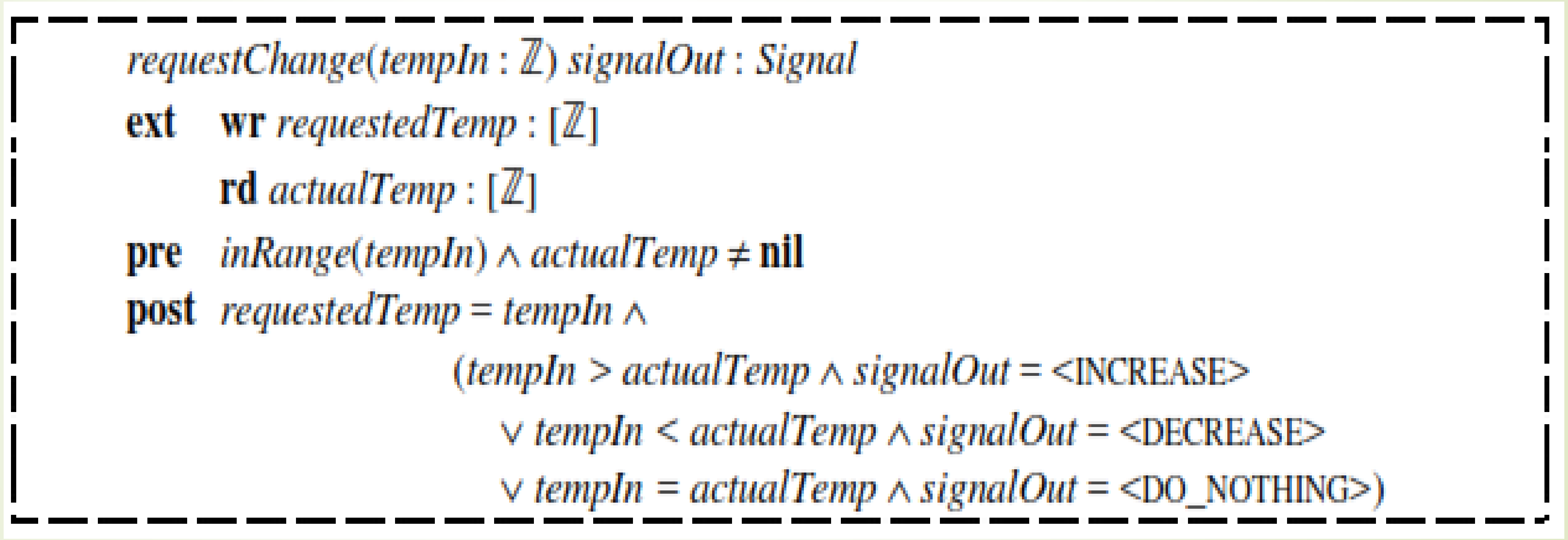
Operation: *setInitialTemp*



setInitialTemp(*tempIn* : $\mathbb{Z}$)
ext wr *actualTemp* : $[\mathbb{Z}]$
pre *inRange*(*tempIn*) $\wedge$ *actualTemp* = **nil**
post *actualTemp* = *tempIn*

In Z; *tempIn?*

Operation to change the temperature



requestChange(*tempIn* : $\mathbb{Z}$) *signalOut* : *Signal*

ext **wr** *requestedTemp* : [$\mathbb{Z}$]

rd *actualTemp* : [$\mathbb{Z}$]

