CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

Theory and Problems of DISCRETE MATHEMATICS, Third Edition, by SEYMOUR LIPSCHUTZ and MARC LARS LIPSON

TAUTOLOGIES AND CONTRADICTIONS

- Some propositions P(p, q, . . .) contain only T in their evaluation or, in other words, they are true for any truth values of their variables. Such propositions are called tautologies.
- Analogously, a proposition P(p, q, . . .) is called a contradiction if it contains only F in its evaluation or, in other words, if it is false for any truth values of its variables.

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

ARGUMENTS

- An argument is an assertion that a given set of propositions P_1, P_2,...,P_n, yields another proposition Q.
- An argument provides support or evidence in favor of one of the others.
- An argument P_1, P_2, ..., $P_n \vdash Q$ is said to be valid if Q is TRUE whenever all the premises P_1, P_2,...,P_n are true.
- Eg: valid argument

$$p, p \rightarrow q \vdash q$$

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

ARGUMENTS

- Now the propositions P1, P2, . . . , Pn are true simultaneously if and only if the proposition P1 ∧ P2 ∧ . . . Pn is true.
- Thus the argument P1, P2, . . . , Pn ⊢ Q is valid if and only if Q is true whenever P1 ∧ P2 ∧ . . . ∧ Pn is true, or, equivalently, if the proposition (P1 ∧ P2 ∧ . . . ∧ Pn) → Q is a tautology.

ARGUMENTS

Lets determine the validity of the following argument:

p → q, ¬ q ⊢ ¬p.

p	q	$[(p$	$\rightarrow$	$q)$	$\wedge$	$\neg$	$q]$	$\rightarrow$	$\neg$	p
T	T	T	T	T	F	F	T	T	F	T
T	F	T	F	F	F	T	F	T	F	T
F	T	F	T	T	F	F	T	T	T	F
F	F	F	T	F	T	T	F	T	T	F
Step		1	2	1	3	2	1	4	2	1

ARGUMENTS

- An argument which is not valid is called fallacy.
- Eg: fallacy argument
- $p \rightarrow q, q \vdash p$

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

ARGUMENTS

Show that the following argument is a fallacy: p → q, ¬p ⊢ ¬q.

p	q	$p \rightarrow q$	$\neg p$	$(p \rightarrow q) \wedge \neg p$	$\neg q$	$[(p \rightarrow q) \wedge \neg p] \rightarrow \neg q$
T	T	T	F	F	F	T
T	F	F	F	F	T	T
F	T	T	T	T	F	F
F	F	T	T	T	T	T

CS636 Formal Methods

Topics Covered

Propositional Logic

References

- All the contents of this video lecture and PPT slides are taken from:

- Formal Software Development From VDM to Java, by Quentin Charatan and Aaron Kans

THREE-VALUED LOGIC

- Classical logic assumes that all expressions evaluate to TRUE or FALSE.
- In reality, this is not always the case when evaluating an expression, because sometimes an expression can be undefined
  - For example, the expression 0/0.
- Undefined terms are very common in programming situations – for example, when a variable is first declared and has not yet been assigned a value.
- In this system of logic a proposition could have the value TRUE, FALSE or UNDEFINED.

AND Operator

P	Q	$P \wedge Q$
T	T	**T**
T	F	**F**
T	UNDEFINED	**UNDEFINED**
F	T	**F**
F	F	**F**
F	UNDEFINED	**F**
UNDEFINED	T	**UNDEFINED**
UNDEFINED	F	**F**
UNDEFINED	UNDEFINED	**UNDEFINED**

OR Operator

P	Q	$P \vee Q$
T	T	**T**
T	F	**T**
T	UNDEFINED	**T**
F	T	**T**
F	F	**F**
F	UNDEFINED	**UNDEFINED**
UNDEFINED	T	**T**
UNDEFINED	F	**UNDEFINED**
UNDEFINED	UNDEFINED	**UNDEFINED**

IMPLICATION Operator

P	Q	$P \Rightarrow Q$
T	T	**T**
T	F	**F**
T	UNDEFINED	**UNDEFINED**
F	T	**T**
F	F	**T**
F	UNDEFINED	**T**
UNDEFINED	T	**T**
UNDEFINED	F	**UNDEFINED**
UNDEFINED	UNDEFINED	**UNDEFINED**

Equivalence Operator

P	Q	$P \Leftrightarrow Q$
T	T	**T**
T	F	**F**
T	UNDEFINED	**UNDEFINED**
F	T	**F**
F	F	**T**
F	UNDEFINED	**UNDEFINED**
UNDEFINED	T	**UNDEFINED**
UNDEFINED	F	**UNDEFINED**
UNDEFINED	UNDEFINED	**UNDEFINED**

Exclusive OR Operator

P	Q	$P \oplus Q$
T	T	**F**
T	F	**T**
T	UNDEFINED	**UNDEFINED**
F	T	**T**
F	F	**F**
F	UNDEFINED	**UNDEFINED**
UNDEFINED	T	**UNDEFINED**
UNDEFINED	F	**UNDEFINED**
UNDEFINED	UNDEFINED	**UNDEFINED**

Negation Operator

P	$\neg P$
T	F
F	T
UNDEFINED	UNDEFINED

CS636 Formal Methods

Topics Covered

Predicate Logic

References

- All the contents of this video lecture and PPT slides are taken from:

- Formal Software Development From VDM to Java, by Quentin Charatan and Aaron Kans

Predicate Logic

- One of the limitations with the propositional logic is that, while it allows us to argue about individual values, it does not give us the ability to argue about sets of values.
- A set is any well-defined, unordered, collection of objects.
  - The set of whole numbers from 1 to 10
  - The set of the days of the week
- In mathematics, we often represent a set elements by lower-case letters. For example:
- A = {s, d, f, h, k}
- B = {a, b, c, d, e, f}
- The symbol ∈ means ‘is an element of’. Therefore the statement ‘d is an element of A’ is written: d ∈ A

Predicate Logic

- For the purpose of reasoning about sets of values, a more powerful tool than the propositional logic has been devised, namely the predicate logic.
- A predicate is a truth-valued expression containing free variables. These allow the expression to be evaluated by giving different values to the variables.
- Once the variables are evaluated they are said to be bound.

Example of predicates

- Predicates can be named with either a single letter, or with a word that expresses the meaning of the predicate; the variables are placed in brackets after the name.
- This is made clear in the following examples:
  - C(x): x is a cat
  - Studies(x,y): x studies y
  - Prime(n): n is a prime number
- A statement such as C(x) can be read C of x.
- Predicates such as those above do not yet have a value – they only have a value when the variables themselves are given a value.

Binding variables

- Predicates do not yet have a value – they only have a value when the variables themselves are given a value.
- Two ways:
  - Substitution
  - Quantification

Binding (Substitution)

- For example, using the previous three predicates:

- C(Simba): Simba is a cat
- Studies(Ali, physics): Ali studies physics
- Prime(3): 3 is a prime number

The above expressions now have a value of TRUE or FALSE.

Binding (Quantification)

- A quantifier is a mechanism for specifying an expression about a set of values. There are three quantifiers that we can use, each with its own symbol:
- Universal Quantifier
- Existential Quantifier
- Unique Existential Quantifier

Quantification

- The universal quantifier ∀:
- This quantifier enables a predicate to make a statement about all the elements in a particular set. For example, if M(x) is the predicate x chases mice, we could write:
- ∀x ∈ Cats ● M(x)
- This reads *For all the x which are members of the set Cats, x chases mice*, or, more simply, *All cats chase mice*.

Quantification

- – The existential quantifier ∃
- In this case, a statement is made about whether or not at least one element of a set meets a particular criterion.
- For example, if, as above, P(n) is the predicate *n is a prime number*, then we could write:
- $\exists n \in \mathbb{N} \bullet P(n)$
- This reads *There exists an n in the set of natural numbers such that n is a prime number*, or, put another way, *There exists at least one prime number in the set of natural numbers*.

Quantification

- –The unique existential quantifier ∃!
- This quantifier modifies a predicate to make a statement about whether or not precisely one element of a set meets a particular criterion.
- For example, if G(x) is the predicate *x is green*, we could write
- ∃!x ∈ Cats ● G(x)
- This would mean *There is one and only one cat that is green*.

CS636 Formal Methods

week [8 to 16]

Topics Covered: 058 – 065

Introducton to Specification in VDM-SL (Part 1)

Intoduction to Specification in VDM-SL

At the end of this chapter you should be able to:

- write a formal specification of a system in VDM-SL
- correlate the components of a UML class diagram with those of a VDM specification
- declare constants and specify functions to enhance the specification
- explain the use of a state invariant to place a global constraint on the system
- explain the purpose of the nil value in VDM

Formal Methods

Topic#058

The Case Study: Requirements Analysis

The Case Study: Requirements Analysis

- The example we will use throughout this chapter will be that of an incubator, the temperature of which 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 developing any software system the first stage in the process involves an analysis of the system and an initial statement of the requirements.
- It is very important, in any requirements definition, to be clear about the system boundaries, and we should make it clear here that in this initial version, control of the hardware lies outside of our system.
- In other words, for the time being we will be specifying a system that simply monitors the temperature of the incubator.

The Incubator - Case Study

- The hardware increments or decrements the temperature of the incubator in response to instructions (from someone or something outside of our system), and 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 degree celsius, nor fall below -10 degree celsius.

Formal Methods

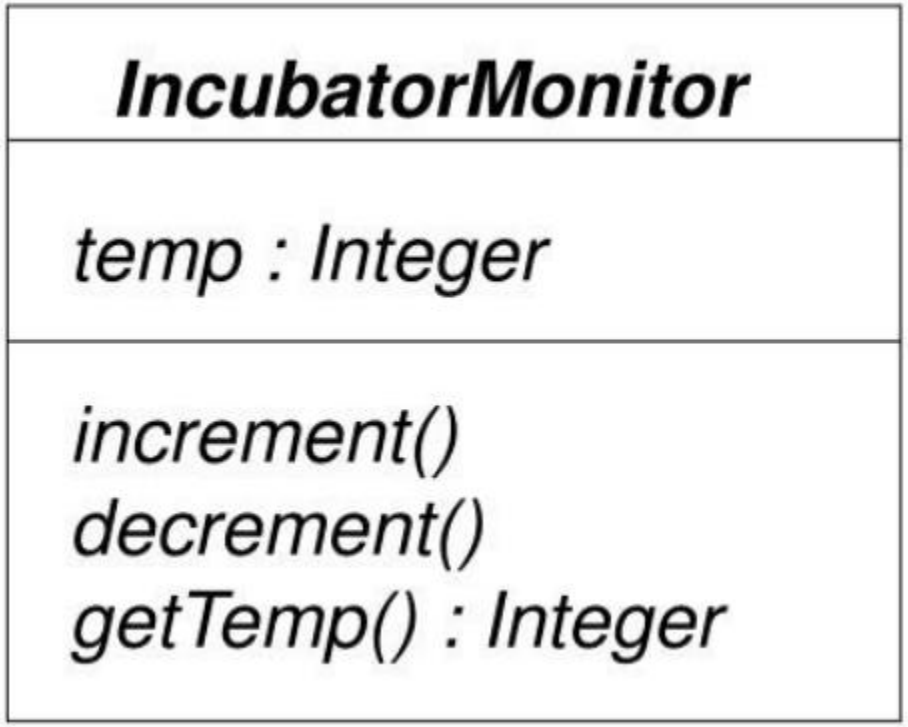
Topic#058

END!

Formal Methods

Topic#059
The UML Specification

The UML Specification



The Specification of the IncubatorMonitor Class

- In the simple system described in the previous slide, we can identify a single class, IncubatorMonitor.
- We have identified one attribute and three methods.
- The single attribute records the temperature of the system and will be of type integer.
- With regard to the methods, the first two 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.

Formal Methods

Topic#059

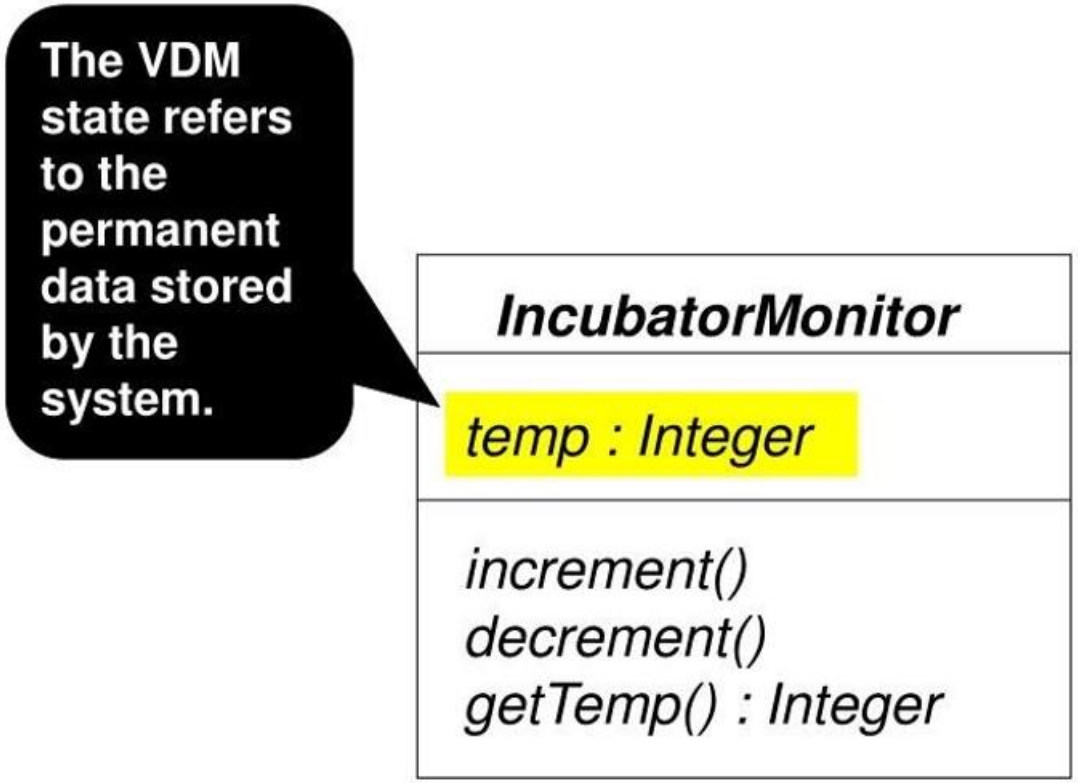
END!

Formal Methods

Topic#060
Specifying the State

Specifying the State

- The first thing we will consider for our formal specification is what 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.



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)

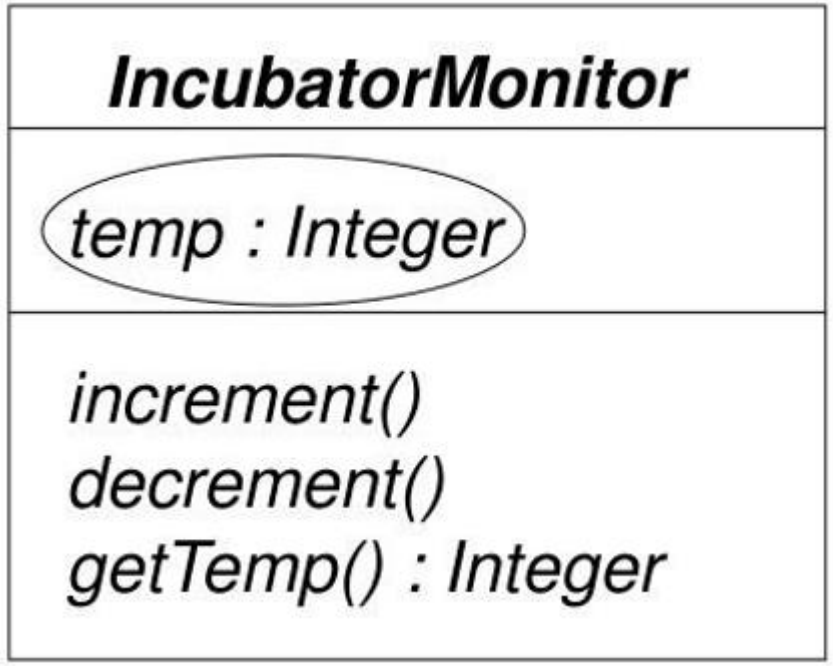
R: real numbers (positive and negative numbers that can include a fractional part)

B: boolean values (TRUE or FALSE)

Char: the set of alphanumeric characters

Specifying the state of Incubator Monitor System

UML



VDM-SL

```
state IncubatorMonitor of
    temp : ℤ
end
```

Formal Methods

Topic#060

END!

Formal Methods

Topic#061
Specifying the Operations

Specifying the Operations

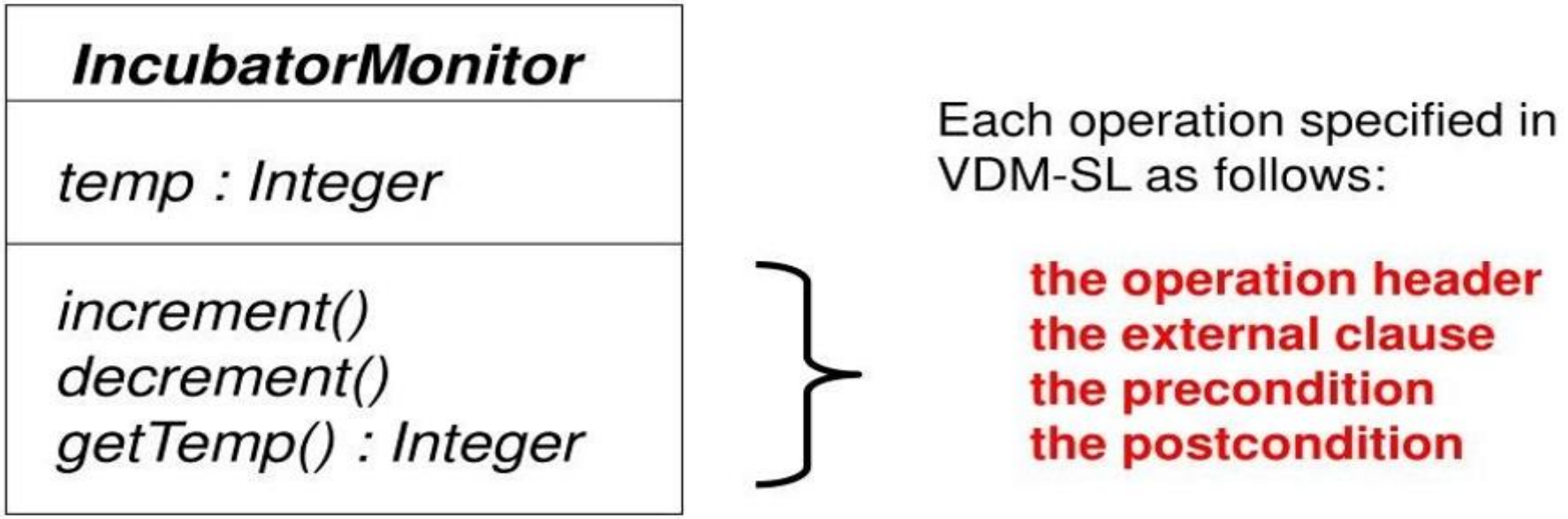
- We need to specify a number of operations that the system should be able to perform and by which means the data (that is the state) can be accessed.
- In VDM we tend to use the word operation, whereas in most object-oriented texts you will tend to see the word method.
- In VDM, operations by definition access the state in some way, either by reading or writing the data, or both.

Operations in VDM-SL

In VDM-SL an operation consists of four sections:

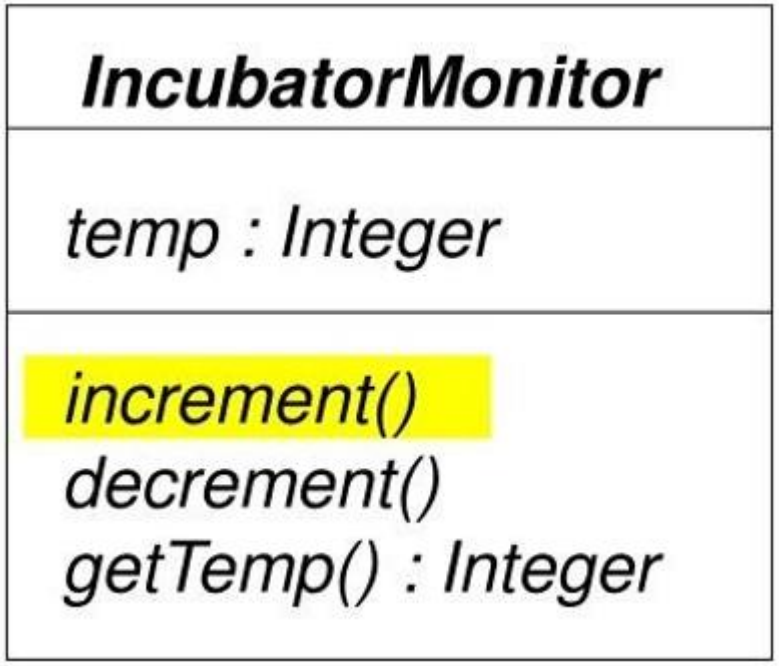
- the operation header
- the external clause
- the precondition
- the postcondition

We need to consider: an operation that records an increment in the temperature; an operation that records a decrement; and one that simply reads the value of the temperature.