pre *inRange*(*tempIn*) $\wedge$ *actualTemp* $\neq$ **nil**

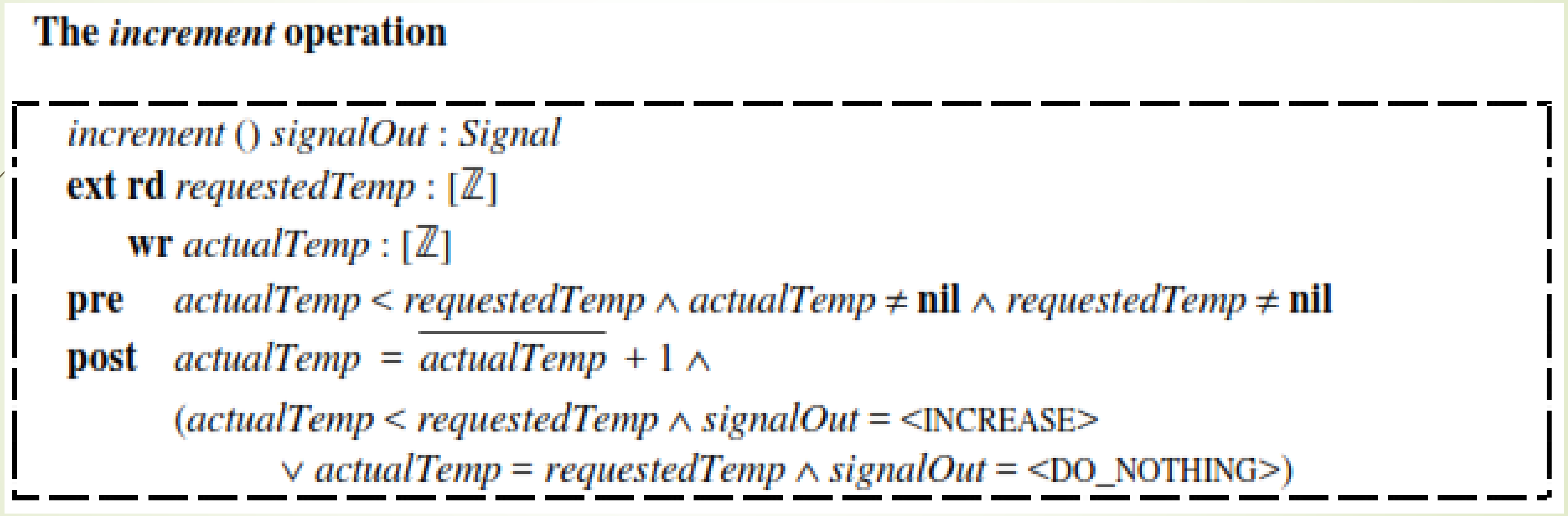
post *requestedTemp* = *tempIn* $\wedge$

(*tempIn* > *actualTemp* $\wedge$ *signalOut* = <INCREASE>

$\vee$ *tempIn* < *actualTemp* $\wedge$ *signalOut* = <DECREASE>

$\vee$ *tempIn* = *actualTemp* $\wedge$ *signalOut* = <DO_NOTHING>)