Increment Operation

UML



VDM-SL

increment()

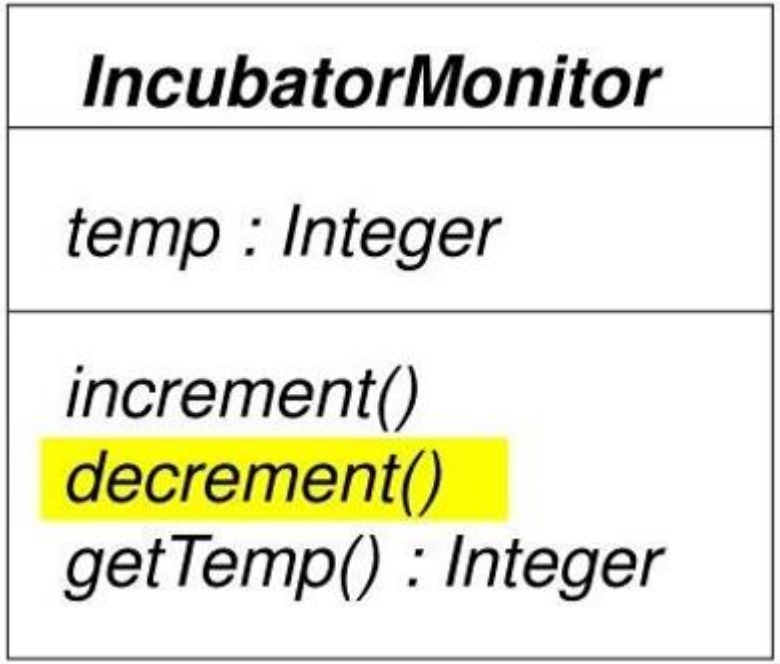
ext wr $temp:\mathbb{Z}$

pre $temp < 10$

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

Decrement Operation

UML



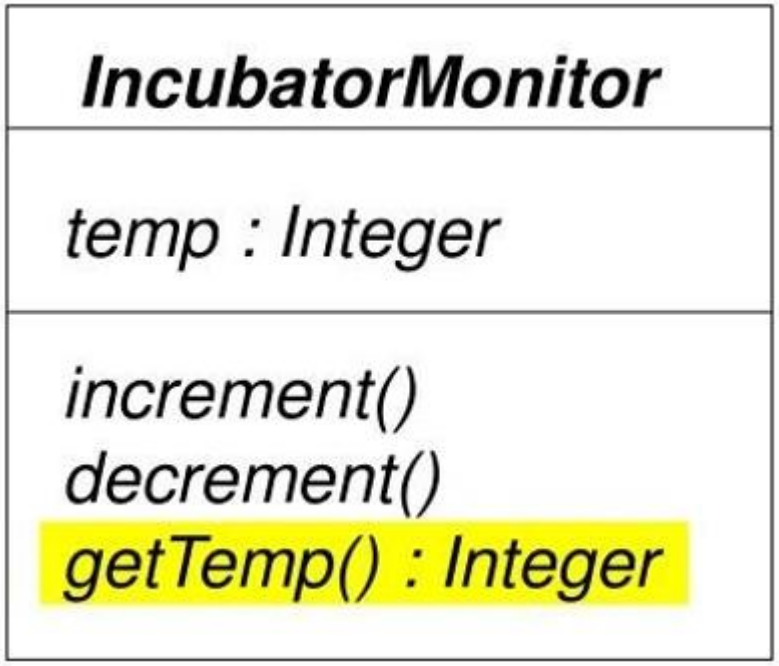
VDM-SL

decrement()

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

getTemp Operation

UML



VDM-SL

getTemp() *currentTemp*: $\mathbb{Z}$
ext rd *temp* : $\mathbb{Z}$
pre TRUE
post *currentTemp* = *temp*

Formal Methods

Topic#061

END!

Formal Methods

Topic#062
Declaring Constants

Declaring Constants

- As with many programming languages, it is possible in VDM-SL to specify constants.
- This is something that is not essential to any specification, but can greatly enhance its readability.
- It is done by using the keyword **values**, and the declaration would come immediately before the state definition.

- In the case of the IncubatorMonitor it would look like this:

values

$MAX : \mathbb{Z} = 10$

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

- These values could then be used in our functions and operations, so, for example, the precondition of the decrement operation would now look like this:

pre $temp > MIN$

Formal Methods

Topic#062

END!

Formal Methods

Topic#063
Specifying Functions

What is a Function?

A function is a set of assignments from one set to another. Thus, the function receives an input value (or values) and maps this to an output value according to some rule – for example it could accept an integer and output the square of that integer, or it could accept the name of a person and output that person's telephone number.

Specifying a Function in VDM-SL

There are two ways in which we can specify a function in VDM-SL:

- Explicitly
- Implicitly

Explicit Function

- The style of this specification is algorithmic, and we explicitly define the method of transforming the inputs to the output. This is illustrated in the following very simple function that adds two numbers together:

$$add: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$
$$add(x, y) \triangleq x + y$$

- The first line is called the **function signature**.
- The second part is the **function definition.**

Implicit Function

- In this method, we use a pre- and postcondition in the same way as we described for operations – a function, of course, does not access the state variables. Here is the add function defined implicitly.

$add(x:\mathbb{R}, y:\mathbb{R})\ z:\mathbb{R}$

pre TRUE

post $z = x + y$

Formal Methods

Topic#063

END!

Formal Methods

Topic#064
Specifying a State Invariant

Specifying a State Invariant

- You have seen that our requirements definition states that the temperature of the incubator must stay within the range -10 to +10 degree celsius. In VDM-SL there is a mechanism by which we can incorporate such a restriction into the specification of the state. This mechanism involves specifying a function known as a state **invariant**.
- By specifying such a function, we are creating a global constraint, rather than just a local constraint as we did with our preconditions.

- The invariant definition uses the keyword **inv**
- In previous topic you studied about function signature. So, in the case of an invariant function, inv, its signature will be:

$$\mathbf{inv}: State \to \mathbb{B}$$

- 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.

- For the IncubatorMonitor system the invariant is specified as:

$$\textbf{inv mk-}IncubatorMonitor(t) \triangleq MIN \leq t \leq MAX$$

- After the keyword inv, we have the expression mk-IncubatorMonitor(t), which effectively is the input to the inv function. This expression is itself a function, and is known as a make function (the mk is pronounced ‘make’).

Invariant gives the opportunity to think about the idea of mathematical proof and integrity checking in connection with software development.

Formal Methods

Topic#064

END!

Formal Methods

Topic#065

Specifying an Initialization Function

Specifying an Initialization Function

- You may already have identified one of the shortcomings of the above specification, namely that we have not yet made any statement about what the value of the temperature should be when the system is first brought into being. It is all very well having operations that increment and decrement the temperature, but if we do not know what the initial value of the temperature was, then they are not very meaningful.
- This problem can be solved by specifying an **initialization function**, which is given the name **init**.
- This function is specified after the declaration of the invariant, and prescribes the conditions that the system must satisfy when it is first brought into being.

- Let us illustrate this with our IncubatorMonitor example. We will assume that the system works in the following way: when the incubator is turned on, its temperature is adjusted until a steady 5 degree celsius is obtained. At this point the software system is activated. Thus, our initialization function should state that when the system is first invoked, the temperature should be set to 5.
- We write the initialization function like this:

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

It is very important to note that the initialization function – as with the operations must preserve the invariant.

Formal Methods

Topic#065

END!

CS636 Formal Methods

Topics Covered: 066 – 073

Introducton to Specification in VDM-SL (Part 2)

Formal Methods

Topic#066
User-defined Types

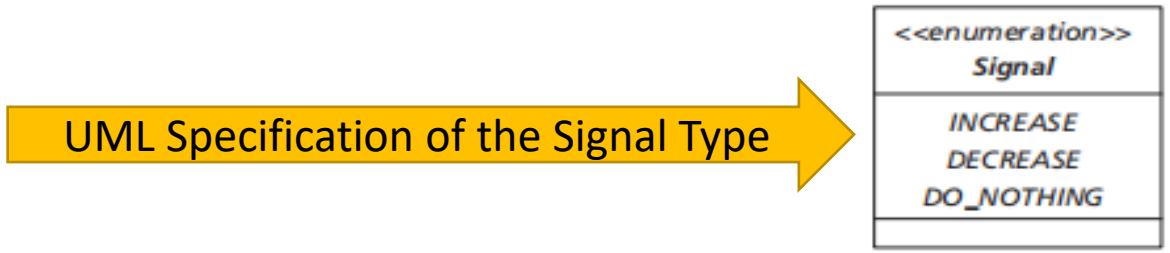
Components of VDM-SL

As we studied, the components of a complete VDM-SL specification are:

- Declaration of constants
- State definition
- Functions
- Operations

User-defined Types

- There is one more component to the specification, which must come at the beginning. This is the declaration of any user-defined types.



- User-defined type such as signal can be defined in a VDM specification.

- The **types** clause is the appropriate place to define new types.
- The Signal type is defined as follows:

```
types

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

- Here we are defining a type by **type construction**. This form of type construction allows enumerated types to be specified formally in VDM-SL.
- Values such as <INCREASE>, <DECREASE> and <DO_NOTHING> are called **quote types**, and a type such as *Signal* is a **union of quote types** in VDM.
- A quote type defines a single value, and at the same time defines a type containing just that value. These quote types correspond to the values specified in the UML diagram at previous slide.
- Byconvention, **type names begin with an upper-case letter**.

Formal Methods

Topic#066

END!

Formal Methods

Topic#067
The nil Value

The nil Value

- It is 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 [N] or [Z] – meaning that a variable of that type can take the value of nil.
- Effectively we are extending the type to include the **nil** value.

Formal Methods

Topic#067

END!

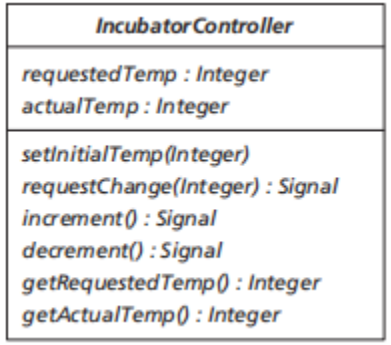
Formal Methods

Topic#068
Improving the Incubator System

Improving the Incubator System

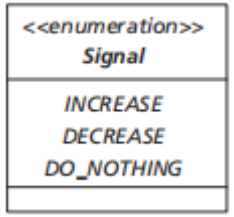
- Make the software more realistic.
- In our enhanced system, the software will not only record the current temperature of the system, but will also control the hardware.
- Our new system will also behave a bit more realistically in regard to the initial temperature of the incubator.

UML diagram for our new system



Specification of the IncubatorController

UML specification of the Signal type



UML specification of the Signal type

Formal Methods

Topic#068

END!

Formal Methods

Topic#069
Specifying the State of the IncubatorController System

Specifying the State of the IncubatorController System

There now need to be two components of the state:

- one to hold the actual temperature
- one to hold the temperature that has been requested

New state definition

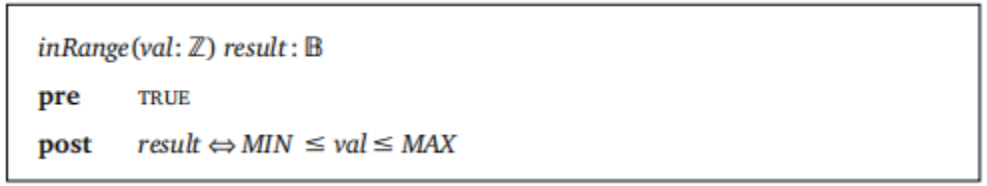
Invariant comprising two conjuncts

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

Now there are two inputs to the make function.

inRange Function

- The purpose of this function is to check whether an integer value, val, is within the range MIN and MAX as defined earlier.
- You can see that the use of the equivalence connective ensures that the output is true if the input is in range, but is otherwise false.



Use of inRange Function in the invariant

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

This makes the invariant much more readable, and also has the advantage that we can re-use our function throughout the specification.

Initialization Function

- Since both the requested temperature and the actual temperature will be undefined at the point when the system is created, these should both be set to nil. Hence the initialization function is:

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

Formal Methods

Topic#069

END!

Formal Methods

Topic#070

Specifying the Operations of the IncubatorController System

Specifying the Operations of the IncubatorController System

- It is going to be necessary to provide an operation that can be used to set the initial temperature of the system – this will be invoked by the hardware when the incubator has established a steady initial temperature.

setInitialTemp operation

- The operation is specified below:

$setInitialTemp(tempIn : \mathbb{Z})$

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

pre $inRange(tempIn) \land actualTemp = \textbf{nil}$

post $actualTemp = tempIn$

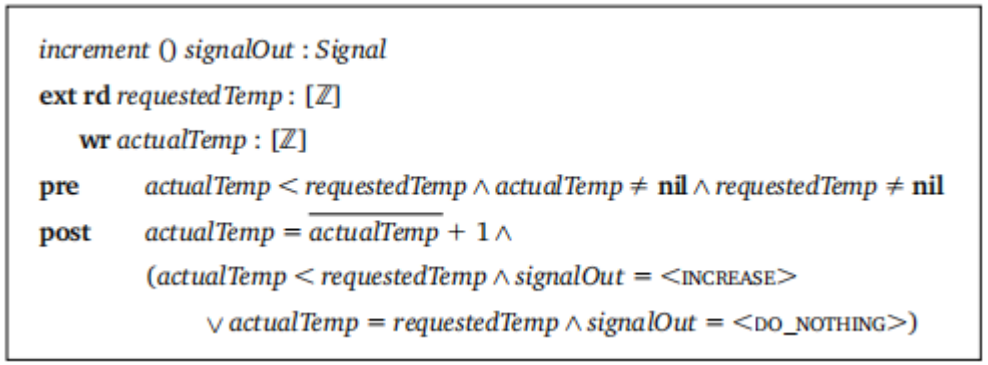
requestChange operation

- The operation is specified as follows:

$$
\begin{aligned}
&\mathit{requestChange}(\mathit{tempIn} : \mathbb{Z})\ \mathit{signalOut} : \mathit{Signal}\\
&\textbf{ext}\ \ \textbf{wr}\ \mathit{requestedTemp} : [\mathbb{Z}]\\
&\qquad \textbf{rd}\ \mathit{actualTemp} : [\mathbb{Z}]\\
&\textbf{pre}\quad \mathit{inRange}(\mathit{tempIn}) \land \mathit{actualTemp} \neq \textbf{nil}\\
&\textbf{post}\quad \mathit{requestedTemp} = \mathit{tempIn} \land\\
&\qquad\qquad (\mathit{tempIn} > \mathit{actualTemp} \land \mathit{signalOut} = \langle \textsc{increase} \rangle\\
&\qquad\qquad \lor\ \mathit{tempIn} < \mathit{actualTemp} \land \mathit{signalOut} = \langle \textsc{decrease} \rangle\\
&\qquad\qquad \lor\ \mathit{tempIn} = \mathit{actualTemp} \land \mathit{signalOut} = \langle \textsc{do\_nothing} \rangle)
\end{aligned}
$$

increment operation

- The operation is specified as follows:



- The decrement operation is similar and does not require further explanation.

Read Operations

requested temperature

getRequestedTemp() *currentRequested* : [$\mathbb{Z}$]

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

pre TRUE

post *currentRequested* = *requestedTemp*

actual temperature

getActualTemp() *currentActual* : [$\mathbb{Z}$]

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

pre TRUE

post *currentActual* = *actualTemp*

Formal Methods

Topic#070

END!

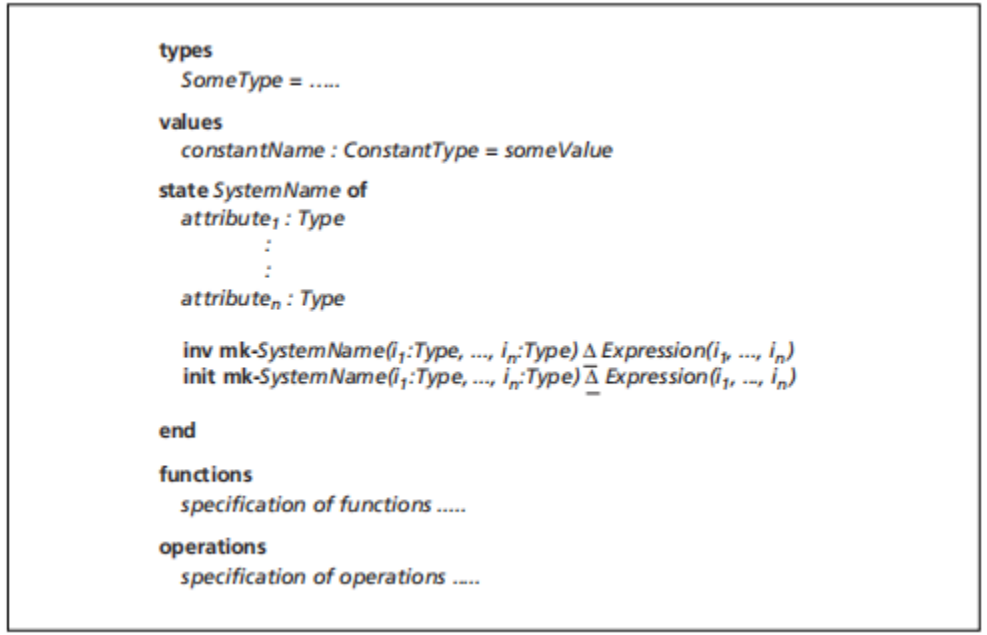
Formal Methods

Topic#071

A Standard Template for VDM-SL Specifications

Template for VDM-SL Specifications

- Generalized template for a VDM-SL specification:



- Not every clause would necessarily appear in every specification.

Formal Methods

Topic#071

END!

Formal Methods

Topic#072

Including Comments

Including Comments

- As with program code, the readability of a VDM-SL specification is greatly enhanced by the inclusion of comments. This is done by introducing the comment with the symbol – –. A new line ends the comment.

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

values
  MAX : ℤ = 10
  MIN : ℤ = −10

state IncubatorController of
  requestedTemp : [ℤ]
  actualTemp : [ℤ]
  -- both requested and actual temperatures must be in range or equal to nil
  inv mk-IncubatorController (r, a) ≜ (inRange(r) ∨ r = nil) ∧ (inRange(a) ∨
                                                              a = nil)
  -- both requested and actual temperatures are undefined when the system is initialized
  init mk-IncubatorController (r, a) ≜ r = nil ∧ a = nil
end
```

Formal Methods

Topic#072

END!

Formal Methods

Topic#073

The Complete Specification of the IncubatorController System

Complete Specification of the IncubatorController System

types
$Signal = \langle \text{INCREASE} \rangle | \langle \text{DECREASE} \rangle | \langle \text{DO\_NOTHING} \rangle$

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* $(r, a) \triangleq (inRange(r) \vee r = \textbf{nil}) \wedge (inRange(a) \vee a = \textbf{nil})$
-- both requested and actual temperatures are undefined when the system is initialized
init mk-*IncubatorController* $(r, a) \triangleq r = \textbf{nil} \wedge a = \textbf{nil}$
end

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

operations
-- an operation that records the intitial temperature of the system
$setInitialTemp(tempIn : \mathbb{Z})$
ext wr *actualTemp* : $[\mathbb{Z}]$
pre $inRange(tempIn) \wedge actualTemp = \textbf{nil}$
post $actualTemp = tempIn$
-- an operation that records the requested temperature and signals the hardware to increase
-- or decrease the temperature as appropriate
$requestChange(tempIn : \mathbb{Z})\ signalOut : Signal$
ext wr *requestedTemp* : $[\mathbb{Z}]$
rd *actualTemp* : $[\mathbb{Z}]$
pre $inRange(tempIn) \wedge actualTemp \neq \textbf{nil}$
post $requestedTemp = tempIn \wedge$
$(tempIn > actualTemp \wedge signalOut = \langle \text{INCREASE} \rangle$
$\vee\ tempIn < actualTemp \wedge signalOut = \langle \text{DECREASE} \rangle$
$\vee\ tempIn = actualTemp \wedge signalOut = \langle \text{DO\_NOTHING} \rangle)$
-- 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* : $[\mathbb{Z}]$
wr *actualTemp* : $[\mathbb{Z}]$
pre $actualTemp < requestedTemp \wedge actualTemp \neq \textbf{nil} \wedge requestedTemp \neq \textbf{nil}$
post $actualTemp = \overline{actualTemp} + 1 \wedge$
$(actualTemp < requestedTemp \wedge signalOut = \langle \text{INCREASE} \rangle$
$\vee\ actualTemp = requestedTemp \wedge signalOut = \langle \text{DO\_NOTHING} \rangle)$
-- 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 $actualTemp > requestedTemp \wedge actualTemp \neq \textbf{nil} \wedge requestedTemp \neq \textbf{nil}$
post $actualTemp = \underline{actualTemp} < 1 \wedge$
$(actualTemp > \overline{requestedTemp} \wedge signalOut = \langle \text{DECREASE} \rangle$
$\vee\ actualTemp = requestedTemp \wedge signalOut = \langle \text{DO\_NOTHING} \rangle)$

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

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

Formal Methods

Topic#073

END!

CS636 Formal Methods

Topic 10.1

Topics Covered

Sets

Sets for System Modelling

- **Many systems deal with data collections**
- ***For such data collections, VDM-SL provides a number of collection types***
  - ***Sets***
  - ***Sequences***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets for System Modelling

- **A set is an unordered collection of objects in which repetition is not significant.**
- ***A collection of objects that are considered unique, and in which ordering is unimportant, the set type is a good candidate.***
- ***A collection of patients registered on the books of a doctor's surgery.***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- **The type constructor –set is appended to the type associated with the elements of the set.**
- **aNumber: $\mathbb{N}$**
- **someNumbers: $\mathbb{N}$-set ($\mathbb{N}$ means Natural Numbers)**
- **someOtherNumbers: $\mathbb{Z}$-set**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Declaration

```
types
        Day =<MON> | <TUE> | <WED> | <THU> | <FRI> | <SAT> | <SUN>
```

```
importantDays: Day-set
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Defining sets

someNumbers = {2, 4, 28, 19, 10}
importantDays = {<FRI>, <SAT>, <SUN>}

someNumbers = {*28, 2, 10, 4, 19*}
importantDays = {<SUN>, <FRI>, <SAT>}

someNumbers = {*28, 2, 10, 2, 4, 19, 2*}
importantDays = {<SUN>, <FRI>, <SAT>, <FRI>, <FRI}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- **Subranges: used when a set of continuous integers is required**

```
someRange  = {5,…,15}
```

- ***A subrange returns the set of all numbers from the first to the last number inclusive.***

```
someRange =  {5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- **When the second number in the range is smaller than the first, the empty set is returned. The empty set is represented in VDM-SL as an empty pair of braces**

```
{7,...,6} = {}
```

```
{6,...,4} = {}
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- **Comprehension: A set is defined by means of an expression and/or a test that each element in the set must satisfy.**

$$someNumbers = \{x \mid x \in \{2,\ldots,6\} \bullet isEven(x)\}$$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- ***In general, set comprehension takes the following form:***

```
someSet  = {expression (x) | binding (x) • test(x)}
```

- ***"|" means "Such that"***
- ***"binding" & "test" determine acceptable values for the free variable (x in this case).***
- ***The bullet (•) separates the binding from the test.***
- ***This free variable is then used in the expression to determine the final value of elements in the new set.***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sets in VDM-SL

- ***expressions can be complex:***

$someOtherNumbers = \{x^2 \mid x \in \{2,\ldots,6\}\}$

$someOtherNumbers = \{2^2, 3^2, 4^2, 5^2, 6^2\}$
$= \{4, 9, 16, 25, 36\}$

- ***type can be used in the binding instead of a set. When using a type the 'is of type' symbol (:) is to be used***

$smallNumbers \ \{x \mid x: \mathbb{N} \bullet 1 \leq x \leq 10\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Finite and Infinite sets

- ***Sets having infinite elements e.g. set of all real numbers are described as infinite sets***
- ***As sets in VDM-SL need to be implemented in machine which is not possible, so in VDM-SL, sets are finite***

```
smallNumbers =  {x | x: ℤ ● x < 0}
```

- ***The above set is not permissible in VDM-SL***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

Topic 10.2

Topics Covered

Set Operations

Set Operations

- ***Three operations that take two sets as input and return a new set as output***
  - ***Set union***
  - ***Set intersection***
  - ***Set difference***
- ***In each case, the types of elements in each set are assumed to be the same.***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Union

- **The union of two sets, j and k returns a set that contains all the elements of the set j and all the elements of the set k.**

J ∪ **K** **(J Union K)**

- **Example:**

if	*j* = {<MON>, <TUE>, <WED>, <SUN>}
and	*k* = {<MON>, <FRI>, <TUE>}
then	*j* ∪ *k* = {<MON>, <TUE>, <WED>, <SUN>, <FRI>}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Intersection

- **The intersection of two sets j and k returns a set that contains all the elements that are common to both j and k.**

J ∩ **K (J intersection K)**

- **Example:**

if	*j* = {<MON>, <TUE>, <WED>, <SUN>}
and	*k* = {<MON>, <FRI>, <TUE>}
then	*j* ∩ *k* = {<MON>, <TUE>}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Difference

- **The difference of j and k is the set that contains all the elements that belong to j but do not belong to k.**

J \ **K** **(J difference K)**

- **Example:**

if	*j* = {<MON>, <TUE>, <WED>, <SUN>}
and	*k* = {<MON>, <FRI>, <TUE>}
then	*j* \ *k* = {<WED>, <SUN>}

Incorrect: {<MON>, <TUE>, <WED>} \ <TUE>	Correct: {<MON>, <TUE>, <WED>} \ {<TUE>}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Equal

- **Two sets i and j are equal if they have same elements**

  i = {a, b, c} j = {b, a, c}

- **Here x and y are not equal:**

  x = {a, b, c} j = {b, a, c, d}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations

- **A set containing just a single element is referred to as a singleton set. Example:**

$J = \{<SUN>\}$

- **Commutative operators**: same result regardless of the order of the parameters

$J \cap K = K \cap J$
$J \cup K = K \cup J$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations

- **Set difference is not commutative: the order of the parameters is significant to the result.**

$j \setminus k \neq k \setminus j$

j = {<MON>, <TUE>, <WED>, <SUN>}
k = {<MON>, <FRI>, <TUE>}
$j \setminus k$ = {<WED>, <SUN>}
$k \setminus j$ = {<FRI>}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Membership

- **This operator checks whether or not a particular element is present in a particular set.**

  Symbol: ∈

- **Non-Membership:**

  Symbol: ∉

Example:

If A = {1, 3, 5, 7}, then 1 ∈ A, but 2 ∉ A

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Subsets

- **It returns TRUE if all the elements in the first set are also elements of the second set and FALSE otherwise**
  - **Example:**

    $\{a, d, e\} \subseteq \{a, b, c, d, e, f\}$ ... ***TRUE***

    $\{a, b, c, d, e, f\} \subseteq \{a, d, e\}$ ... ***FALSE***

  - **Returns TRUE if both sets are equal**

    $\{a, d, e\} \subseteq \{d, a, e\}$ ... ***TRUE***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Subsets

- **It returns TRUE if all the elements in the first set are also elements of the second set and FALSE otherwise**
  - **Example:**

    $\{a, d, e\} \subseteq \{a, b, c, d, e, f\}$ ... ***TRUE***

    $\{a, b, c, d, e, f\} \subseteq \{a, d, e\}$ ... ***FALSE***

  - **Returns TRUE if both sets are equal**

    $\{a, d, e\} \subseteq \{d, a, e\}$ ... ***TRUE***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Set Operations: Cardinality

- **The cardinality operator of VDM-SL (card) returns the number of elements in a given set.**

```
card {7, 2, 12}  = 3
card {4,...,10}  = 7
card { }  = 0
```

- **As repetition is not significant in sets, repeated elements are counted only once when calculating the cardinality**

```
card {7, 2, 12, 2, 2}  =card {7, 2, 12}  = 3
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

Topic 10.3

Topics Covered

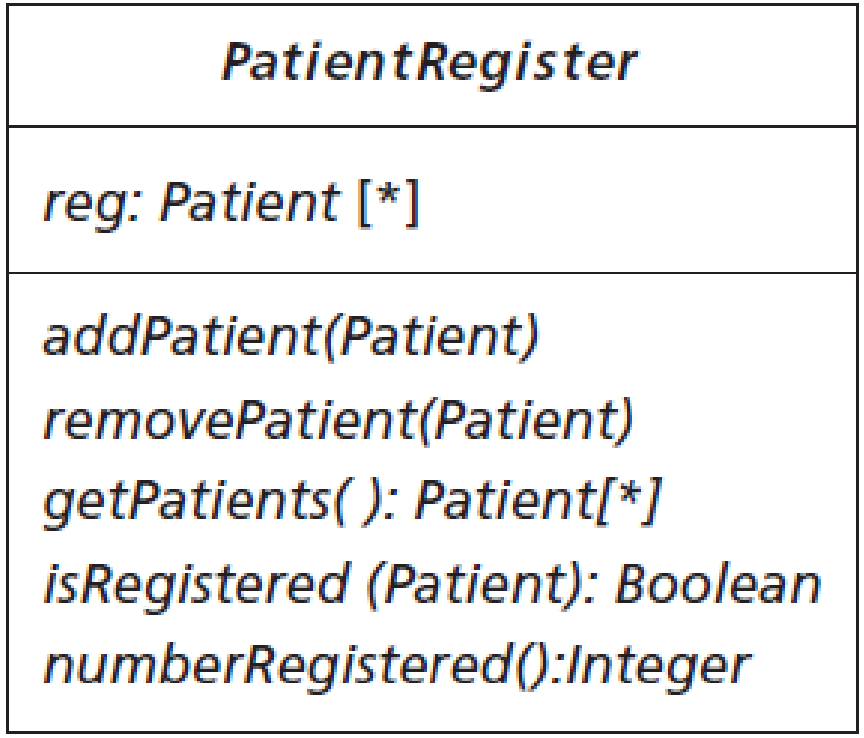
Patient Register Class

Patient Surgery Clinic

- **At maximum 200 patients can register**
- ***Patients can be added/Removed***
- ***List of Patients and Number of Patients should be returned***
- ***Check if a patient is registered or not***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

UML representation



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
Patient TOKEN

values
LIMIT: ℕ = 200

state *PatientRegister* **of**
reg: *Patient*-**set**

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
Patient = TOKEN

values
LIMIT: ℕ = 200

state *PatientRegister* **of**
reg: *Patient*-**set**

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

inv mk-*PatientRegister* (*r*) $\triangleq$ **card** $r \leq LIMIT$

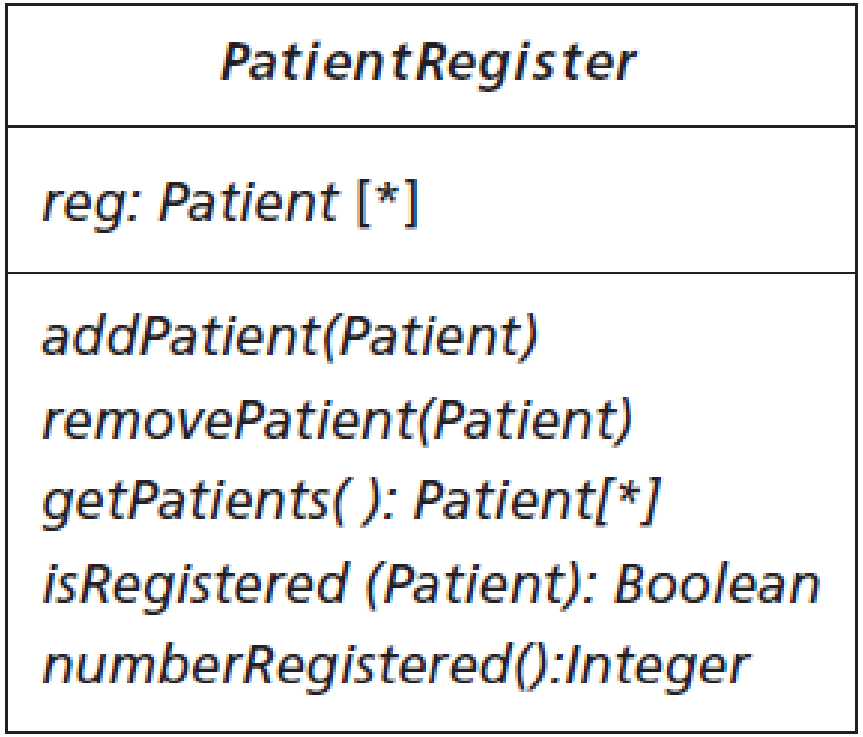
inv mk-*PatientRegister* (*r*) $\triangleq$ $r = \{\}$

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
Patient = TOKEN
values=
LIMIT: ℕ = 200
state *PatientRegister* **of**
reg: *Patient*-**set**
inv mk-*PatientRegister* (*r*) $\triangleq$ **card** *r* $\leq$ *LIMIT*
inv mk-*PatientRegister* (*r*) $\triangleq$ *r* = {}
end



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

addPatient (patientIn: Patient)
ext wr reg: *Patient*-**set**
pre *patientIn* $\notin$ *reg* $\wedge$ **card** *reg* $<$ *LIMIT*
post $reg = \overline{reg} \cup \{patientIn\}$

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

removePatient (*patientIn*: *Patient*)
ext wr *reg*: *Patient*-**set**
pre *patientIn* ∈ *reg*
post *reg* = $\overline{reg}$ \ {*patientIn*}

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

getPatients ( ) *output*: *Patient*-**set**
ext rd *reg*: *Patient*-**set**
pre TRUE
post *output* = *reg*

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

isRegistered (*patientIn*: *Patient*) *query*: $\mathbb{B}$
ext rd *reg*: *Patient*-**set**
pre TRUE
post *query* $\Leftrightarrow$ *patientIn* $\in$ *reg*

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

numberRegistered ( ) *total*: ℕ
ext rd *reg*: *Patient*-**set**
pre TRUE
post *total* = **card** *reg*

PatientRegister
reg: Patient [*]
addPatient(Patient) *removePatient(Patient)* *getPatients( ): Patient[*]* *isRegistered (Patient): Boolean* *numberRegistered():Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

Topic 10.4

Topics Covered

The Airport Class and Implementation in VDM-SL

Airport Class

- **Consider a system that keeps track of aircraft that are allowed to land at a particular airport. Aircraft must apply for permission to land at the airport prior to landing. When an aircraft arrives to land at the airport it should only have done so if it had previously been given permission. When an aircraft leaves the airport its permission to land is also removed.**

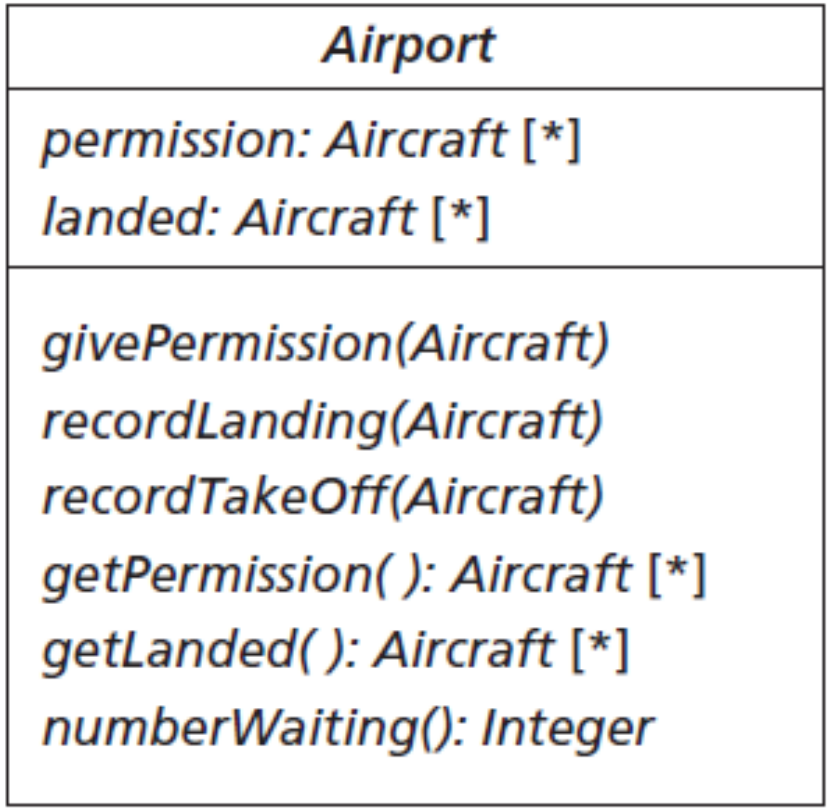
Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Airport Class

- **givePermission: records the fact that an aircraft has been granted permission to land at the airport.**
- **recordLanding: records an aircraft as having landed at the airport.**
- **recordTakeOff: records an aircraft as having taken off from the airport.**
- **getPermission: returns the aircrafts currently recorded as having permission to land**
- **getLanded: returns the aircrafts currently recorded as having landed**
- **numberWaiting: returns the number of aircrafts granted permission to land but not yet landed.**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

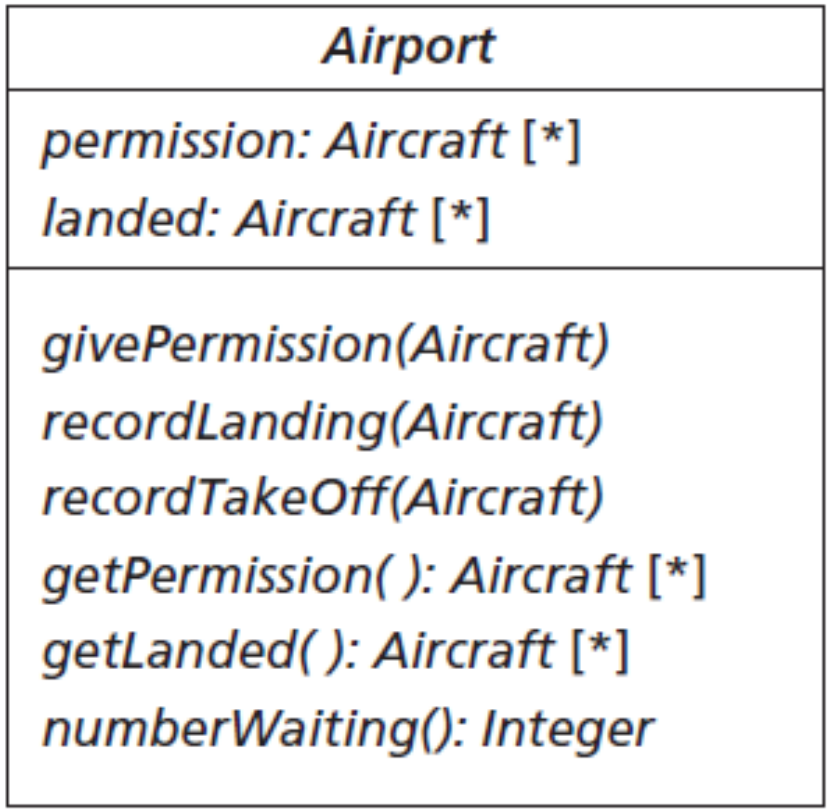
UML representation



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

```
types
Aircraft = TOKEN
state Airport of
    permission: Aircraft-set
    landed: Aircraft -set
inv mk-Airport(p, l) △ l ⊆ p
init mk-Airport (p, l) △ p = { } ∧ l = { }
end
```



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

givePermission (*craftIn*: *Aircraft*)
ext wr *permission*: *Aircraft* **-set**
pre *craftIn* ∉ *permission*
post *permission* = $\overline{permission}$ ∪ {*craftIn*}

Airport
permission: Aircraft [*] *landed: Aircraft* [*]
givePermission(Aircraft) *recordLanding(Aircraft)* *recordTakeOff(Aircraft)* *getPermission( ): Aircraft* [*] *getLanded( ): Aircraft* [*] *numberWaiting(): Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

recordLanding (*craftIn*: *Aircraft*)
ext rd *permission*: *Aircraft* **-set**
wr *landed*: *Aircraft* **-set**
pre $craftIn \in permission \wedge craftIn \notin landed$
post $landed = \overline{landed} \cup \{craftIn\}$

Airport
permission: Aircraft [*] *landed: Aircraft* [*]
givePermission(Aircraft) *recordLanding(Aircraft)* *recordTakeOff(Aircraft)* *getPermission( ): Aircraft* [*] *getLanded( ): Aircraft* [*] *numberWaiting(): Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

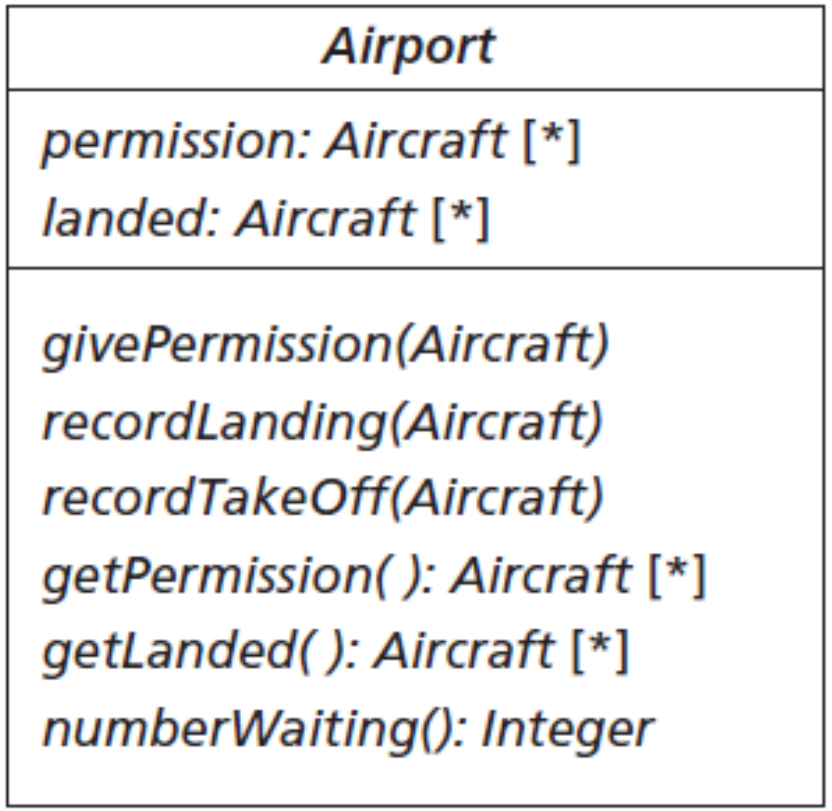
recordTakeOff (*craftIn*: *Aircraft*)
ext wr *permission*: *Aircraft* **-set**
wr *landed*: *Aircraft* **-set**
pre *craftIn* ∈ *landed*
post *landed* = $\overline{landed}$ \ {*craftIn*} ∧ *permission* = $\overline{permission}$ \ {*craftIn*}

Airport
permission: Aircraft [*] *landed: Aircraft* [*]
givePermission(Aircraft) *recordLanding(Aircraft)* *recordTakeOff(Aircraft)* *getPermission( ): Aircraft* [*] *getLanded( ): Aircraft* [*] *numberWaiting(): Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

getPermission ( ) *out*: *Aircraft* **-set**
ext rd *permission*: *Aircraft* **-set**
pre TRUE
post *out = permission*