Operation to increase the temperature



The *increment* operation

increment () *signalOut* : *Signal*

ext rd *requestedTemp* : $[\mathbb{Z}]$

wr *actualTemp* : $[\mathbb{Z}]$

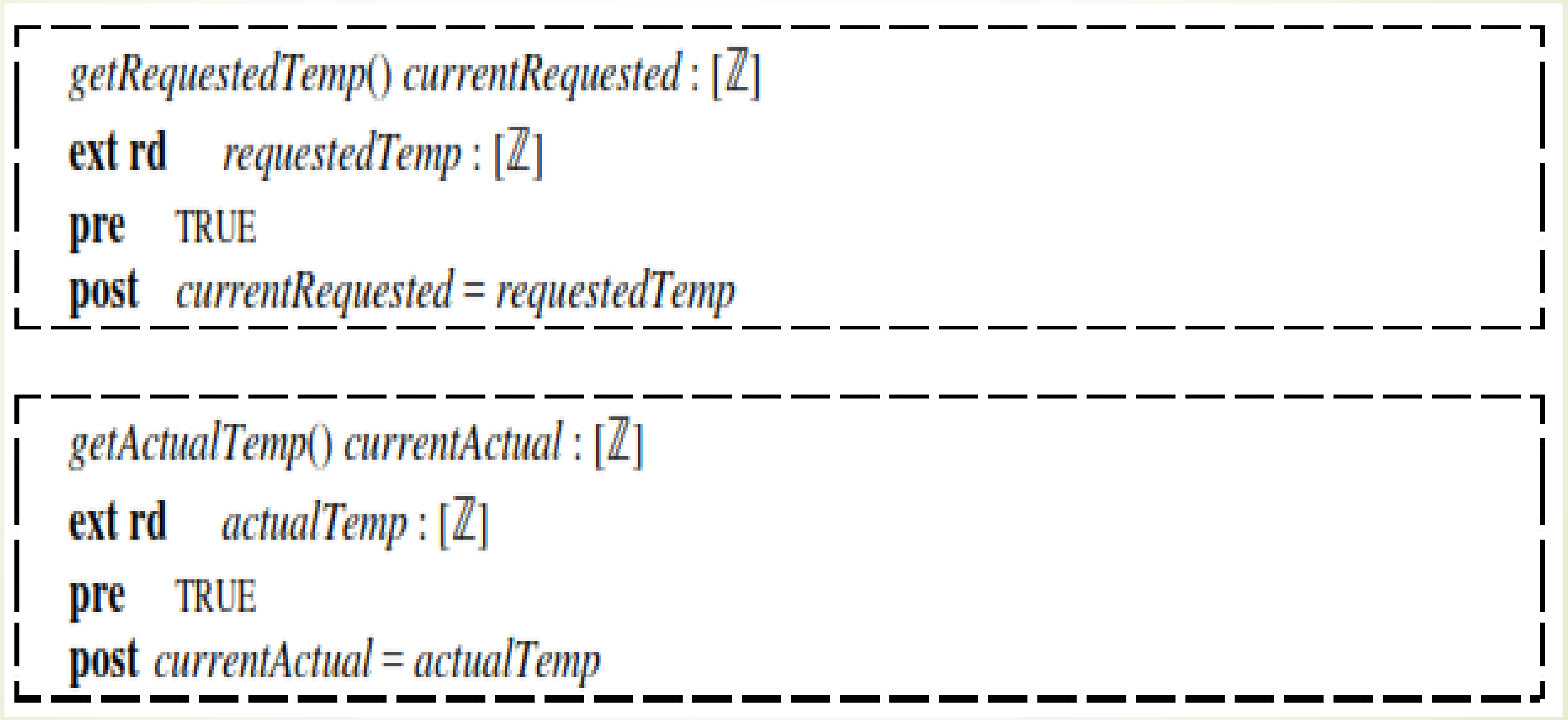
pre $actualTemp < requestedTemp \wedge actualTemp \neq \textbf{nil} \wedge requestedTemp \neq \textbf{nil}$