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

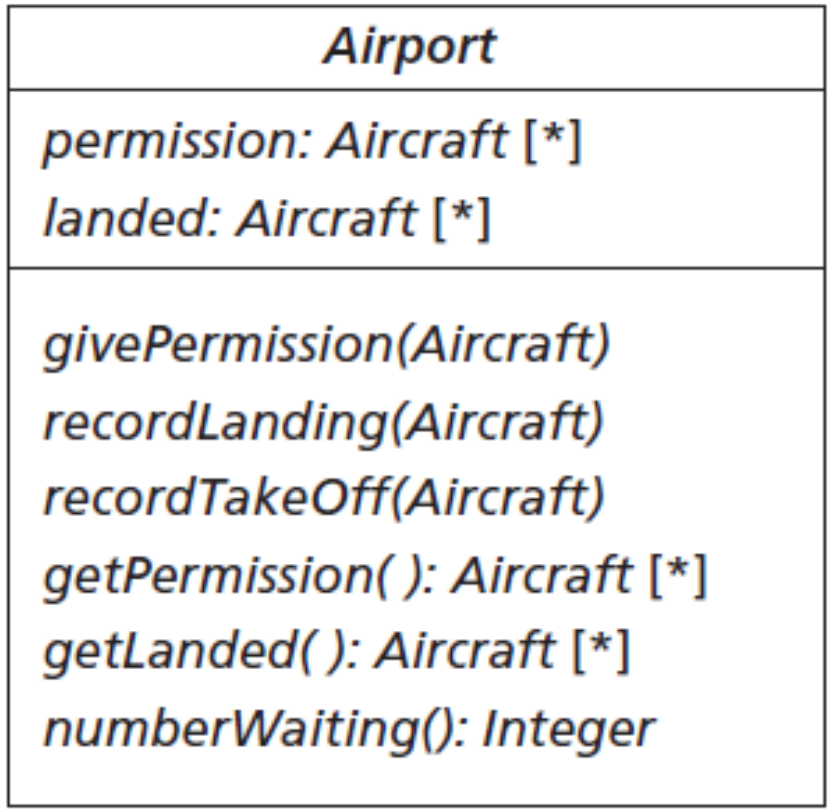
getLanded ( ) *out*: *Aircraft* **-set**
ext rd *landed*: *Aircraft* **-set**
pre TRUE
post *out* = *landed*

Airport
permission: Aircraft [*] *landed: Aircraft* [*]
givePermission(Aircraft) *recordLanding(Aircraft)* *recordTakeOff(Aircraft)* *getPermission( ): Aircraft* [*] *getLanded( ): Aircraft* [*] *numberWaiting(): Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

numberWaiting( ) *total*: ℕ
ext rd *permission*: *Aircraft* **-set**
rd *landed*: *Aircraft* **-set**
post *total* = **card** (*permission* \ *landed*)



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

11.1

Topics Covered

Sequences

Sequences

- **Sequence is a collection of objects, however**
  - ***A sequence is an ordered collection of objects.***
  - ***In a sequence, repetitions are significant.***
- ***Notation:***
  - ***A sequence is specified by enclosing its members in square brackets.***

  s = [a, d, f, a, d, d, c]

  queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequences

- **As sequence is an ordered collection, so:**
  - **[a, d, f] ≠ [a, f, d]**
- **Empty Sequence is expressed as:**
  - **[ ]**
- **The elements of a sequence are numbered, starting from 1, from left to right. We can refer to a particular element of a sequence by placing the position of the element in brackets**
- **s(3)=f queue(4) = WINSTON**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

s = [a, d, f, a, d, d, c]

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

- **The len operator gives us the length of the sequence.**
  - ***len s = 7***
  - ***len queue = 5***

- **The elems operator returns a set that contains all the members of the sequence (removes the duplicates):**
  - ***elems s = {a, d, f, c}***
  - ***elems queue = {MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH}***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

s = [a, d, f, a, d, d, c]

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

- **The head (hd) operator gives us the first element in the sequence;**
- **The tail (tl) operator gives us a sequence containing all but the first element**
  - ***hd s = s(1) = a***
  - ***tl s = [d, f, a, d, d, c]***
  - ***hd queue = MICHAEL***
  - ***tl queue = [VARINDER, ELIZABETH, WINSTON, JUDITH]***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

- **The concatenation operator (^) operates on two sequences, and returns a sequence that consists of the two sequences joined together**

```
if      first = [ w, e, r, w ]
   and second = [ t, w, q ]
   then
   first^second = [ w, e, r, w, t, w, q ]
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

- **The override operator, †, takes a sequence and gives us a new sequence with a particular element of the old sequence overridden by a new element.**
- **The generalized form is “s † m”**
  - **s is a sequence and m is a map**
  - **[a, c, d, e] † {2 -> x, 4 -> y} = [a, x, d, y]**
- **The override operator is undefined if any index is invalid.**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

s = [a, d, f, a, d, d, c]

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

- **The inds operator returns a set of all the indices of the sequence.**
  - ***inds s = {1, 2, 3, 4, 5 , 6, 7}***
  - ***inds queue = {1, 2, 3, 4, 5}***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

s = [a, d, f, a, d, d, c]

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

- **A subsequence operator is defined to allow us to extract a part of a sequence between two indices.**
  - ***subseq(s, 2, 5) = [d, f, a, d]***
- ***The language allows us to write this in the following, more convenient, way***
  - ***s(2, … , 5) = [d, f, a, d]***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequence Operators

s = [a, d, f, a, d, d, c]

queue = [MICHAEL, VARINDER, ELIZABETH, WINSTON, JUDITH]

- **The subsequence operator is undefined if either index is out of range, or if the first index is greater than the second**
  - ***s(1, … ,0) = []***
  - ***s(8, … , 7) = []***
  - ***s(2, … , 2) = [d]***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

11.2

Topics Covered

Defining a Sequence by Comprehension

Defining a Sequence by Comprehension

- **We can define a sequence by comprehension**
- **sequence of odd numbers from 1 to 20**
  - **[a | a ∈ {1, … , 20} ● is-odd(a)]**
  - ***is-odd* is a function that returns TRUE if a is odd and FALSE if a is even**
- **Generic Form:**
  - **[expression(a) | a ∈ SomeSet ● test (a)]**
- **When constructing the sequence, these values are considered in order, smallest first.**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Defining a Sequence by Comprehension

- **Often sequence comprehension is used to ‘filter’ a sequence.**
  - **s1 = [2, 3, 4, 7, 9, 11, 6, 7, 8, 14, 39, 45, 3]**
  - **s2 = [s1(i) | i ∈ inds s1 ● s1(i) > 10]**
  - **s2 would evaluate to the sequence [11, 14, 39, 45].**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

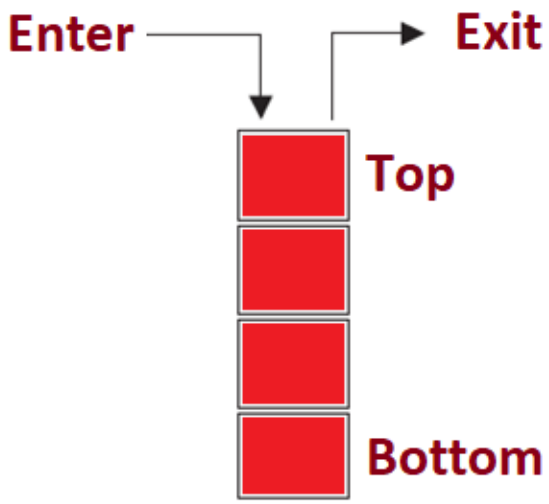
Sequence Type in VDM-SL

- **Set is declared by appending the word –set at the end of the type contained in set**
- **To declare a variable to be of type sequence we place an asterisk after the name of the type contained within the sequence.**
  - **seq : $\mathbb{Z}$***
  - **convoy : SpaceCraft***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Applications of Sequence: Stack

- **A stack is a data structure, an ordered list that follows last-in-first-out (LIFO) protocol**

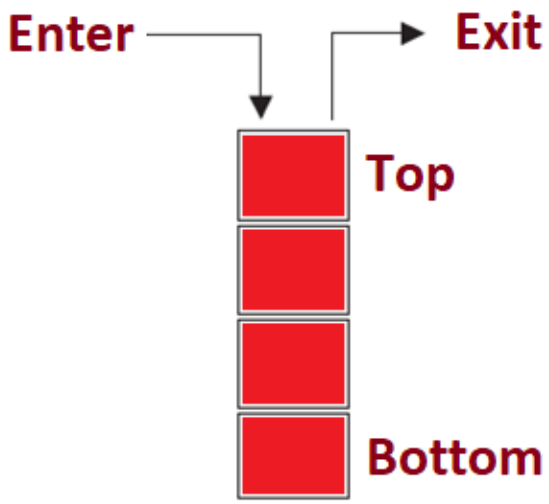


Stack
stack: Element[*]
push(Element) *pop(): Element* *isEmpty(): Boolean*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Applications of Sequence: Stack

- **A stack is a data structure, an ordered list that follows last-in-first-out (LIFO) protocol**



Stack
stack: Element[*]
push(Element) *pop(): Element* *isEmpty(): Boolean*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Specifying the State of the Stack

```
types
        Element = TOKEN
```

```
state Stack of
    stack : Element*
    init mk-Stack(s) ≜ s = []
end
```

Stack
stack: Element[*]
push(Element) *pop(): Element* *isEmpty(): Boolean*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Specifying the Operations on the Stack

push(itemIn : Element)
ext wr *stack : Element**
pre *TRUE*
post *stack = [itemIn] ^* $\overline{stack}$

Stack
stack: Element[*]
push(Element) *pop(): Element* *isEmpty(): Boolean*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Specifying the Operations on the Stack

pop() itemRemoved : Element
ext wr *stack : Element**
pre *stack ≠ []*
post *stack =* ***tl*** *$\overline{stack}$ ^ itemRemoved =* ***hd*** *$\overline{stack}$*

isEmpty() query : $\mathbb{B}$
ext rd *stack* : *Element**
pre TRUE
post *query⇔stack* = []

Stack
stack: Element[*]
push(Element) *pop(): Element* *isEmpty(): Boolean*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

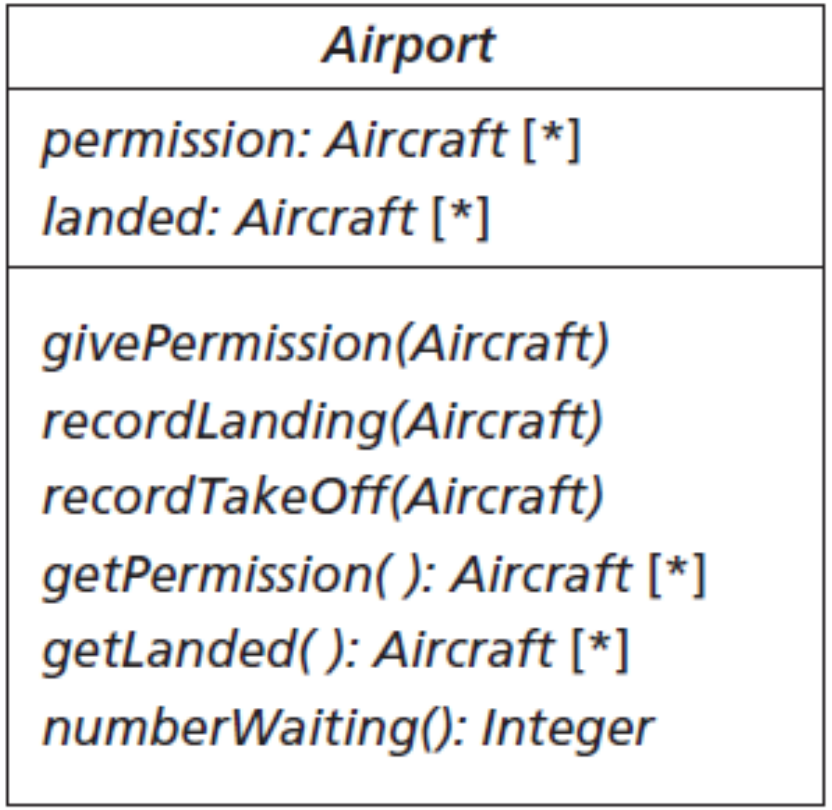
CS636 Formal Methods

11.3

Topics Covered

Rethinking the Airport System

UML representation



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
Aircraft = TOKEN
state *Airport2* **of**
permission: *Aircraft*-**set**
landed: *Aircraft* **–set**
*circling : Aircraft**
init mk-*Airport2* (*p, l, c*) $\triangleq$ *p* = { } ∧ *l* = { } ∧ *c* = []
end

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

isUnique(seqIn : Aircraft) query : $\mathbb{B}$*
pre TRUE
post query$\Leftrightarrow\forall i_1, i_2 \in$ **inds** seqIn $\bullet$ $i_1 \neq i_2 \Rightarrow$seqIn$(i_1) \neq$ seqIn(i_2)

inv mk-Airport2 (p,l,c) $\triangleq$ *l* $\subseteq$ *p*
$\wedge$ *elems c* $\subseteq$ *p*
$\wedge$ *elems c* $\cap$ *l* = { }
$\wedge$ *isUnique(c)*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

allowToCircle (craftIn : Aircraft)
ext wr *circling : Aircraft**
rd *permission : Aircraft-set*
rd *landed : Aircraft-set*
pre *craftIn ∈ permission ∧ craftIn ∉* ***elems*** *circling ∧ craftIn ∉ landed*
post *circling =* $\overline{circling}$ *^ [craftIn]*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

recordLanding( )
ext wr *circling : Aircraft**
wr *landed : Aircraft-set*
pre *circling* ≠ []
post $landed = \overline{landded} \cup \{ \textbf{hd } \overline{circling} \} \wedge circling = \textbf{tl } \overline{circling}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

11.4

Topics Covered

Sequences: Some useful functions

Sequences: Some useful functions

- **Sometimes it is useful to have a function that returns the last element in a sequence**
- **Also sometimes you need a function that returns the sequence with the last element removed.**
- ***These are not standard VDM-SL functions, so here we will see description of few of these functions***
- ***You can call them custom build functions and can include anywhere you need in your VDM-SL specification***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequences: Some useful functions

last(*sequenceIn* : *Element**) *elementOut* : *Element*
pre *sequenceIn* ≠ []
post *elementOut* = *sequenceIn*(**len** *sequenceIn*)

allButLast(*sequenceIn* : *Element**) *sequenceOut* : *Element**
pre *sequenceIn* ≠ []
post *sequenceOut* *sequenceIn*(1, … , (**len** *sequenceIn* - 1))

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Sequences: Some useful functions

find(*sequenceIn* : *Element**, *element* : *Element*) *position* : $\mathbb{N}$
pre *element* $\in$ **elems** *sequenceIn*
post *sequenceIn*(*position*) = *element*

findFirst(*sequenceIn* : *Element**, *element* : *Element*) *position* : $\mathbb{N}$
pre *element* $\in$ *sequenceIn*
post *sequenceIn*(*position*) = *element* $\wedge$ $\forall i \in$ **inds** *sequenceIn* $\bullet$ *sequenceIn (i)* = *element* $\Rightarrow$ *position* $\leq i$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

12.1

Topics Covered

Composite objects

Composite Objects

- **There will be occasions when you need to associate more than one type with an object.**
- ***e.g. the "Car" object => registration number, make, price etc.***
- ***The appropriate type for the object as a whole would then be a composite of all the types of its internal data.***
- ***We call such a type a composite object type***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

composite type definition

```
TypeName :: fieldname1 : Type1
            fieldname2 : Type2
            :
```

'::'* symbol is called *"Composed of"

Individual values are called fields

Example:

```
Time:: hour: ℕ
       minute: ℕ
       second: ℕ
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Composite Object Operators

- **make function: creates new object of given composite type**
  - ***mk-CompositeObjectTypeName (parameter list)***

inv mk-Airport2 (p,l,c) $\triangleq$ *l* $\subseteq$ *p* $\wedge$ *elems c* $\subseteq$ *p* $\wedge$ *elems c* $\cap$ *l* = { } $\wedge$ *isUnique(c)*

mk-*Airport2*: *Aircraft*-**set** x *Aircraft*-**set** x *Aircraft** $\rightarrow$*Airport2*

mk-*Time*: $\mathbb{N}$ **x** $\mathbb{N}$ **x** $\mathbb{N}$ $\rightarrow$ **Time**

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Composite Object Operators

- **How to use mk-Time function**

```
someTime = mk-Time (16, 20, 44)
```

- ***How to specify the invariants***

```
Time:: hour: ℕ
       minute: ℕ
       second: ℕ
inv mk-Time (h, m, s) ≜ h < 24 ∧ m < 60 ∧ s <  60
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Composite Object Operators

- **Individual fields of composite objects are selected (read) by using dot operator '.' followed by the name of the field.**