post $actualTemp = \overline{actualTemp} + 1 \wedge$

$(actualTemp < requestedTemp \wedge signalOut = \text{<INCREASE>}$

$\vee\ actualTemp = requestedTemp \wedge signalOut = \text{<DO\_NOTHING>})$

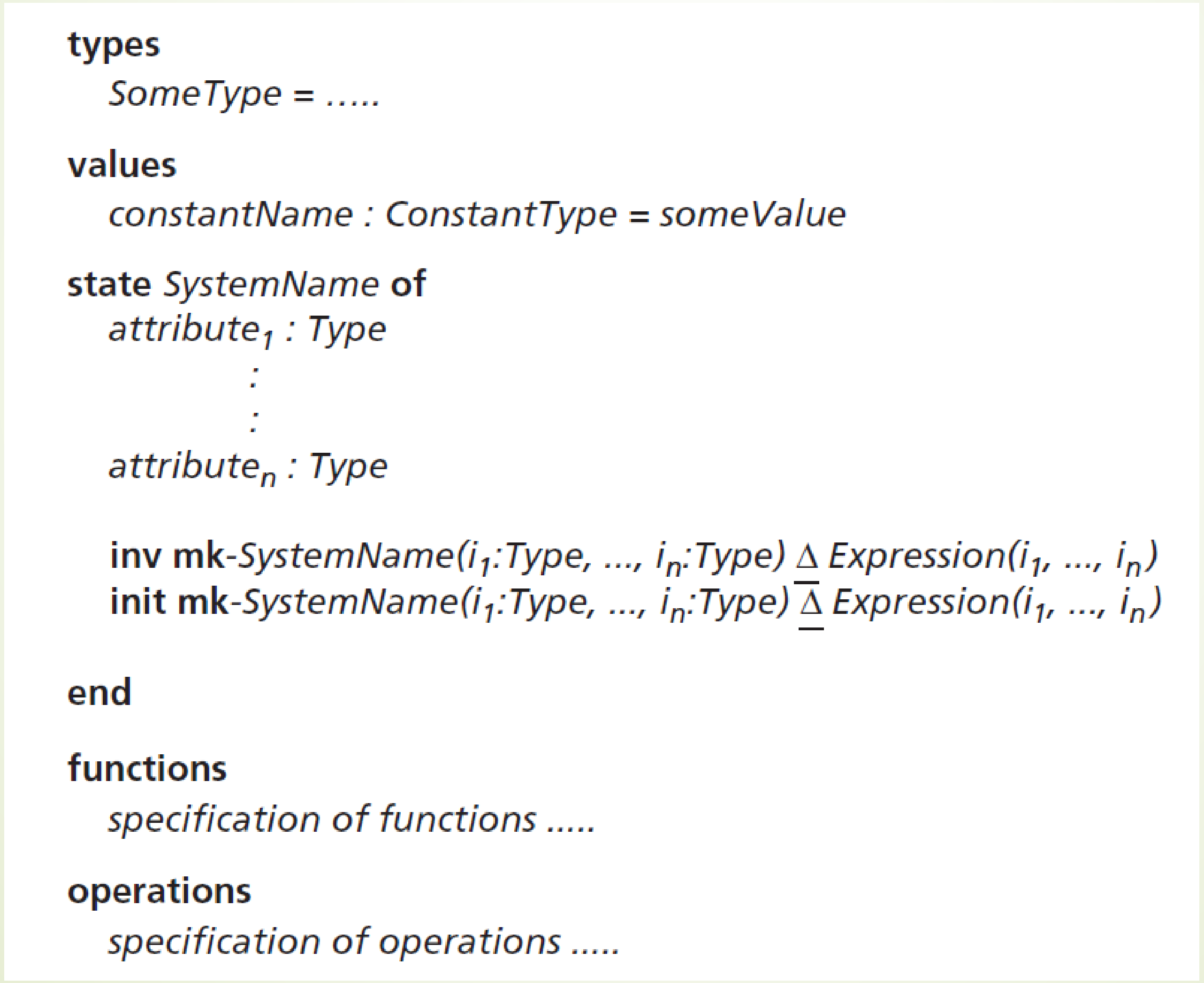
Read operations for the requested temperature and the actual temperature:



getRequestedTemp() currentRequested : $[\mathbb{Z}]$
ext rd *requestedTemp* : $[\mathbb{Z}]$
pre TRUE
post *currentRequested = requestedTemp*

getActualTemp() currentActual : $[\mathbb{Z}]$
ext rd *actualTemp* : $[\mathbb{Z}]$
pre TRUE
post *currentActual = actualTemp*

Standard template of VDM SL specification



types
    SomeType = ...