```
someTime.minute = 20
someTime.hour = 16
```

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Composite Object Operators

- **mu (μ) function: returns one composite object from another but with one or more fields changed.**

newTime= μ(someTime, hour ↦ 15)

thisTime = μ(someTime, minute ↦ 0, second ↦ 0)

thisTime = mk-Time(someTime.hour,0,0)

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

12.2

Topics Covered

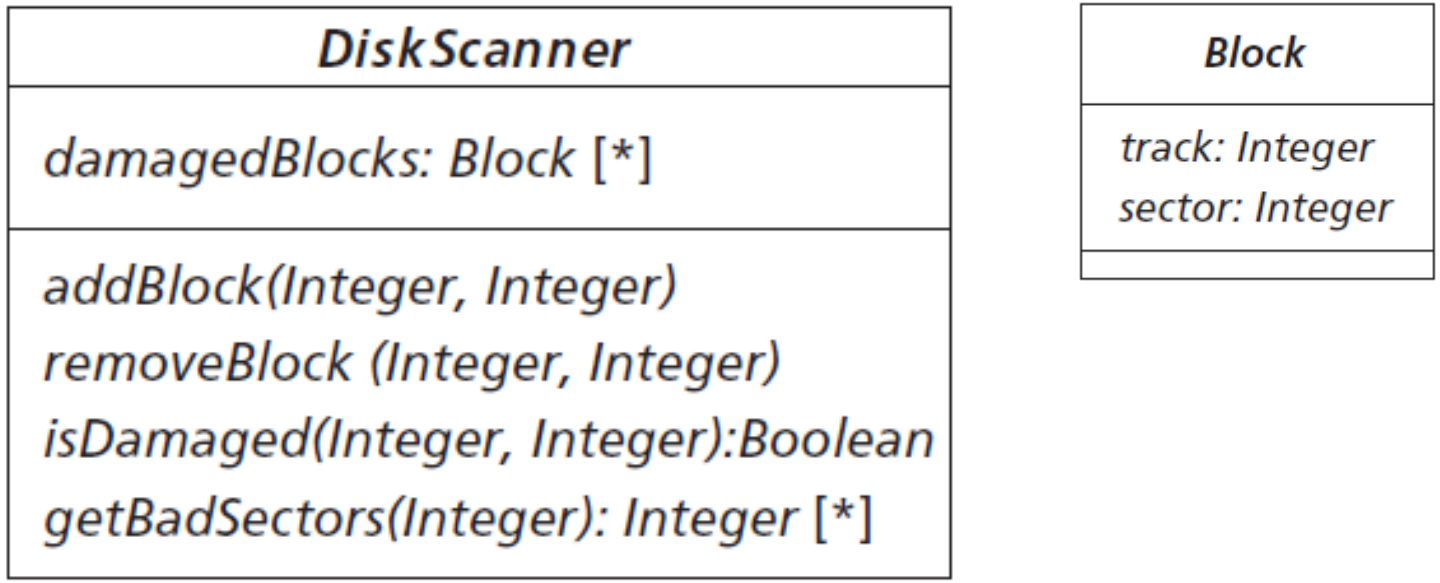
DiskScanner System

DiskScanner System

- **software designed to keep track of damaged blocks on the surface of a disk.**
- ***A disk is divided into a number of tracks***
- ***each track into a number of sectors.***
- ***block = track + sector number***
- ***DiskScanner class***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

UML representation



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
Block : : *track*: $\mathbb{N}$
sector: $\mathbb{N}$

state *DiskScanner* **of**
damagedBlocks: Block-**set**

init mk-*DiskScanner* (*dB*) $\triangleq$ *dB* = { }

Block
track: Integer *sector: Integer*

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

addBlock **(trackIn:** $\mathbb{N}$ **, sectorIn:** $\mathbb{N}$)
ext wr damagedBlocks: Block-**set**
pre mk-Block (trackIn, sectorIn) $\notin$ damagedBlocks
post damagedBlocks =damagedBlocks $\cup$ {**mk**-Block (trackIn, sectorIn)}

*removeBlock***(trackIn:** $\mathbb{N}$ **, sectorIn:** $\mathbb{N}$)
ext wr damagedBlocks: Block-**set**
pre mk-Block (trackIn, sectorIn) $\in$ damagedBlocks
post damagedBlocks =damagedBlocks \ {**mk**-Block (trackIn, sectorIn)}

DiskScanner
damagedBlocks: Block [*]
addBlock(Integer, Integer) *removeBlock (Integer, Integer)* *isDamaged(Integer, Integer):Boolean* *getBadSectors(Integer): Integer* [*]

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

isDamaged (*trackIn*:ℕ, *sectorIn*:ℕ) *query*: 𝔹
ext rd *damagedBlocks*: *Block*-**set**
pre TRUE
post *query*⇔**mk**-*Block* (*trackIn, sectorIn*) ∈ *damagedBlocks*

getBadSectors (*trackIn*:ℕ) *list*: ℕ-**set**
ext rd *damagedBlocks*: *Block*-**set**
pre TRUE
post *list* = {*b.sector* | *b* ∈ *damagedBlocks* ● *b.track* = *trackIn*}

DiskScanner
damagedBlocks: Block [*]
addBlock(Integer, Integer) *removeBlock (Integer, Integer)* *isDamaged(Integer, Integer):Boolean* *getBadSectors(Integer): Integer* [*]

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

12.3

Topics Covered

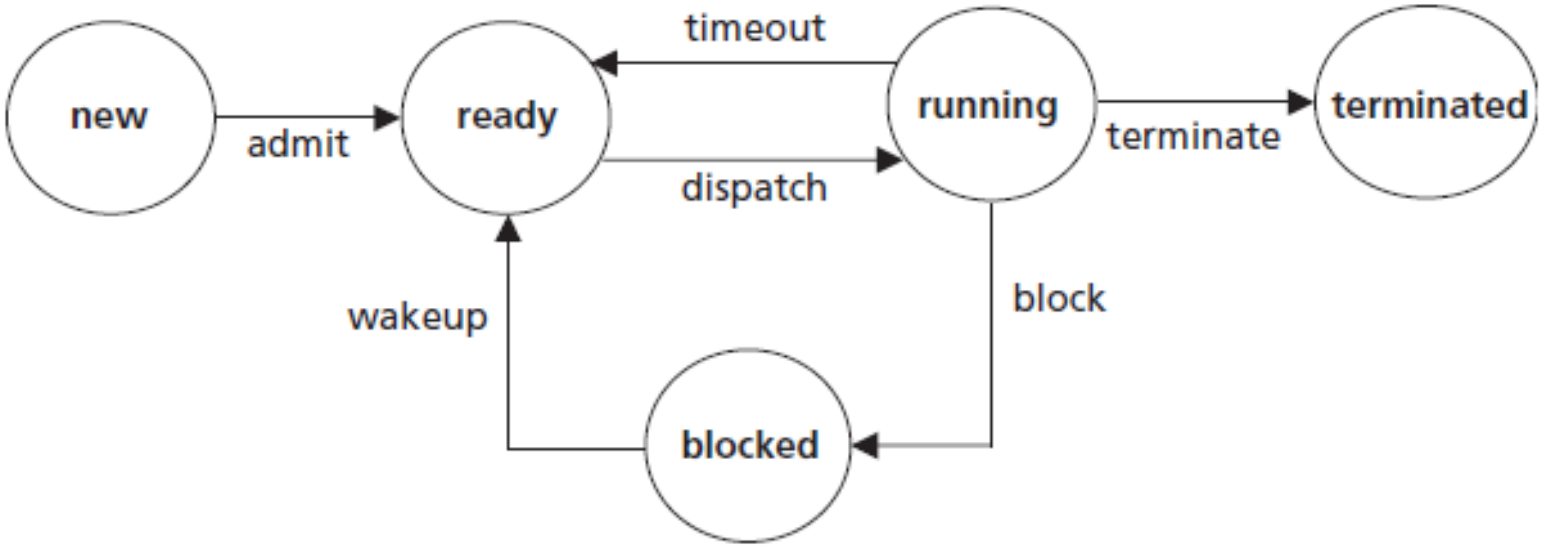
A Process Management System

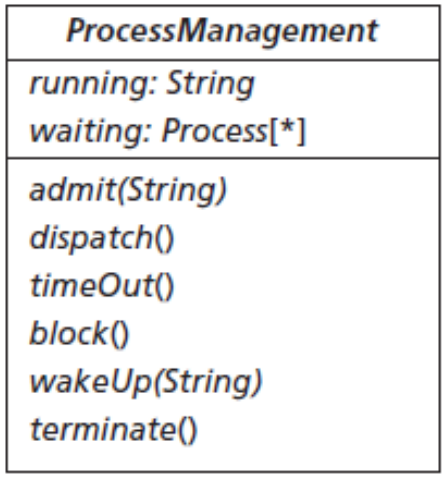
Process Management System

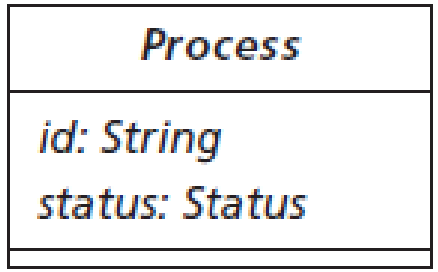
- **Process management system for a multitasking operating system.**
- ***Processes are identified by a unique process identification number (pid).***
- ***We will specify a simple first-in-first-out policy***

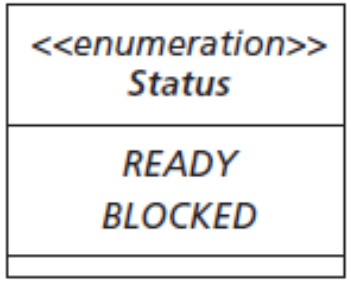
Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

UML representation





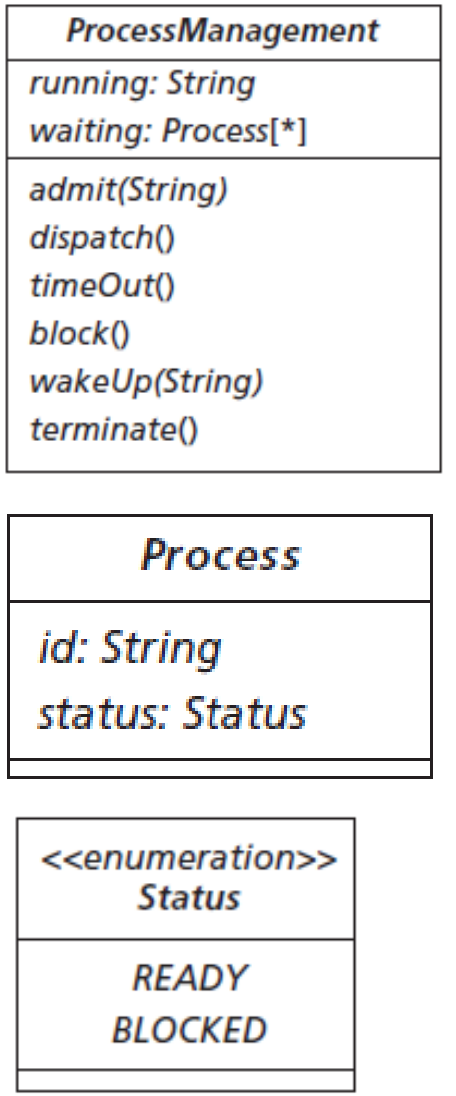




Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

types
String = Char*
Status = <READY> | <BLOCKED>
Process :: *id* : *String*
status : *Status*
state *ProcessManagement* **of**
running : [*String*]
waiting : *Process**
inv mk-*ProcessManagement* (*run, wait*) $\triangleq$ (*run* = **nil** $\vee$ $\neg\exists i \in$**inds** *wait* $\bullet$ *wait*(*i*).*id* = *run*) $\wedge$ $\forall$ *i, j* $\in$ **inds** *wait* $\bullet$ *i* $\neq$ *j* $\Longrightarrow$ *wait*(*i*).*id* $\neq$ *wait*(*j*).*id*
init mk-*ProcessManagement* (*run, wait*) $\triangleq$ *run* = **nil** $\wedge$ *wait* = [ ]
end



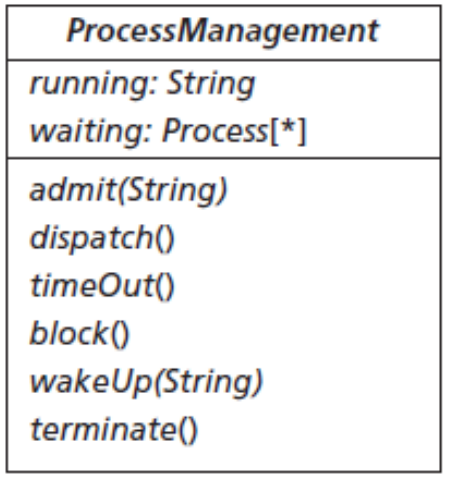
Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

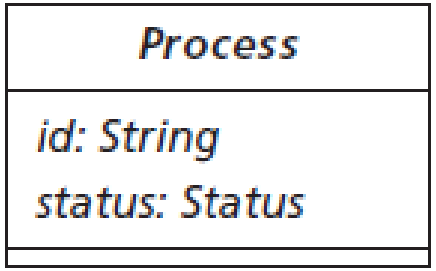
Some Useful Functions

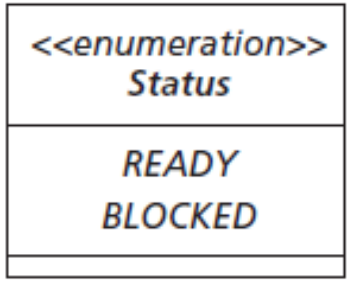
findPos(*qIn* : *Process**, *idIn* : *String*) *pos* : ℕ
pre ∃*p* ∈ **elems** *qIn* ● *p.id* = *idIn*
post *qIn*(*pos*).*id* = *idIn*

findNext(*qIn* : *Process**) *pos* : ℕ
pre ∃*p* ∈ **elems** *qIn* ● *p.status* = <*READY*>
post *qIn*(*pos*).*status* = <*READY*> ∧ ¬∃*i* ∈ {1,...,pos-1} ● qIn(i).status = <READY>

remove(*qIn* : *Process**, posIn: ℕ) *qOut* : *Process**
pre *posIn* ∈ **inds** *qIn*
post *qOut* = *qIn*(1,..., *posIn*-1) ∧ *qIn*(*posIn* + 1,..., **len** *qIn*)







Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

12.4

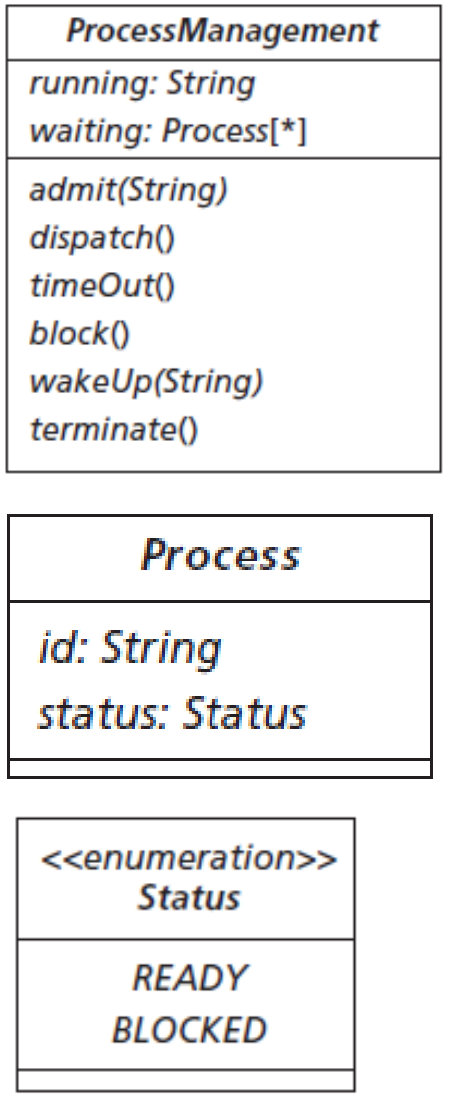
Topics Covered

A Process Management System: Modeling operations in VDM-SL

Implementation in VDM-SL

admit(*idIn*: *String*)
ext wr *waiting*: *Process**
rd *running*: [*String*]
pre (*running* = **nil** ∨ *idIn* ≠ *running*) ∧ ∀*p* ∈ **elems** *waiting* ● *p.id* ≠ *idIn*
post *waiting* = $\overline{waiting}$^[**mk**-*Process*(*idIn*, <READY>)]

dispatch( )
ext wr *running*: [*String*]
wr *waiting*: *Process*∗
pre *running* = **nil** ∧ ∃*p* ∈ **elems** *waiting* ● *p.status*=<READY>
post *running* = $\overline{waiting}$ (*findNext*($\overline{waiting}$)).*id*
∧ *waiting* = *remove*($\overline{waiting}$, *findNext*($\overline{waiting}$))

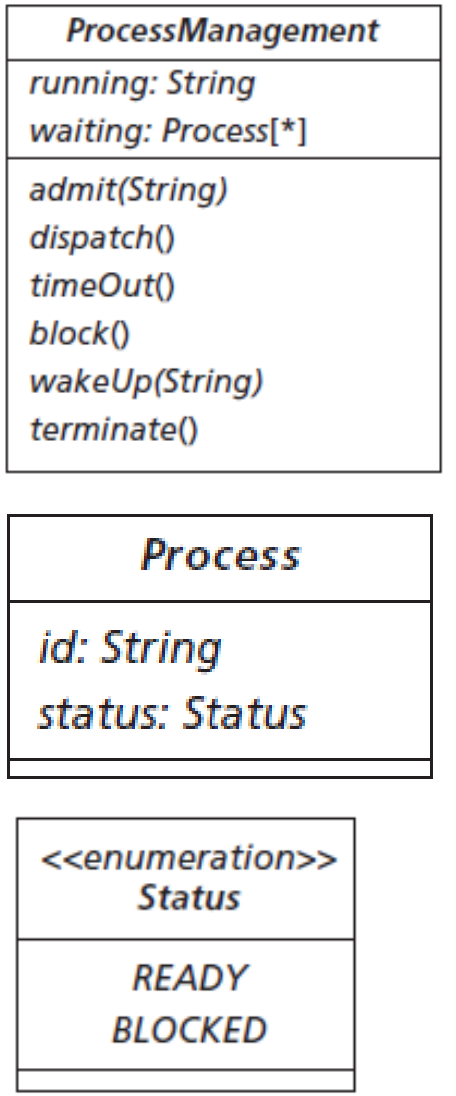


Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

timeOut( )
ext wr *running*: [*String*]
wr *waiting*: *Process*∗
pre *running* ≠ **nil**
post *waiting* = $\overline{waiting}$ ∧ [**mk**–*Process*($\overline{running}$, <READY>)] ∧ *running* = **nil**

block( )
ext wr *running*: [*String*]
wr *waiting*: *Process*∗
pre *running* ≠ **nil**
post *waiting* = $\overline{waiting}$ ∧ [**mk**–*Process*($\overline{running}$, <BLOCKED>)] ∧ *running* = **nil**

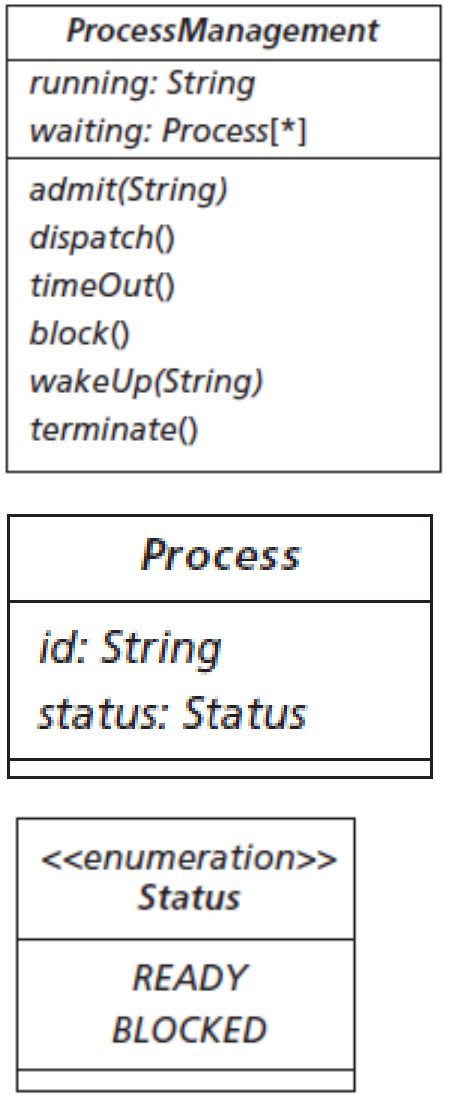


Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Implementation in VDM-SL

wakeUp(*idIn*: *String*)
ext wr *waiting*: *Process*∗
pre *waiting*(*findPos*(*waiting, idIn*)).*status* = <BLOCKED>
post *waiting* = $\overline{waiting}$ † {*findPos*($\overline{waiting}$, *idIn*) ↦ **mk**–*Process*(*idIn*, <READY>)}

terminate(( )
ext wr *running*: [*String*]
pre *running* ≠ **nil**
post *running* = **nil**



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

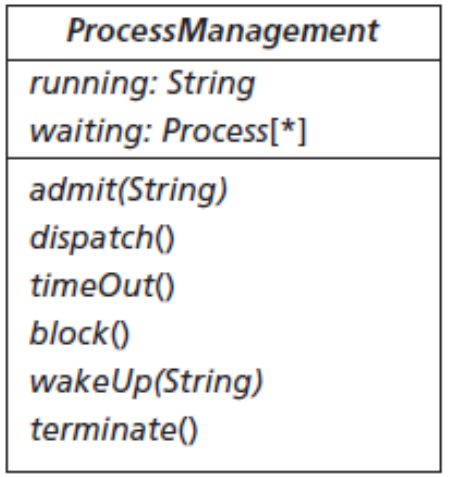
THE LET… IN CLAUSE

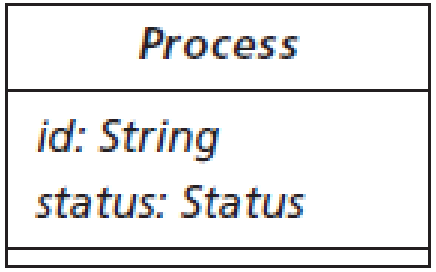
let *name* = *sub-expression*
in *expression*(*name*)

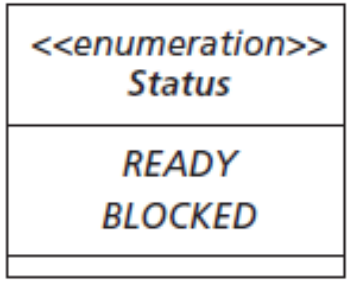
Example

post $running = \overline{waiting}\ (findNext(\overline{waiting})).id$
$\wedge\ waiting = remove(\overline{waiting}, findNext(\overline{waiting}))$

post **let** $next = findNext\ (\overline{waiting})$
in $running = \overline{waiting}(next).id$
$\wedge\ waiting = remove(\overline{waiting}, next)$







Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

THE LET… IN CLAUSE

- **Example**

pre *waiting*(*findPos*(*waiting, idIn*)).*status* = <BLOCKED>

pre let *pos* = *findPos*(*waiting, idIn*)
in *waiting*(*pos*).*status* = BLOCKED

post *waiting* = $\overline{waiting}$ † {*findPos*($\overline{waiting}$, *idIn*)
↦ **mk**–*Process*(*idIn*, <READY>)}

post let *pos* = *findPos*($\overline{waiting}$, *idIn*)
in *waiting* = $\overline{waiting}$ † {*pos* ↦ **mk**-*Process*(*idIn*, <READY>)}

post *waiting* = $\overline{waiting}$ † {*findPos*($\overline{waiting}$, *idIn*)
↦ **mk**–*Process*(*idIn*, <READY>)}

post let *pos* = *findPos*($\overline{waiting}$, *idIn*)
in let *wakeProcess* = **mk**-*Process*(*idIn*, <READY>)
In waiting = $\overline{waiting}$ † {*pos* ↦ *wakeProcess*}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

Topic 13.1

Topics Covered

Maps

Maps

- **Computing systems often involve relating two types of value together**
- ***A map is a special sort of set, which contains a set of maplets.***
- ***Each maplet connects an element of one set to an element of another set***
  - ***The first set is referred to as the domain***
  - ***The second is referred to as the range.***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Maps

Sensor	Condition
A	LOW
B	NORMAL
C	NORMAL
D	HIGH
E	NORMAL
F	NORMAL

$sensors = \{A \longmapsto \langle LOW \rangle, B \longmapsto \langle NORMAL \rangle, C \longmapsto \langle NORMAL \rangle, D \longmapsto \langle HIGH \rangle, E \longmapsto \langle NORMAL \rangle, F \longmapsto \langle NORMAL \rangle\}$

$m = \{a \longmapsto y, b \longmapsto x, C \longmapsto x, d \longmapsto z\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Maps

- **By definition, all the domain elements in a map are unique.**
- **The ordering of the maplets is not significant – the map m above could be specified, without changing the meaning, as:**

$m = \{a \longmapsto y, b \longmapsto x, c \longmapsto x, d \longmapsto z\}$

$m = \{d \longmapsto z, a \longmapsto y, c \longmapsto x, b \longmapsto x\}$

Empty map $\{\longmapsto\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Operators

Domain and Range Operator

The *domain* operator, **dom**, returns the set of all the domain elements of the maplets.

The *range* operator, **rng**, returns the set of all the range elements.

$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$

$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$

$m3 = \{ f \mapsto 2, g \mapsto 6\}$

- **dom** $m1 = \{a, b, c, d, e\}$
- **rng** $m1 = \{1, 2, 3, 4 \}$
- **dom** $m2 = \{a, f, c\}$
- **rng** $m2 = \{1, 2, 7\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Operators

- **Union Operator**

  $m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$

  $m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$

  $m3 = \{ f \mapsto 2, g \mapsto 6\}$

  $m1 \cup m3 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4, f \mapsto 2, g \mapsto 6\}$

- **The union operator is defined only if no two domain elements are the same; if this is not the case, then union is undefined**

  "$m1 \cup m2$" and "$m2 \cup m3$" are undefined

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Operators

Override (†) Operator:

- **In the case where two or more domain elements are the same in both maps, we can use the override operator (†).**
- **If the domain element of a maplet is the same in both sets, the second maplet wins.**

$m1 = \{a \longmapsto 1, b \longmapsto 2, c \longmapsto 2, d \longmapsto 3, e \longmapsto 4\}$

$m2 = \{a \longmapsto 2, f \longmapsto 1, c \longmapsto 7\}$

$m3 = \{ f \longmapsto 2, g \longmapsto 6\}$

$m1 \dagger m2 = \{a \longmapsto 2, b \longmapsto 2, c \longmapsto 7, d \longmapsto 3, e \longmapsto 4, f \longmapsto 1\}$

$m3 \dagger m2 = \{ f \longmapsto 1, g \longmapsto 6, a \longmapsto 2, c \longmapsto 7\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Operators

Domain Restriction (⩤) Operator:

- **Defined with two operands. The first is a set and the second is a map.**
- **The result yields a map that contains only those maplets whose domain element is in the set. For example**

$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$

$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$

$m3 = \{ f \mapsto 2, g \mapsto 6\}$

$\{a, c, e\} \lhd m1 = \{a \mapsto 1, c \mapsto 2, e \mapsto 4\}$

$\{e, f\} \lhd m2 = \{f \mapsto 1\}$

$\{\} \lhd m3 = \{\mapsto\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Operators

Domain Deletion (⩤) Operator:

- **Behaves similar to Domain Restriction operator but it deletes the maplet in questions.**

$m1 = \{a \mapsto 1, b \mapsto 2, c \mapsto 2, d \mapsto 3, e \mapsto 4\}$

$m2 = \{a \mapsto 2, f \mapsto 1, c \mapsto 7\}$

$m3 = \{ f \mapsto 2, g \mapsto 6\}$

$\{a, c, e\} \lhd m1 = \{b \mapsto 2, d \mapsto 3\}$

$\{e, f\} \lhd m2 = \{a \mapsto 2, c \mapsto 7\}$

$\{\} \lhd m3 = \{f \mapsto 2, g \mapsto 6\}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Map Application

- **If we apply our map to a particular domain element, then the result is the range element.**

  $m1 = \{a \longmapsto 1, b \longmapsto 2, c \longmapsto 2, d \longmapsto 3, e \longmapsto 4\}$

  $m2 = \{a \longmapsto 2, f \longmapsto 1, c \longmapsto 7\}$

  $m3 = \{ f \longmapsto 2, g \longmapsto 6\}$

  m1(d) = 3

  m2(f) = 1

  m3(f) = 2

  m3(x) is undefined

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Using the Map Type in VDM-SL

- **To declare a variable to be of type Map we use a special arrow $\xrightarrow{\mathrm{m}}$**
- **For example, to declare a variable m that maps characters to natural numbers we would write:**

$$c: Char \xrightarrow{m} \mathbb{N}$$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

13.2

Topics Covered

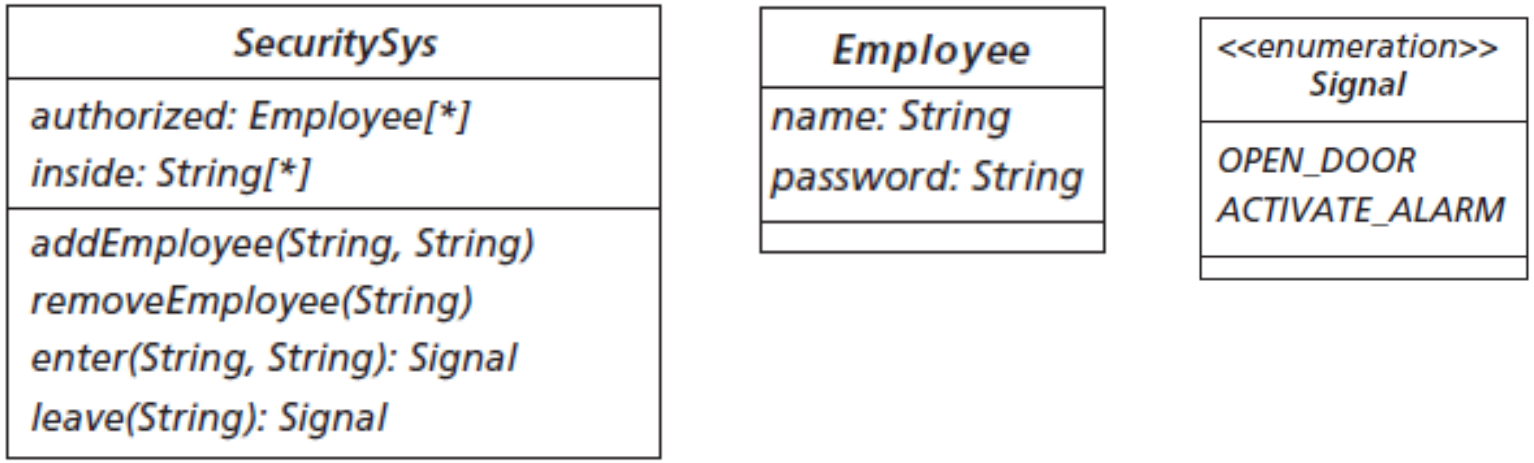
Specifying a High-Security Building

a High-Security Building

Only authorized employees are allowed entry to the building and each one consists of a user name (which is unique) and a password, both of which must be supplied when the individual wishes to enter the building. If the details are correct, a signal is sent to the hardware instructing it to open the door, and the member of staff is recorded as being inside the building. When the member of staff wishes to leave the building, the individual supplies his or her user name, and as long as the user is recorded as being currently inside the building, a signal is sent to the hardware to open the door, and the employee is recorded as having left.

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

a High-Security Building



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

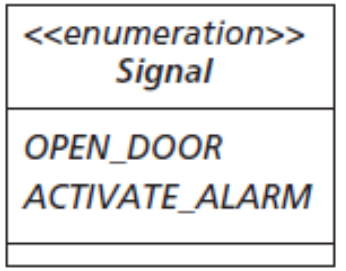
VDM-SL Specification

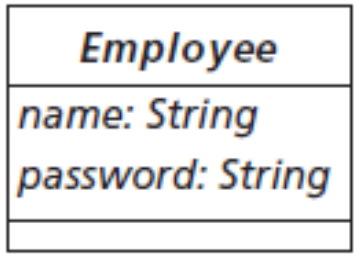
types
*String = Char**
Signal=<OPEN_DOOR>|<ACTIVATE_ALARM>

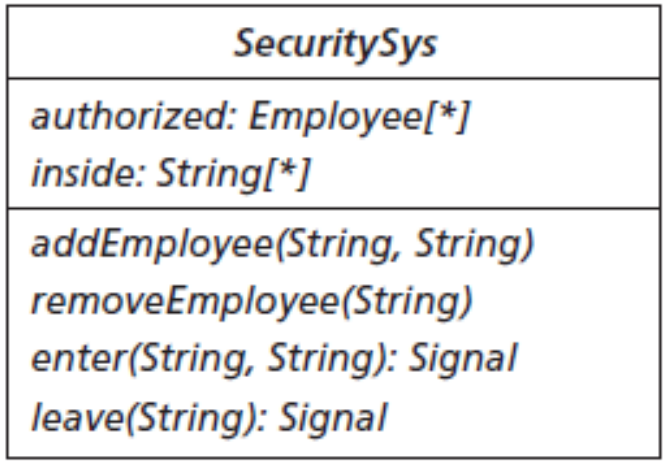
$$authorized: string \xrightarrow{m} string$$

state *SecuritySys* **of**

$$authorized: string \xrightarrow{m} string$$
*inside : String-**set***







Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

inv mk-*SecuritySys*(*a*,*i*) $i \subseteq \mathbf{dom}\ a$
init mk-*SecuritySys*(*a*,*i*) $\triangleq\ a = \{\mapsto\} \wedge i = \{\}$

addEmployee(nameIn : String, passwordIn : String)
ext wr authorized: string $\xrightarrow{m}$ *string*