values
    constantName : ConstantType = someValue

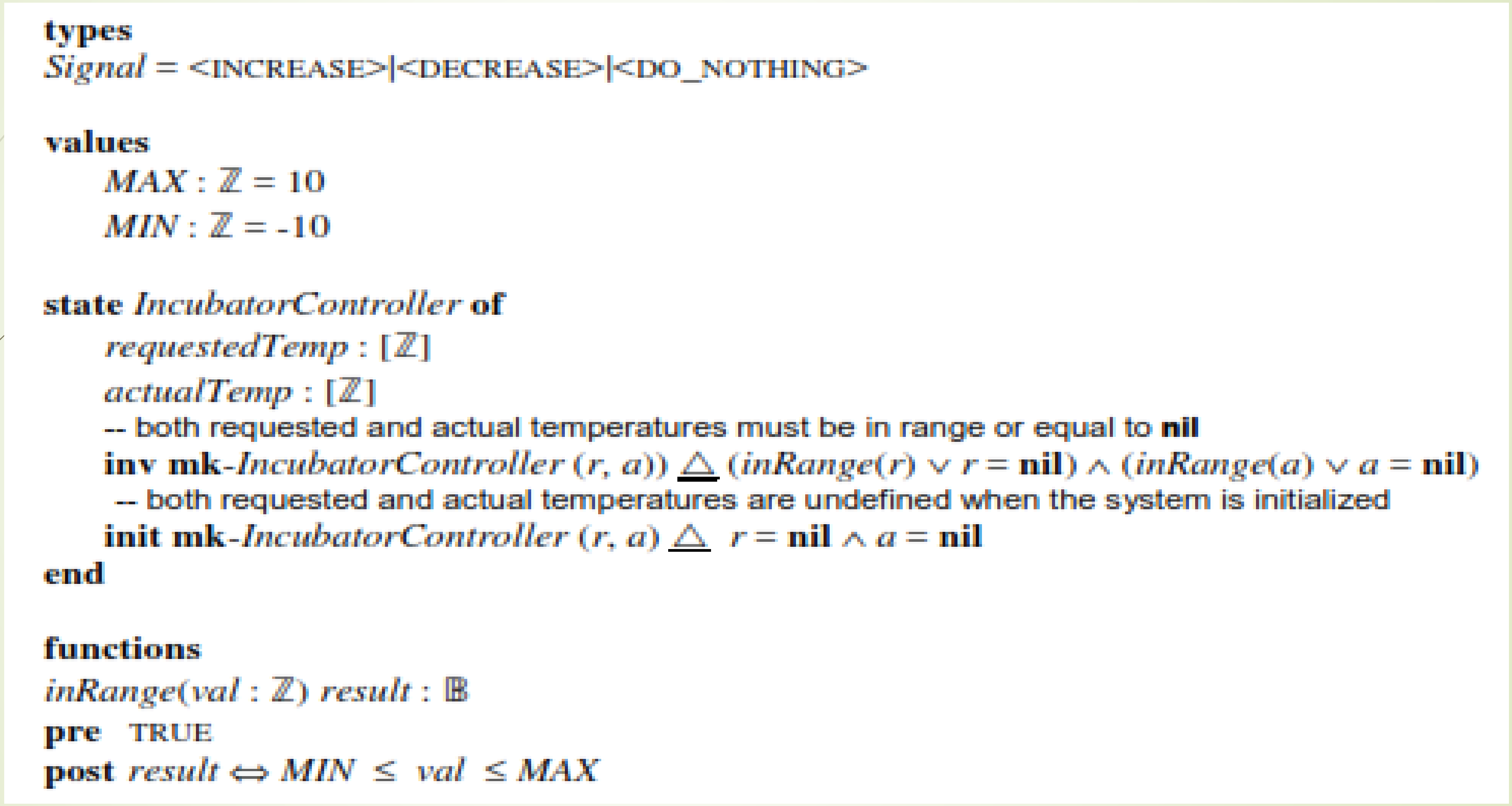
state *SystemName* **of**
    attribute$_1$ *: Type*
                :
                :
    attribute$_n$ *: Type*

    inv mk-*SystemName*(i_1:*Type*, ..., i_n:*Type*) Δ *Expression*(i_1, ..., i_n)
    init mk-*SystemName*(i_1:*Type*, ..., i_n:*Type*) $\underline{\overline{\Delta}}$ *Expression*(i_1, ..., i_n)

end

functions
    specification of functions .....

operations
    specification of operations .....

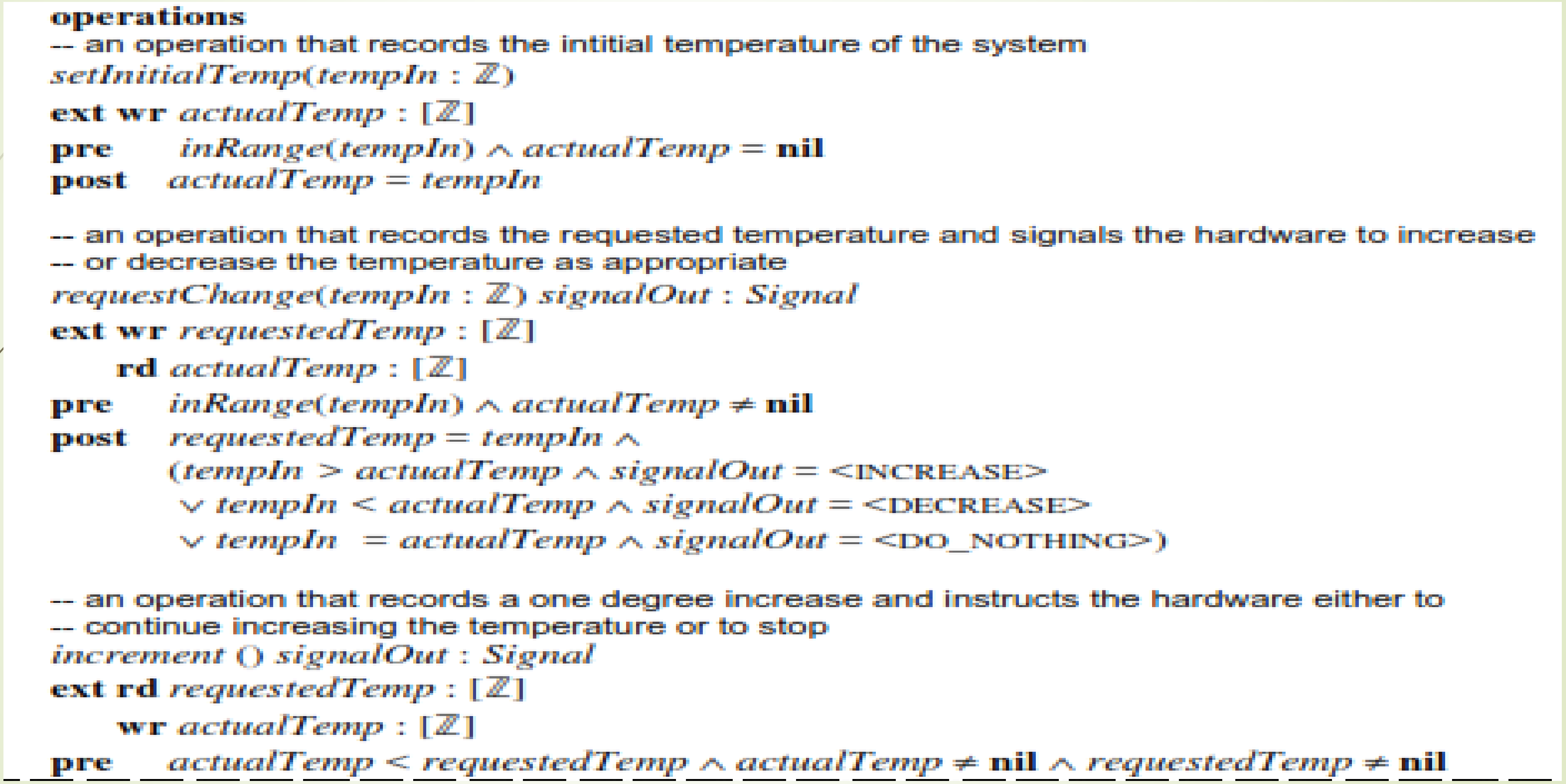


types
Signal = <INCREASE>|<DECREASE>|<DO_NOTHING>

values
$MAX : \mathbb{Z} = 10$
$MIN : \mathbb{Z} = -10$

state *IncubatorController* **of**
$requestedTemp : [\mathbb{Z}]$
$actualTemp : [\mathbb{Z}]$
-- both requested and actual temperatures must be in range or equal to **nil**
inv mk-*IncubatorController* $(r, a)) \underline{\triangle} (inRange(r) \vee r = \mathbf{nil}) \wedge (inRange(a) \vee a = \mathbf{nil})$
-- both requested and actual temperatures are undefined when the system is initialized
init mk-*IncubatorController* $(r, a) \underline{\triangle}\ r = \mathbf{nil} \wedge a = \mathbf{nil}$
end

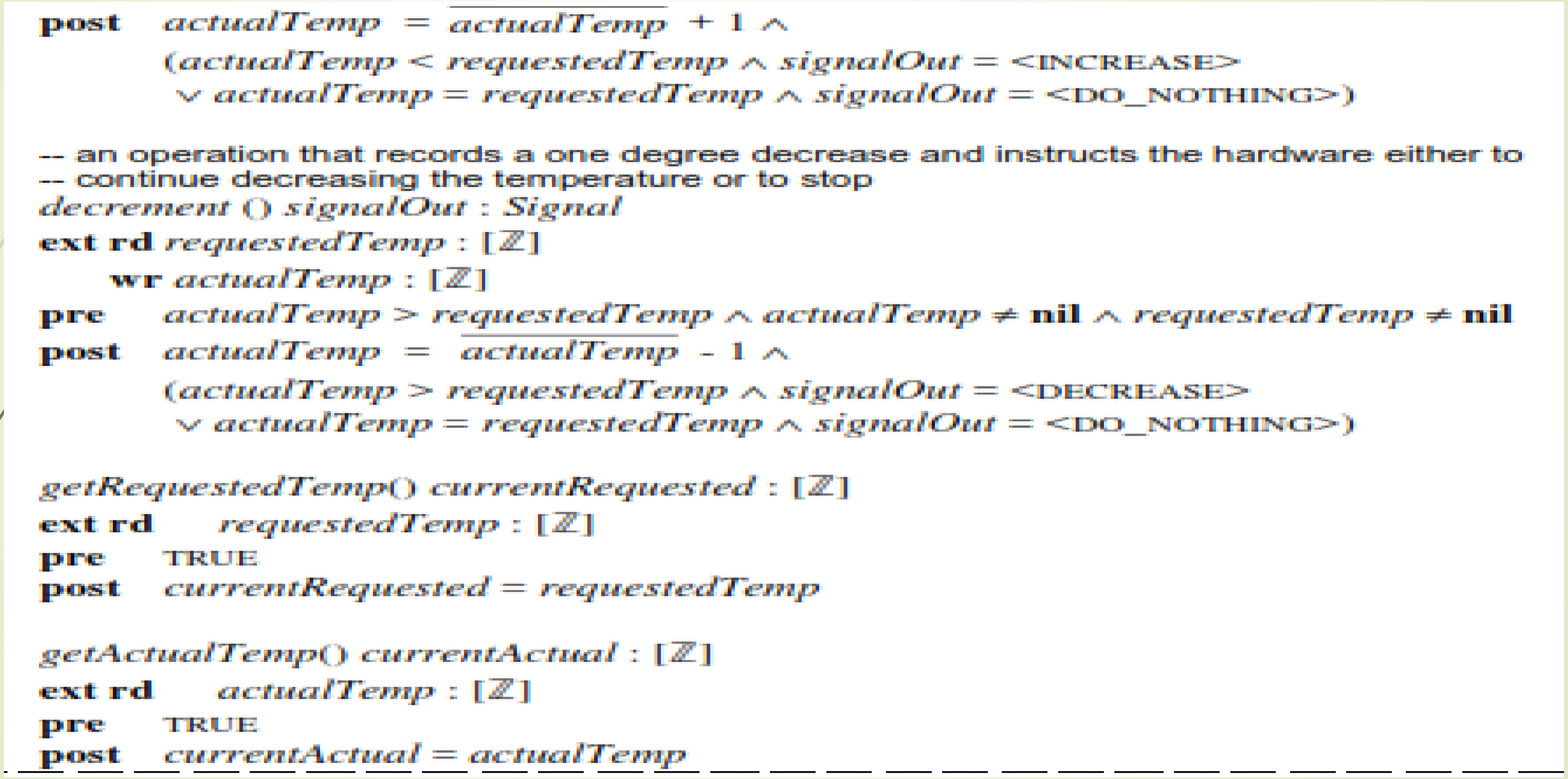
functions
$inRange(val : \mathbb{Z})\ result : \mathbb{B}$
pre TRUE
post $result \Leftrightarrow MIN \le val \le MAX$



```
operations
-- an operation that records the intitial temperature of the system
setInitialTemp(tempIn : ℤ)
ext wr actualTemp : [ℤ]
pre     inRange(tempIn) ∧ actualTemp = nil
post    actualTemp = tempIn

-- an operation that records the requested temperature and signals the hardware to increase
-- or decrease the temperature as appropriate
requestChange(tempIn : ℤ) signalOut : Signal
ext wr requestedTemp : [ℤ]
    rd actualTemp : [ℤ]
pre     inRange(tempIn) ∧ actualTemp ≠ nil
post    requestedTemp = tempIn ∧
        (tempIn > actualTemp ∧ signalOut = <INCREASE>
         ∨ tempIn < actualTemp ∧ signalOut = <DECREASE>
         ∨ tempIn  = actualTemp ∧ signalOut = <DO_NOTHING>)

-- an operation that records a one degree increase and instructs the hardware either to
-- continue increasing the temperature or to stop
increment () signalOut : Signal
ext rd requestedTemp : [ℤ]
    wr actualTemp : [ℤ]
pre     actualTemp < requestedTemp ∧ actualTemp ≠ nil ∧ requestedTemp ≠ nil
```