pre nameIn $\notin$ *dom authorized*
post authorized $=$ $\overline{\textit{authorized}}$ $\cup$ *{nameIn* $\mapsto$ *passwordIn}*

removeEmployee(nameIn : String)
ext wr authorized: string $\xrightarrow{m}$ *string*
*rd inside: String-**Set***
pre nameIn $\in$ *dom authorized* $\wedge$ *nameIn* $\notin$ *inside*
post authorized = {nameIn} $\lhd$ $\overline{\textit{authorized}}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

enter(*nameIn* : *String*, *passwordIn* : *String*) *signal* : *Signal*
ext rd *authorized* : *String* $\xrightarrow{m}$ *String*
wr *inside* : *String*-**set**
pre TRUE
post (*authorized*(*nameIn*) = *passwordIn* ∧ *nameIn* ∉ $\overline{inside}$)
∧ (*inside* = $\overline{inside}$ ∪ {*nameIn*} ∧ *signal* = <OPEN_DOOR>)
∨ (*authorized*(*nameIn*) ≠ *passwordIn* ∨ *nameIn* ∈ *inside*)
∧ (*inside* = $\overline{inside}$ ∧ *signal* = <ACTIVATE_ALARM>)

leave(*nameIn* : *String*) *signal* : *Signal*
ext wr *inside* : *String*-**set**
pre TRUE
post *nameIn* ∈ $\overline{inside}$ ∧ *inside* = $\overline{inside}$ \{*nameIn*} ∧ *signal* = <OPEN_DOOR>
∨ *nameIn* ∉ *inside* ∧ *inside* = $\overline{inside}$ ∧ *signal*=<ACTIVATE_ALARM>

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

13.3

Topics Covered

Robot Monitoring System

Robot Monitoring System

- **A software system that monitors a number of robots working at a space station.**
  - ***Each robot will have a unique name and a mode, which can be WORKING, IDLE or BROKEN.***
  - ***There are two sectors, A and B, in which a robot can be set to work.***

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

Robot Monitoring System

addRobot: accepts the name of a new robot and records the fact that this robot has been added to the collection. Its mode is set to idle and it is therefore not allocated a sector to work in.

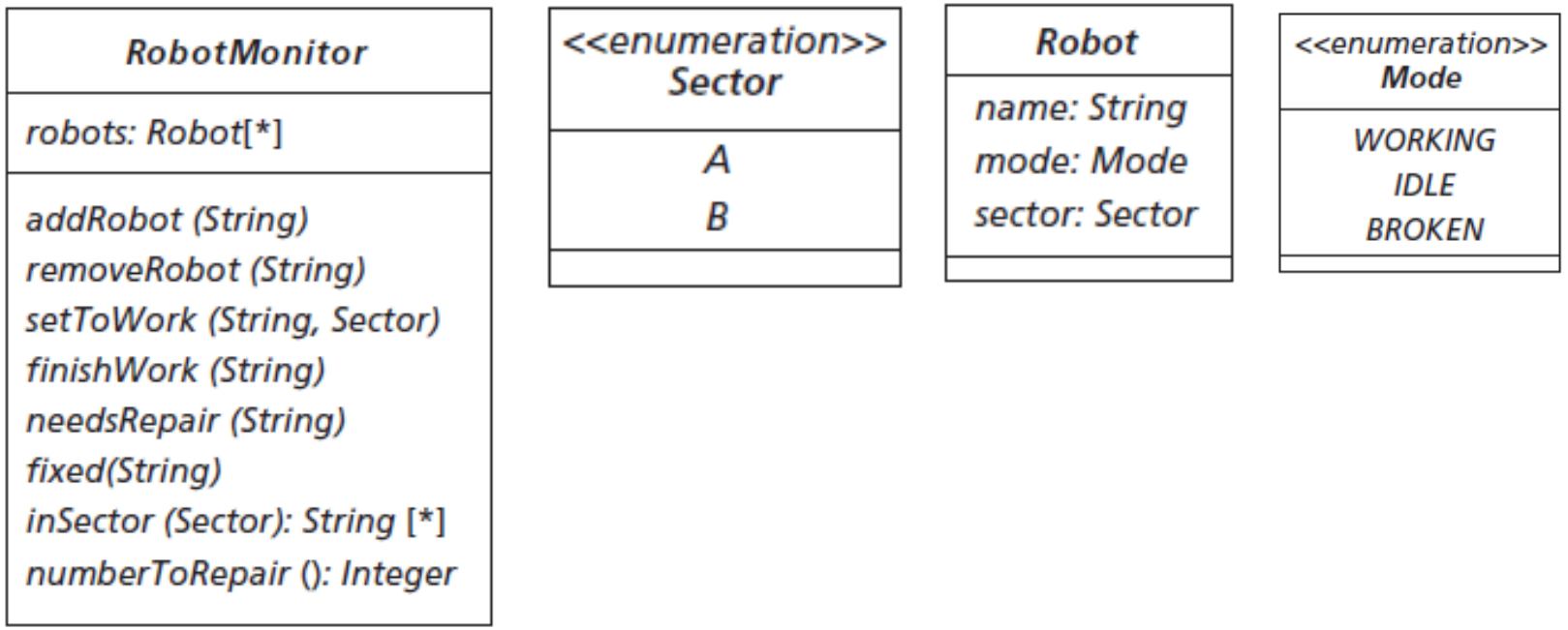
removeRobot: accepts the name of a robot and records the removal of this robot from the system.

setToWork: accepts the name of a robot, that must currently be idle, and records the fact that it has been set to work in a given sector.

- **finishWork**: accepts the name of a robot, and records the fact that this robot has been removed from the sector and that its mode has been set to idle.
- **needsRepair**: as above but records its mode as broken.
- **fixed**: accepts the name of a broken robot and records that that its mode has been set to idle.
- **inSector**: accepts a given sector and returns the names of those robots in that sector.
- **numberToRepair:** returns the number of broken robots.

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

a High-Security Building



Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

types
String = *Char**
Mode = <WORKING>|<IDLE>|<BROKEN>
Sector= <A>|<B>
Robot :: *name*: *String*
mode: *Mode*
sector: [*Sector*]

inv mk-*Robot*(-, *m*, *s*) $\triangleq$ *m* = WORKING $\Leftrightarrow$ *s* $\neq$ **nil**

state *RobotMonitor* **of**
robots : *String* $\xrightarrow{m}$ *Robot*
inv mk-*RobotMonitor*(*r*) $\triangleq$ $\forall$ *n* $\in$ **dom** *r* $\bullet$ *n* = *r*(*n*).*name*
init mk-*RobotMonitor*(*r*) $\triangleq$ *r* = {$\mapsto$}
end

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

addRobot (nameIn: String)

ext wr *robots: String* $\xrightarrow{m}$ *Robot*
pre *nameIn* $\notin$ *dom robots*
post *robots* = $\overline{\textit{robots}}$ $\cup$ {*nameIn* $\mapsto$ *mk-Robot (nameIn, <IDLE>, nil)*}

removeRobot (nameIn: String)

ext wr *robots: String* $\xrightarrow{m}$ *Robot*
pre *nameIn* $\in$ ***dom*** *robots* $\wedge$ *robots(nameIn).mode* $\neq$ *<WORKING>*
post *robots* = {*nameIn*} $\lhd$ $\overline{\textit{robots}}$

setToWork (nameIn: String,sectorIn:Sector)

ext wr *robots: String* $\xrightarrow{m}$ *Robot*
pre *nameIn* $\in$ ***dom*** *robots* $\wedge$ *robots(nameIn).mode* = *<IDLE>*
post *robots* = $\overline{\textit{robots}}$ † { *nameIn* $\mapsto$ **mk**-*Robot*(*nameIn*, <WORKING>, *sectorIn*) }

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

finishWork (nameIn: String)

ext wr $robots: String \xrightarrow{m} Robot$

pre $nameIn \in \mathbf{dom}\ robots \wedge robots(nameIn).mode = \langle WORKING \rangle$

post $robots = \overline{robots} \dagger \{ nameIn \mapsto \mathbf{mk}\text{-}Robot(nameIn, \langle IDLE \rangle, nil) \}$

needsRepair (nameIn: String)

ext wr $robots: String \xrightarrow{m} Robot$

pre $nameIn \in \mathbf{dom}\ robots$

post $robots = \overline{robots} \dagger \{ nameIn \mapsto \mathbf{mk}\text{-}Robot(nameIn, \langle BROKEN \rangle, nil) \}$

fixed (nameIn: String)

ext wr $robots: String \xrightarrow{m} Robot$

pre $nameIn \in \mathbf{dom}\ robots \wedge robots(nameIn).mode = \langle BROKEN \rangle$

post $robots = \overline{robots} \dagger \{ nameIn \mapsto \mu(\overline{robots}(nameIn), mode \mapsto \langle IDLE \rangle) \}$

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

VDM-SL Specification

inSector (*sectorIn:Sector*) *result* : *String*-**set**
ext rd *robots: String* $\xrightarrow{m}$ *Robot*
pre *TRUE*
post *result* = {*r.name* | *r* ∈ **rng** *robots* ● *r.sector* = *sectorIn*}

numberToRepair() *number* : ℕ
ext rd *robots: String* $\xrightarrow{m}$ *Robot*
pre *TRUE*
post *number* = **card** {*r* | *r* ∈ **rng** *robots* ● *r.mode* = <BROKEN>}

Reference: Formal Software Development From VDM to Java by Quentin Charatan and Aaron Kans

CS636 Formal Methods

week 14

Topics Covered: 128 – 137

Introducton to Z - Part 1

Formal Methods

Topic#128
Introduction to Z

Z-Language

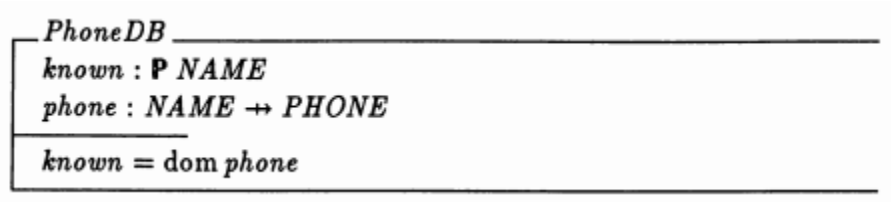
- The Z notation is a language and a style for expressing formal specifications of computing systems.
- It is based on a typed set theory, and the notion of a “schema” is one of its key features.
- A schema consists of a collection of named objects with a relationship specified by some axioms, and Z provides notations for defining schemas and later combining them in various ways, so that large specifications can be built up in stages.
- Schemas can have generic parameters, and there are operations in Z for creating instances of generic schemas.

Use of Schemas

- The use of schemas allows the specification to be presented gradually, with a close correspondence between the mathematical text and prose commentary. This makes it easy to explain the mathematics as it is presented and to relate the variables in the mathematical text to the system they describe.

Example: a small database

- The database contains a number of people's names, and against each name is stored a telephone number. It has operations for adding a new name and telephone number, and for enquiring what number is stored against a given name. The *state-space* of the database system is described by a schema called PhoneDB:



Possible state of the system

- Possible state of the system is:

$$
\begin{aligned}
known &= \{\,\text{Smith}, \text{Jones}, \text{Robinson}\,\} \\
phone &= \{\,\text{Smith} \mapsto 01\text{–}325\text{–}4939, \\
&\qquad\ \ \text{Jones} \mapsto 0865\text{–}54141, \\
&\qquad\ \ \text{Robinson} \mapsto 0865\text{–}54141\,\}
\end{aligned}
$$

Formal Methods

Topic#128

END!

Formal Methods

Topic#129
Why formal semantics?

Overview

- A guiding principle of the Z approach to specification has been the use of the ordinary structures of mathematics in the writing of software specifications.
- There are several advantages in this: the familiar language of sets and relations proves to be sufficient to describe succinctly the abstract structures needed in programming and is already known to every mathematician and also to many non-specialists.
- Applied mathematicians - spend little time worrying about the “formal semantics” of the notations they use and the “rules of inference” used to manipulate them.

Why formal semantics?

Why should we be concerned with these things when try to apply mathematics to the new sphere of software design?

-> A first answer lies on the nature of the notations themselves.

-> A second argument in favour of formal semantics is its consequences for the practice of specification.

Formal Methods

Topic#129

END!

Formal Methods

Topic#130
Meta-circularity

Meta-circularity

- The formal semantics of the Z notation given in this book is itself written using Z as a meta-language.
- This idea of using a notation to give a “meta-circular” description of its own semantics forms a long-standing tradition in Computer Science.
- In informal terms, if someone didn’t understand Z at all, we could hardly expect his understanding to be improved by a look at a formal semantics written in Z, although perhaps if he had a partial understanding, he could use the semantics to clear up some remaining areas of doubt.
- More formally, we might hope that the desired semantics might be found as a “fixed-point” of the definition.

Example

- A more subtle problem is that two slightly different but inconsistent suppositions about the semantics can both be supported by a meta-circular definition.
- **For example**, unless special precautions are taken, a meta-circular definition of a programming language will not tell us whether parameters to subroutines are passed by name or value.
- If we read the text of the definition under the assumption that parameters are passed by name, then the definition will appear to describe call-by-name, and if we assume call-by-value semantics, then the definition will appear to describe call-by-value: compare.

Why do these criticisms not invalidate a meta-circular definition of Z?

- Semantics consists simply of the development of certain mathematical theory, and this development could, at least in principle, be carried out without using Z, but rather say the basic language of first-order logic.

This argument encourages us to see the meta-circular semantics ultimately as an informal sketch of a semantics which might be formalized fully in some more primitive but less expressive logical language.

Formal Methods

Topic#130

END!

Formal Methods

Topic#131

Z and other methods

Z and other methods

- Styles are divided into model-oriented methods
- Aim of a specification is to construct an abstract model of the information system being specified.
- And property-oriented or algebaric methods, where the aim is to describe a system in terms of its desired properties, without constructing and explicit model.

Methods

Model oriented methods

- Z
- VDM (Vienna Development Method)

Algebaric methods

- Clear
- OBJ
- ACT ONE

- This distinction between model-oriented and property oriented methods is not as clear-cut as it might at first appear; in practice, Z specifications often describe certain aspects of systems by giving axioms which must be satisfied by the system, and this amounts to a property-oriented specification.
- Algebaric specifications often describe a collection of a basic data-types in a property oriented way, then use these to build a model of the system being specified.

Model-oriented methods

- The closest method to Z is the “Vienna Development Method (VDM)”, which originated at the IBM Vienna Laboratory, and has been developed in the work of Dines Bjorner and Cliff Jones.
- In their aims, VDM and Z are quite similar, but there are a number of differences of style which give each method its own advantages and disadvantages.

- As an aid to comparison, what follows is a specification in the language of VDM of the telephone-number database as we discussed previously. The state of the system is describes in VDM as follows:

$$
\begin{aligned}
&\mathit{PhoneDB}:: \mathit{known}: \textbf{set of } \mathit{NAME} \\
&\qquad\qquad \mathit{phone}: \textbf{map } \mathit{NAME} \textbf{ to } \mathit{PHONE} \\
&\textbf{where} \\
&\mathit{inv\text{-}PhoneDB} \triangleq \mathit{known} = \textbf{dom}\, \mathit{phone}.
\end{aligned}
$$

AddPhone operation

AddPhone(*name*: *NAME*, *number*: *PHONE*)
ext wr *known*: **set of** *NAME*
wr *phone*: **map** *NAME* **to** *PHONE*
pre $name \notin known$
post $known = \overleftarrow{known} \cup name \wedge$
$phone = \overleftarrow{phone} \cup \{name \mapsto number\}.$

FindPhone operation

FindPhone(*name*: *NAME*) *number*: *PHONE*
ext rd *known*: **set of** *NAME*
rd *phone*: **map** *NAME* **to** *PHONE*
pre $name \in known$
post $number = phone(name).$

Similarities between Z and VDM

- Both in the style of the methods and in mathematics used to support them.
- Both use ordinary mathematical structures--sets, functions, and sequences--to model data.
- Both use the notation of predicate logic to describe operations on the data.
- In both methods, a specification typically consists of the description of a state space followed by the description of operations which change the state.

Algebaric specifications

- Algebaric specifications begin with a world which is far simpler than the rich universe of structured types assumed by Z and VDM.
- Their basic cocabulary is just some named sets and some total functions on these sets.
- Specifications describe the properties these functions are required to satisfy, typically by giving equations which relate the functions to each other.

Clear specification language

- As an example, here is the telephone number database specified once more, this time in the algebaric specification language Clear.
- The specification starts with two basic types, names and telephone numbers.
- We need to be able to tell if two names are same, so names come equipped with an equality test:

const *Name* =
  enrich *Bool* **by**
    sorts *name*
    opns $\_ == \_ : name, name \rightarrow bool$
    eqns $n == n = true$
      $n == m = m == n$
      $n == m \wedge m == p \Rightarrow n == p = true$
  enden

- For telephone number, no equality test is needed, but there must be a special number *unknown* which is the result of trying to find a name not in the database.