post $actualTemp = \overline{actualTemp} + 1 \land$
$(actualTemp < requestedTemp \land signalOut = \text{<INCREASE>}$
$\lor\ actualTemp = requestedTemp \land signalOut = \text{<DO\_NOTHING>})$

-- an operation that records a one degree decrease and instructs the hardware either to
-- continue decreasing the temperature or to stop

decrement () signalOut : Signal
ext rd $requestedTemp : [\mathbb{Z}]$
  wr $actualTemp : [\mathbb{Z}]$
pre $actualTemp > requestedTemp \land actualTemp \neq \textbf{nil} \land requestedTemp \neq \textbf{nil}$
post $actualTemp = \overline{actualTemp} - 1 \land$
$(actualTemp > requestedTemp \land signalOut = \text{<DECREASE>}$
$\lor\ actualTemp = requestedTemp \land signalOut = \text{<DO\_NOTHING>})$

getRequestedTemp() currentRequested : $[\mathbb{Z}]$
ext rd $requestedTemp : [\mathbb{Z}]$
pre TRUE
post $currentRequested = requestedTemp$

getActualTemp() currentActual : $[\mathbb{Z}]$
ext rd $actualTemp : [\mathbb{Z}]$
pre TRUE
post $currentActual = actualTemp$

Reference and reading material

- Chapter # 3: An Introduction to Specification in VDM-SL of the book "Formal Software Development, from VDM to Java" by Quentin Charatan and Aaron Kans