```
const Phone =
   theory
      sorts phone
      opns unknown : phone
   endth
```

Sorting can be described as a higher-order function which maps ordering relations to functions from sequences to sequences.

Formal Methods

Topic#131

END!

Formal Methods

Topic#132

The world of sets - I

The world of sets

- Specifications in Z describe sets, and its constructs have their meaning in operations on sets. This makes it natural to start an account of the semantics of Z by describing:
  - What sets there are?
  - What essential properties they have?
  - What operations can be performed on them?
- Set theory as an axiomatic theory of first-order logic

ZF (Zermelo and Fraenkel) axiom system

The ZF axioms describe a universe of “pure” sets in which everything is a set: this universe can be conceived as being built up starting with just the empty set.

Sets and membership

- A specification for the world of sets can be obtained by taking each of the ZF axioms and defining in Z a corresponding operation on sets.
- Take as an example the *union* axiom:

$$\forall x.\ \exists y.\ \forall z.\ z \in y \Leftrightarrow (\exists w.\ z \in w \wedge w \in x).$$

- This says that for each set x, there is a set y whose elements are exactly the elements of elements of x.

- Now let W stands for the “world” of sets, and let E stands for the membership relation on W. An important property of E is its extensionality, that any two sets with the same elements are equal:

$$[W]$$

$$\_ \in \_ : W \leftrightarrow W$$

$$\forall x, y : W \bullet (\forall z : W \bullet z \in x \Leftrightarrow z \in y) \Rightarrow x = y.$$

- The union operation can be defined as a function from W to W:

$$union : W \to W$$

$$\forall x, y : W \bullet$$
$$y \in union(x) \Leftrightarrow (\exists z : W \bullet y \in z \land z \in x).$$

- The other operations of set theory can be described in a similar way.

- By formulating the axioms of set theory in this way, we obtain a specification in Z which describes a world of sets W and a collection of set-theoretic operations taking sets in W to other sets in W.

Formal Methods

Topic#132

END!

Formal Methods

Topic#133

The world of sets - II

Basic operations

- The other operations on sets can be specified in the same way as union.
- The *null* set axiom just asserts the existence of a set with no elements, which can null:

$$\frac{null : W}{\forall x : W \bullet \neg\, x \in null.}$$

- The pair-sets axiom allows the two-element set {x,y} to be constructed for any sets x and y, and we call the operation taking x and y to the W representation of {x,y} by the name *pair*:

$$\frac{pair : W \times W \to W}{z \in pair(x, y) \Leftrightarrow z = x \vee z = y.}$$

- Of course, pair(x,y) has the single element x. This allows singleton sets to be constructed; the operation *sing* taking x to the W representation of {x} can be defined in the terms of pair:

$$\begin{array}{|l} sing : W \to W \\ \hline sing(x) = pair(x, x). \end{array}$$

- The combination of union and pair allows binary unions to be formed:

$$\begin{array}{|l} \_ \sqcup \_ : W \times W \to W \\ \hline x \sqcup y = union(pair(x, y)). \end{array}$$

- A set of x is a subset of another set y if every element of x is also an element of y:

$$\_ \sqsubseteq \_ : W \leftrightarrow W$$
$$y \sqsubseteq x \Leftrightarrow (\forall z : W \bullet z \in y \Rightarrow z \in x).$$

- The power set axiom asserts the existence of the power set of P x of any set x; its elements are just the subset of x:

$$power : W \rightarrow W$$
$$y \in power(x) \Leftrightarrow y \sqsubseteq x.$$

- The axiom of *infinity* ensures that there is at least one infinite set in W. There are several formulations, but we choose one which describes a certain set *bigset,* which must be infinite, for it contains ∅ and is closed under the operation taking x to x U {x}:

$$
\begin{array}{|l}
bigset : W \\
\hline
null \in bigset \\
\forall x : W \bullet x \in bigset \Rightarrow x \sqcup sing(x) \in bigset.
\end{array}
$$

- Without this axiom, there is no gurantee that the world of sets will contain an infinite set at all.

- The axiom of *seperation* is actually an axiom scheme: if

$$\phi(a, b_1, \ldots, b_n)$$

- is a formula of set theory with free variables among a, b_1, ..., b_n, and x, w_1, ..., w_n are sets, then the axiom asserts the existence of a set

$$y = \{ z \in x \mid \phi(z, w_1, \ldots, w_n) \}.$$

- For present purposes, it is enough to identify the formula $\varnothing$ with the subset of W it describes. Just as some formulae of set theory describe collections too large to be sets, so there are some subsets S of W too large to be "represented" in W: there may be no x in W with

$$\forall z : W \bullet z \in x \Leftrightarrow z \in S.$$

- This is the reason why the axiom of seperation must give the set y as a subset of an already-known set x. it is modelled by the operation *filter*:

$$\begin{array}{|l} filter : W \times \mathbb{P}\, W \to W \\ \hline \forall x, z : W; S : \mathbb{P}\, W \bullet \\ \quad z \in filter(x, S) \Leftrightarrow z \in x \wedge z \in S. \end{array}$$

Derived operations

- The six operations *union, null, pair, power, bigset*, and *filter*, together with the axioms of extensionality and regularity, form the basis of our view of set theory.
- In addition to these primitive operations, some other operations for forming tuples and Cartesian products are needed in the sementics of Z.
- These operations could in principle be defined in terms of the primitive operations--one way of doing this would be to define an ordered pair constructor which encoded (x,y) as

$$\{\{x\},\{x,y\}\}.$$

- This encoding *couple* could be defined in terms of the primitive operations on the world of sets:

$$\begin{array}{|l} couple : W \times W \to W \\ \hline couple(x, y) = pair(sing(x), pair(x, y)). \end{array}$$

- Tuples with more than two elements could then be constructed by iterating the ordered pair construction, and so on. But a different approach is taken here; the operation are characterized by axioms and not defined explicitly in terms of the basic operations.
- The consistency of the axiomatic presentation could be proved by using the basic operations to give an explicit construction like this one.

- One derived operation takes a finite sequence of sets and forms a "tuple" from it: the operation maps $x_1, ..., x_n$ into a set representing in W the tuple $(x_1, ..., x_n)$.

$$tuple : \text{seq } W \to W$$

$$\forall x, y : \text{seq } W \bullet$$
$$\#x = \#y \land tuple(x) = tuple(y) \Rightarrow x = y.$$

- The axiom says that for two tuples of the same length to be equal, they must have the same components. It does not require, for example, that

$$tuple\ \langle x, y \rangle \neq tuple\ \langle p, q, r \rangle.$$

- This freedom allows tuple to be defined constructively by iterating the couple operation.

Formal Methods

Topic#133

END!

Formal Methods

Topic#134

Types

Types

- Every variable introduced in a Z specification is given a type.
- There are several reasons for this:

-> The first reason is technical, and is connected with the operation comprehension by which a set can be made from any schema: if A is a schema, then {A} is the set of all bindings made from models of A.

-> Practice of reading

-> Writing specifications

- The theory of types can be made decideable, so that it is possible to check automatically that a specification is well-typed.

- Experience with programming languages shows that type-checking is a valuable way of catching minor errors, such as functions applied to the wrong number of arguments, and these errors are likely to be just as common in specifications as in programs.

Syntax of types

- In the z notation, the abstract syntax of types can be taken as

$$
\begin{aligned}
TYPE ::= & \; givenT \langle\langle NAME \rangle\rangle \\
| \; & \; powerT \langle\langle TYPE \rangle\rangle \\
| \; & \; tupleT \langle\langle \mathrm{seq}\ TYPE \rangle\rangle \\
| \; & \; schemaT \langle\langle IDENT \pfun TYPE \rangle\rangle .
\end{aligned}
$$

- This corresponds with the more informal notation as follows:

$$
\begin{array}{ll}
X & \widehat{=}\ givenT\ X \\
\mathbb{P}\ a & \widehat{=}\ powerT\ a \\
a_1 \times \cdots \times a_n & \widehat{=}\ tupleT\ \langle a_1, \ldots, a_n \rangle \\
(\!| x_1 : a_1; \ldots; x_n : a_n |\!) & \widehat{=}\ schemaT\ \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}.
\end{array}
$$

- If given is an alphabet of names for given types; we let Type(given) be the set of all types built from names in given:

$$
\begin{array}{l}
Type : \mathbb{P}\, NAME \rightarrow \mathbb{P}\, TYPE \\
names : TYPE \rightarrow \mathbb{P}\, NAME \\
\hline
Type(given) = \{\, a : TYPE \mid names(a) \subseteq given \,\} \\
names(givenT\ X) = \{X\} \\
names(powerT\ a) = names(a) \\
names(tupleT\ as) = \bigcup names(\mathrm{ran}\ as) \\
names(schemaT\ am) = \bigcup names(\mathrm{ran}\ am).
\end{array}
$$

Semantics of types

- So far, types have been regarded just as formal expressions.
- But the important thing about a type is that it determines a set of values which are its elements: we call this set the *carrier* of the type.
- If gset : NAME -> W as signs a set to each name is given, the carrier of each type in Type(given) can be found by interpreting the type-constructors as operations in the world of sets.

- We write *Carrier gset* a for the set of elements of the type a, and define this by structural recursion over the syntax of types:

$$
\begin{array}{|l}
\hline
Carrier : (NAME \nrightarrow W) \rightarrow (TYPE \nrightarrow W) \\
\hline
Carrier\ gset\ (givenT\ X) \cong gset(X) \\
Carrier\ gset\ (powerT\ a) \cong power(Carrier\ gset\ a) \\
Carrier\ gset\ (tupleT\ as) \cong cproduct(map\ (Carrier\ gset)\ as) \\
Carrier\ gset\ (schemaT\ am) \cong sproduct(map\ (Carrier\ gset)\ am).
\end{array}
$$

- An important characteristic of the type constructor is that they are monotonic with respect to inclusion--for example, if $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$, then

$$A_1 \times A_2 \subseteq B_1 \times B_2.$$

Type substitutions

- When a generic schema is instantiated with actual parameters, types are filled in for the given-set names which are its formal parameters: this means that a substitution has to take place on the types of the variables of the schema. this process is described by the function *tsubst*:

$$
\begin{array}{|l}
tsubst : (NAME \pfun TYPE) \fun (TYPE \pfun TYPE) \\
\hline
tsubst\ f\ (givenT\ X) \cong f(X) \\
tsubst\ f\ (powerT\ a) \cong powerT\ (tsubst\ f\ a) \\
tsubst\ f\ (tupleT\ as) \cong tupleT\ (map\ (tsubst\ f)\ as) \\
tsubst\ f\ (schemaT\ am) \cong schemaT\ (map\ (tsubst\ f)\ am).
\end{array}
$$

- The function on the previos slide has several noteworthy properties. It has the the syntactic property that if the domain and range of *f* use certain alphabets, then so does *tsubst f*:

$$\vdash f \in given \to Type(given') \Rightarrow \\ tsubst\, f \in Type(given) \to Type(given').$$

- This is proved by a simple structural induction, as is the following property:

$$\vdash f \in given \to Type(given') \wedge g \in given' \to Type(given'') \Rightarrow \\ (tsubst\, g) \circ (tsubst\, f) = tsubst\, ((tsubst\, g) \circ f).$$

Formal Methods

Topic#134

END!

Formal Methods

Topic#135

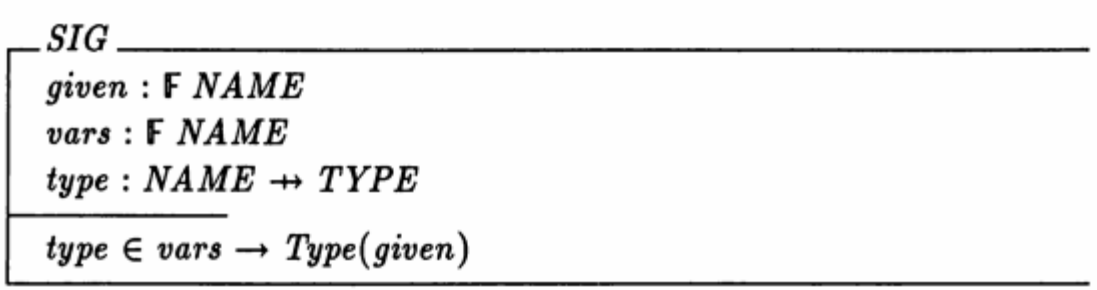
Signatures, structures and varieties

Signatures, structures and varieties

- The declartaive information in a schema is captured in its signature: this records the names of the schema's components or local variables, their types, and the given-set names assumed by the schema.
- **Signatures** are the finite objects, and are thus suitable for mechanical representations and manipulation.
- But a schema contains more information than just the declarations; the axiom part of the schema can describe a relationship among the variables , and this information can be captured by describing which **structures**--assignment of values to the variables--satisfy the axiom part.
- A signature together with the class of appropriately shaped structures is called **variety**.

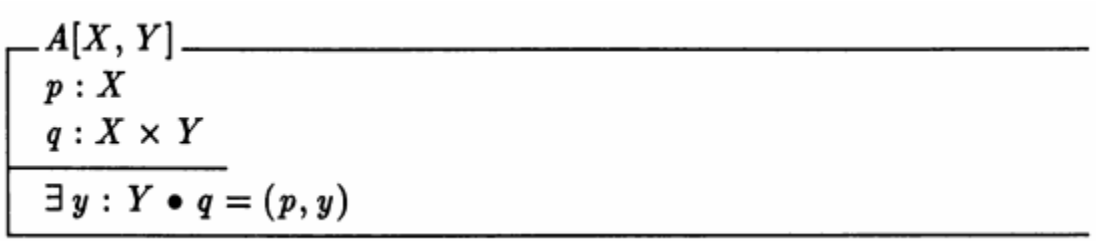
Signatures

- A signature defines an alphabet of given-set names, from which types can be built, and an alphabet of variables names, and it assigns a type to each variable:



- This axiom says that the typing function *type* assigns a type to exactly those variables in the alphabet *vars*, and these types formed from the given-set names in the alphabet given.

- The following is a simple example of a schema:



- This has given-set names X and Y, and variables p and q. The signature of the schema A is

$$
\begin{aligned}
&\mu\, SIG \mid \\
&\quad given = \{\text{`}X\text{'}, \text{`}Y\text{'}\} \land \\
&\quad vars = \{\text{`}p\text{'}, \text{`}q\text{'}\} \land \\
&\quad type = \{\text{`}p\text{'} \mapsto \text{`}X\text{'}, \text{`}q\text{'} \mapsto tupleT\, \langle \text{`}X\text{'}, \text{`}Y\text{'} \rangle\}.
\end{aligned}
$$

Structures

- The information below the horizontal line--the axiom part of a schema --is captured by regarding sentences as determining a class of “structures”. taking the schema A as an example again, the structure

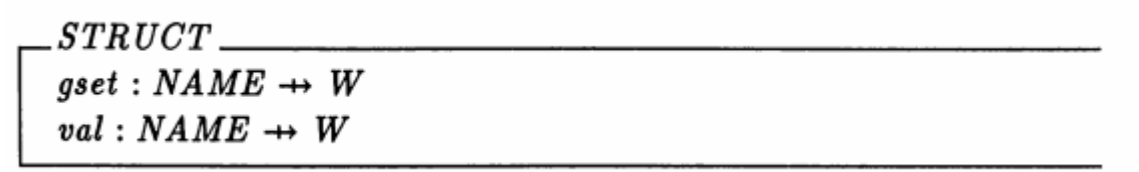
$$
\begin{aligned}
&`X' \mapsto \mathbb{N} \\
&`Y' \mapsto \{a, b, c\} \\
&`p' \mapsto 3 \\
&`q' \mapsto (3, b)
\end{aligned}
$$

satisfies the axiom, but the structure

$$
\begin{aligned}
&`X' \mapsto \{f, g, h\} \\
&`Y' \mapsto \{a, b, c\} \\
&`p' \mapsto h \\
&`q' \mapsto (g, b),
\end{aligned}
$$

although it also accords with the signature, fails to satisfy the axiom, because the value of p is not the same as the first component of the value of q.

- So a structure takes certain given-set names and variables and gives them values in the world of sets:



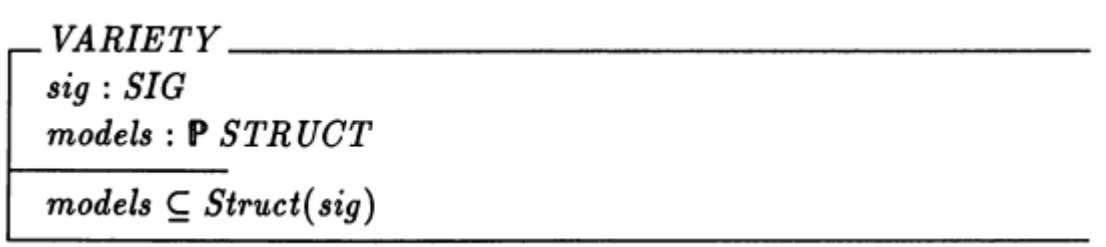
- The mapping *gset* is used to interpret given-set names, and *val* is used to interpret variables.
- The first requirement on the structure for a schema is that they be consistent with the signature: the domains of the *gset* and *val* mappings should be the alphabets of the signature, and the value given to each variable must be an element of its type.

- We define the function Struct to give the set of structures consistent with signature:

$$
\begin{array}{|l}
Struct : SIG \rightarrow \mathbb{P}\, STRUCT \\
\hline
Struct = \\
\quad \lambda\, SIG \bullet \{\, STRUCT \mid \\
\qquad \mathrm{dom}\, gset = given \land \\
\qquad \mathrm{dom}\, val = vars \land \\
\qquad (\forall\, v : vars \bullet val\, v \in Carrier\, gset\, (type\, v))\, \}.
\end{array}
$$

Variety

- Now a variety--the meaning of a schema--can be defined as a signature together with a set of structures for the signature:



- The *set models* will typically be smaller than *Struct(sig)* because some structures will fail to satisfy the axioms of the schema.

Formal Methods

Topic#135

END!

Formal Methods

Topic#136
Notation for denotaional semantics

Notation for denotational semantics

- The semantic is based on sets rather than Scott domains (Scott domain is an algebraic, bounded-complete cpo[complete partial order]), and merits the name “denotational” because it follows the style in which the meaning of a composite phrase is defined in terms of the meaning of its immediate constituents.

Abstract syntax notation

- We adapt the Abstract syntax notation of Z to allow constructors which mimic the intended concrete syntax of the object language.
- So instead of giving the syntax of expressions as, say,

$$
\begin{aligned}
EXP ::= & \ var \langle\langle IDENT \rangle\rangle \\
| & \ plus \langle\langle EXP \times EXP \rangle\rangle \\
| & \ times \langle\langle EXP \times EXP \rangle\rangle
\end{aligned}
$$

we might write the following:

$$
\begin{aligned}
EXP ::= & \ IDENT \\
| & \ EXP + EXP \\
| & \ EXP * EXP.
\end{aligned}
$$

- If these production rules were regarded as a context-free grammer, the resulting language would be ambiguous--in the previous slide example, the string “x+y*z” could be parsed as either “x+(y*z)”--but we don’t regard this ambiguity as important, because the intention is still that the production rules describe certain tree structures.
- When constructors from the abstract syntax are applied, the resulting term is written in open-face square bracket $[\![ ]\!]$, as is usual in denotational semantics.

- So the notation

$$[\![(e_1 + e_2) * e_3]\!]$$

means the same as

$$times(plus(e_1, e_2), e_3)$$

in the conventional notation. This extension to Z helps to make the semantic equations more readable; the definitions could in principle all be written using the conventional notation, but the result would be a loss of clarity and no real gain in rigour.

Strong equality

- The second extension is the use of the strong equality sign $\cong$ in defining partial functions.
- In denotational semantics, one often needs to define a function whose domain is identical with the domain of definition of an expression, and it becomes tedious to write out an explicit axiom to fix the domain.
- In such situations, the strong equality sign $\cong$ can be used; the equation

$$t_1 \cong t_2$$

is true when either both t1 and t2 are defined and they have the same value, or both are undefined, and is false otherwise.

- This means that the specification

$$f, g, h : \mathbb{N} \pfun \mathbb{N}$$
$$\forall x : \mathbb{N} \bullet f(x) \cong g(x) + h(x).$$

is wholly equivalent to the second of those above: in particular *f(x)* will be defined just when *g(x)+h(x)* is defined, i.e. just when both *g(x)* and *h(x)* are defined. In consequence, $dom\ f = dom\ g \cap dom\ h$.

Generalized mu-terms

- The third and final extension to the Z notation is a generalized form of the μ-*term* which gives something like the effect of a **let**-definition in an **ISWIM** like programming language. An example of such a generalized μ-*term* is

$$\mu\, x : \mathbb{N} \mid x * x = 16 \bullet x + 3.$$

- This has value of 7, because there is just one value of x, namely x=4, which satifies the predicate x*x=16, and with this value of x, the term x+3 takes the value 7.

Formal Methods

Topic#136

END!

Formal Methods

Topic#137

The language of schemas

The language of schemas

- We shall consider a little language of schemas, in which schema expressions can be combined with operations of conjunction, disjunction and projection:

$$
\begin{aligned}
SEXP ::= & \cdots \\
& \mid SEXP \wedge SEXP \\
& \mid SEXP \vee SEXP \\
& \mid SEXP \restriction SEXP \\
& \mid \cdots
\end{aligned}
$$

- The meaning of a schema expression is defined by a semantic function *sexp* which, given an environment of schema definitions, maps schema expressions to varieties.

$$sexp : ENV \rightarrow SEXP \twoheadrightarrow VARIETY$$

$$\ldots$$

$$sexp\ \rho\ [\![se_1 \wedge se_2]\!] \cong combine(sexp\ \rho\ [\![se_1]\!], sexp\ \rho\ [\![se_2]\!])$$

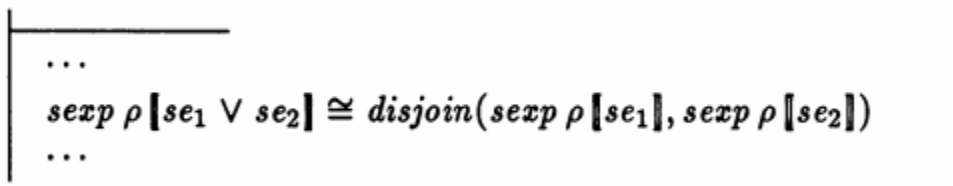
$$\ldots$$

- The meaning of the conjunction of two schema expressions is obtained by putting together the varieties corresponding to the two arguments using the auxiliary function *combine*.

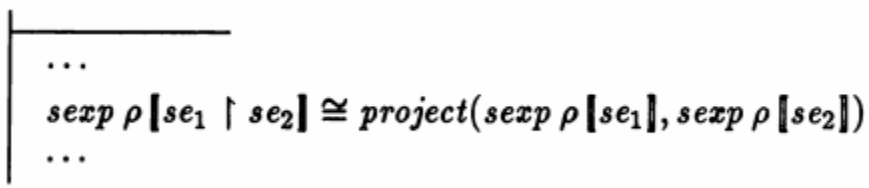
- This joins the signatures using the function *join*, which identifies the common variables, and the class of models is described in terms of the function *restrict*.
- The models of the conjoined schema are those which satisfy, in a certain sense, the axioms of both the argument.

$$
\begin{array}{|l}
combine : VARIETY \times VARIETY \pfun VARIETY \\
\hline
combine(\theta VARIETY_1, \theta VARIETY_2) \cong \\
\quad \mu\, VARIETY' \mid \\
\qquad sig' \cong join(sig_1, sig_2) \land \\
\qquad models' = \\
\qquad\quad \{\, M : Struct(sig') \mid \\
\qquad\qquad restrict\ sig_1\ M \in models_1 \land \\
\qquad\qquad restrict\ sig_2\ M \in models_2 \,\}.
\end{array}
$$

- The disjunction of two schemas A and B is defined in terms of an operation *disjoin*:



- The semantics of schema projection is defined in terms of an operation *project*:



Formal Methods

Topic#137

END!

CS636 Formal Methods

week 15

Topics Covered: 138 – 147

Introducton to Z - Part 2

Formal Methods

Topic#138
The Semantics of Z

The semantics of Z

- In Z language, specifications may introduce named schemas, global given-set names and global constants specified by axioms.
- Schemas may have generic parameters, and there are notations for taking an instance of a generic schema and for combining schemas with operations of the schema calculus.
- The abstract syntax for the language uses keywords rather than the more suggestive boxes for dealing schemas. this helps to make the semantic equations more concise, but the same semantics can be used with the more usual concrete syntax based on boxes.

- Level numbers are used to distinguish identifiers declared at different lexical levels, and this provides the means to model the Z scope rules.
- One important feature of the Z notation is the facility for making global generic definitions, used extensively in the standard library or tool-kit of mathematical definitions: the concept of relation, for example, is generic in the sets from which the domain and range are drawn.

Formal Methods

Topic#138

END!

Formal Methods

Topic#139
Language Summary

Language summary

- The syntax of the langage has been simplified to make the semantics more concise.
- One simplification is the use of keywords instead of boxes, and another is the omission of infix function and relation symbols; these are just a more readable way to write certain function applications and membership predicates.
- A specification consists of a sequence of definitions, which are of **three** kinds:

-> Definitions of global-set names

-> Definition of global variables

-> Definition of schemas

Definitions of global given-set names

- These introduce a number of given-set names global to the whole specification: typically, these given sets will be sets assumed known in the specification, but whose details are not important.
- An example might be the sets of valid file-names and data blocks in a filing system.
- Example

given *FILENAME, BLOCK.*

In the usual syntax of Z, these names are introduced by writing them in square brackets:

[*FILENAME, BLOCK*].

Definitions of global variables

- These introduce variables which are global to the whole specification, and allow axioms which constrain their values.
- The axioms need not determine the values of the variables exactly, but may leave some “looseness”: this allows objects to be named whose details are not needed in the specification; it also allows requirements to be captured accurately without unduly constraining the implementation.

- Examples:

$$
\begin{array}{l}
\textbf{let} \\
\quad f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
| \\
\quad \forall\, n, m : \mathbb{N} \bullet f(n, m) = 10 * n + m \\
\textbf{end.}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let} \\
\quad encode : CHAR \rightarrowtail \mathbb{N}; \\
\quad decode : \mathbb{N} \twoheadrightarrow CHAR \\
| \\
\quad decode = encode^{-1} \\
\textbf{end.}
\end{array}
$$

- In the usual syntax, definitions of global variables are written with a vertical bar on the left, but without bars above and below, for example:

$$
\left|\begin{array}{l}
f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\hline
\forall\, n, m : \mathbb{N} \bullet f(n, m) = 10 * n + m.
\end{array}\right.
$$

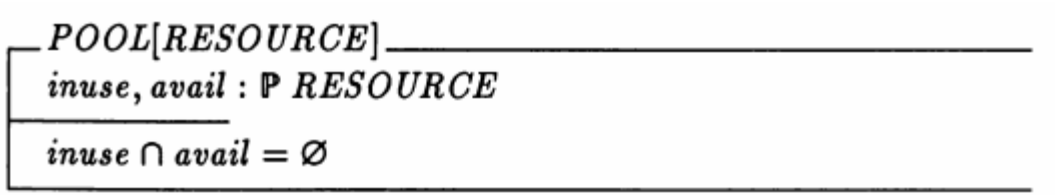
Definition of schemas

- Each of these introduces a new schema name, which may have generic parameters.
- The right-hand side of the definition may be simply the text of the schema, or it may be a schema expression built up from already-defined schema names.
- Examples:

$$\textbf{let } POOL[RESOURCE] = \textbf{schema}$$
$$\quad inuse, avail : \mathbb{P}\, RESOURCE$$
$$|$$
$$\quad inuse \cap avail = \varnothing$$
$$\textbf{end.}$$

$$\textbf{let } LOOKUP = (LOOKUP1 \wedge Ok) \vee ErrNotFound.$$

- In the usual syntax, definitions like the first one are given by attaching the schema name to a box containing the text:



- Definitions of one schema in terms of others use the definition sign $\widehat{=}$:

$$LOOKUP \mathrel{\widehat{=}} (LOOKUP1 \land Ok) \lor ErrNotFound.$$

- A *schema designator* is an applied occurance of a schema. As well as the schema name itself, it may contain a decoration to be applied uniformly to all the components of the schema, and some actual generic parameters.
- Schema designators may be used in schema expressions, and also in *declarations*, the parts of schema bodies which introduce variables.
- The axiom part of the schema body is a *predicate*, and these may be built up in any of the usual ways from the most basic predicates, which are equations and membership predicates between *terms*.

Syntax summary

Specifications

$$
\begin{aligned}
SPEC ::= & \ \textbf{given}\ IDENT, \ldots, IDENT \\
| & \ \textbf{let}\ SCHEMA\ \textbf{end} \\
| & \ \textbf{let}\ WORD[IDENT, \ldots, IDENT] = SEXP \\
| & \ SPEC\ \textbf{in}\ SPEC.
\end{aligned}
$$

Schema expressions

$$
\begin{aligned}
SEXP ::= & \ \textbf{schema}\ SCHEMA\ \textbf{end} \\
| & \ SDES \\
| & \ \neg\ SEXP \\
| & \ SEXP \wedge SEXP \\
| & \ SEXP \vee SEXP \\
| & \ SEXP \Rightarrow SEXP \\
| & \ SEXP \upharpoonright SEXP \\
| & \ SEXP \setminus (IDENT, \ldots, IDENT) \\
| & \ \exists\ SCHEMA \bullet SEXP \\
| & \ \forall\ SCHEMA \bullet SEXP.
\end{aligned}
$$

Schema designators

$$SDES ::= WORD\ DECOR[TERM, \ldots, TERM].$$

Schema bodies

$$SCHEMA ::= DECL \mid PRED.$$

Declarations

$$
\begin{aligned}
DECL ::= \; & IDENT : TERM \\
| \; & SDES \\
| \; & DECL ; \; DECL.
\end{aligned}
$$

Predicates

$$
\begin{aligned}
PRED ::= \; & TERM = TERM \\
| \; & TERM \in TERM \\
| \; & \mathbf{true} \\
| \; & \mathbf{false} \\
| \; & \neg \, PRED \\
| \; & PRED \wedge PRED \\
| \; & PRED \vee PRED \\
| \; & PRED \Rightarrow PRED \\
| \; & \exists \, SCHEMA \bullet PRED \\
| \; & \forall \, SCHEMA \bullet PRED.
\end{aligned}
$$

Terms

$$
\begin{array}{lll}
TERM ::= IDENT & & \text{– identifier.} \\
\mid \varnothing[TERM] & & \text{– null set.} \\
\mid \{TERM, \ldots, TERM\} & & \text{– extensive set.} \\
\mid \{\, SCHEMA \bullet TERM \,\} & & \text{– comprehension.} \\
\mid SDES & & \text{– schema designator.} \\
\mid \mathbb{P}\ TERM & & \text{– power-set.} \\
\mid (TERM, \ldots, TERM) & & \text{– tuple.} \\
\mid TERM \times \cdots \times TERM & & \text{– Cartesian product.} \\
\mid \theta\ WORD\ DECOR & & \text{– } \theta\text{-term.} \\
\mid TERM\ .\ IDENT & & \text{– selection.} \\
\mid TERM(TERM) & & \text{– function application.} \\
\mid \lambda\, SCHEMA \bullet TERM & & \text{– } \lambda\text{-term.} \\
\mid \mu\, SCHEMA \bullet TERM & & \text{– } \mu\text{-term.}
\end{array}
$$

Identifiers and Primitive classes

- Identifiers

$$IDENT ::= WORD\ DECOR$$

- Primitive classes

$$
\begin{array}{ll}
WORD & \text{– undecorated identifiers.} \\
DECOR & \text{– decoration.}
\end{array}
$$

Formal Methods

Topic#139

END!

Formal Methods

Topic#140
Modelling Scope

Modelling scope

- Nested scopes are introduced into Z specifications when global variables are defined and then used in the definition of schemas, and by use of the quantifiers $\forall$ and $\exists$ and the quantifiers-like constructs beginning λ, μ and $\{$.
- Because the same identifier may be introduced in several nested declarations, we need some way of describing which declaration is linked with each applied occurrence of an identifier.
- A solution is to work with signatures which contain not the identifiers themselves, but “names” consisting of an identifier tagged with the lexical level of its declarations.

Names

- A name is an identifier tagged with a level number. The outermost part of a specification is level 0, the variables of a schema are at level 1, and so on: higher numbers are needed e.g. for quantifiers.
- The operation *Tag* of tagging is a bijection because each name determines uniquely the identifier and level number. For convenience, a curried variant *tag* of the tagging function is introduced also.

$$[IDENT, NAME]$$

$$LEVEL \mathrel{\widehat=} \mathbb{N}$$

$$\begin{array}{|l} Tag : LEVEL \times IDENT \rightarrowtail NAME \\ tag : LEVEL \rightarrow IDENT \rightarrow NAME \\ \hline tag\ k\ x = Tag(k, x). \end{array}$$

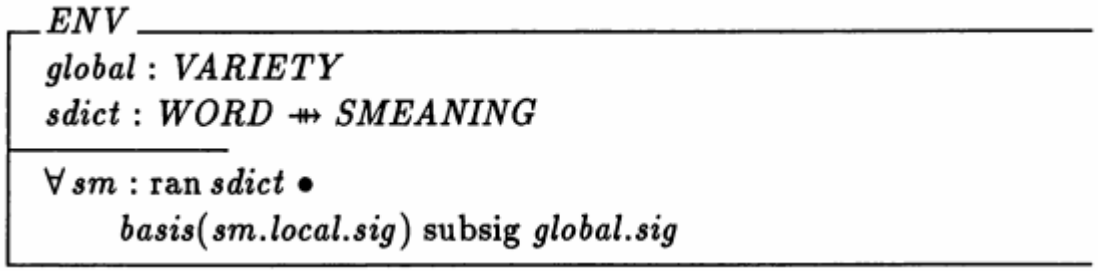
Rules to determine the name

- Two rules are used to determine the name to which any applied occurrence of an identifier refers:

1. Names at higher levels hide those at lower levels.
2. Variables hide given-set names at the same level.

Environments

- The global given-set names and variables are combined to make a global signature, and a class of models for this signature allows the specified relationship among these names to be recorded. This gives a variety *global*, which is made part of the environment.
- There is also dictionary *sdict* mapping each previously defined schema name to its definition, an object of type SMEANING:



- The meaning recorded for a schema name in the environment consists of a variety *local* and a sequence *fparam* which records the order of the formal generic parameters:

$$
\begin{array}{|l}
\textit{SMEANING} \\
\hline
local : VARIETY \\
fparam : \operatorname{seq} IDENT \\
\hline
fparam^{-1} \in locids(local.sig.given) \rightarrowtail \mathbb{N} \\
local.sig.given \cup local.sig.vars \subseteq basename \cup localname \\
local.sig \in \operatorname{dom} basis \\
\hline
\end{array}
$$

Operations on environments

- Conceptually, all specifications start with the empty environment *arid:*

$$
\begin{array}{|l}
\hline
arid : ENV \\
\hline
arid = \\
\quad \mu\, ENV \mid \\
\qquad global.sig = null\_sig \wedge \\
\qquad global.models = Struct(global.sig) \wedge \\
\qquad sdict = \emptyset. \\
\hline
\end{array}
$$

- Environments are extended by declaring new variables and given-set names, and defining new schemas. The commonest operation is to add new variables and given-set names with axioms relating them, and this is modelled by the function *enrich*:

$$
\begin{array}{|l}
enrich : ENV \times VARIETY \pfun ENV \\
\hline
enrich = \\
\quad \lambda\, \rho : ENV;\ V : VARIETY \mid \rho.global \ \mathrm{subvar}\ V \bullet \\
\quad \mu\, \rho' : ENV \mid \\
\qquad \rho'.global = V \land \\
\qquad \rho'.sdict = \rho.sdict.
\end{array}
$$

- The other way of extending an environment is by adding a new schema:

$$
\begin{array}{|l}
\hline
add\_schema : ENV \times WORD \times SMEANING \pfun ENV \\
\hline
add\_schema = \\
\quad \lambda\, \rho : ENV; w : WORD; sm : SMEANING \mid \\
\qquad w \notin \mathrm{dom}\ \rho.sdict \land \\
\qquad basis(sm.local.sig)\ \mathrm{subsig}\ \rho.global.sig \bullet \\
\quad \mu\, \rho' : ENV \mid \\
\qquad \rho'.global = \rho.global \land \\
\qquad \rho'.sdict = \rho.sdict \cup \{w \mapsto sm\}. \\
\hline
\end{array}
$$

Formal Methods

Topic#140

END!

Formal Methods

Topic#141
Declarations

Declarations

- Declarations introduce new variables and associate them with types.
- There are two elementary kinds of declaration:

1- One of these introduces a single variable, the type being given by a set-valued term.

2- The other introduces all the variables of a schema by means of a schema designator (SDES).

- Declarations may be combined with ‘;’ and such a composite declaration introduces all the variables introduced by either of its arguments: a vriables introduced by both arguments must have the same type in both.

$$\begin{aligned} DECL ::= & \; IDENT : TERM \\ | & \; SDES \\ | & \; DECL ; \; DECL. \end{aligned}$$

- As a convenience, the declaration

$$x, y, z : X$$

is defined to be syntactic sugar for the composite declaration

$$x : X; y : X; z : X.$$

- The main purpose of a declaration is to extablish a signature, but the terms giving the types of variables and the schemas which are included may also contain information given by axioms.
- This information is preserved by making the result of the semantic function a variety: the models in this variety are the ones which satisfy the axioms.

$$
\begin{array}{|l}
\hline
decl : ENV \to LEVEL \to DECL \nrightarrow VARIETY \\
\hline
decl\ \rho\ k\ [\![x : t]\!] \cong \\
\quad \mu\, tt : TMEANING \mid tt \cong term\ \rho\ k\ [\![t]\!] \bullet \\
\quad \mu\, a : TYPE \mid tt.type = powerT\ a \bullet \\
\quad new\_var(\rho.global, tag\ k\ x, a, tt.eval) \\
decl\ \rho\ k\ [\![sd]\!] \cong sdes\ \rho\ k\ [\![sd]\!] \\
decl\ \rho\ k\ [\![d_1; d_2]\!] \cong combine(decl\ \rho\ k\ [\![d_1]\!], decl\ \rho\ k\ [\![d_2]\!]).
\end{array}
$$

Formal Methods

Topic#141

END!

Formal Methods

Topic#142

Terms

Terms

- There are several ways of forming terms in Z:

$$
\begin{array}{lll}
TERM ::= & IDENT & \text{– identifier.} \\
| & \varnothing[TERM] & \text{– null set.} \\
| & \{TERM, \ldots, TERM\} & \text{– extensive set.} \\
| & \{\, SCHEMA \bullet TERM \,\} & \text{– comprehension*.} \\
| & SDES & \text{– schema designator.} \\
| & \mathbb{P}\ TERM & \text{– power-set.} \\
| & (TERM, \ldots, TERM) & \text{– tuple.} \\
| & TERM \times \cdots \times TERM & \text{– Cartesian product.} \\
| & \theta\ WORD\ DECOR & \text{– } \theta\text{-term.} \\
| & TERM\ .\ IDENT & \text{– selection.} \\
| & TERM(TERM) & \text{– function application.} \\
| & \lambda\, SCHEMA \bullet TERM & \text{– } \lambda\text{-term.} \\
| & \mu\, SCHEMA \bullet TERM & \text{– } \mu\text{-term*.}
\end{array}
$$

- In the two forms marked with an asterisk, the term following the dot may be omitted: the default is the ‘characteristic tuple’ of the schema before the dot, which is determined by its declaration part.
- If this has the simple form

$$x:t,$$

then the characteristic tuple is just the variable x. If it is a schema designator,

$$A'[t_1,\ldots,t_n],$$

the characteristic tuple is the θ-term θ A’.

- Finally, if the declaration has the form

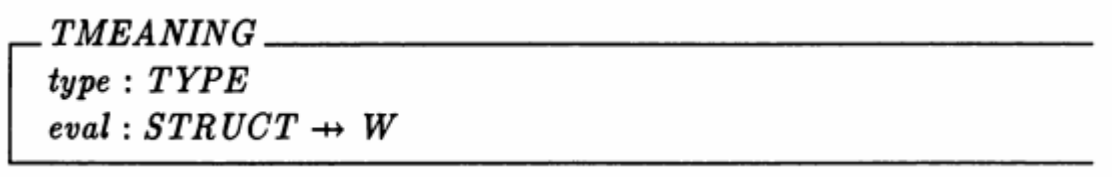
$$d_1; d_2; \ldots; d_n,$$

with $n \geq 2$, and $d_1, d_2, \ldots, d_n$ take the two simple forms above, then the characteristic tuple is

$$(t_1, t_2, \ldots, t_n),$$

where for $1 \leq i \leq n$, t_i is the characteristic tuple of d_i.

- The meaning of a term is a pair consisting of a type and a partial function giving the value of the term in models of its environment:



- The semantic function for terms is

$$term : ENV \to LEVEL \to TERM \nrightarrow TMEANING$$

Two important properties of terms

- Two important properties of terms are that:

-> the type of a term contains only given-set names from its environment

-> the value of a term is always an element of its type

- More formally, the properties are that if

$$\rho, k \vdash t :: a$$

$$\rho, k, M \vdash t \Rrightarrow u,$$

and

then

(i) a ϵ Type (p.global.sig.given), and

(ii) u E Carrier M.gset a.

Well-typing rules

- Identifiers

$$\frac{find\,(\rho.global.sig, k)\,x \cong vref(v) \quad \rho.global.sig.type(v) = a}{\rho, k \vdash x :: a}$$

$$\frac{find\,(\rho.global.sig, k)\,x \cong gref(G)}{\rho, k \vdash x :: \mathbb{P}\ G}$$

- Null set

$$\frac{\rho, k \vdash t :: \mathbb{P}\ a}{\rho, k \vdash \varnothing[t] :: \mathbb{P}\ a}$$

- Extensive set

$$\frac{\rho, k \vdash t_i :: a \quad (1 \le i \le n)}{\rho, k \vdash \{t_1, \ldots, t_n\} :: \mathbb{P}\ a}$$

- Comprehension

$$\frac{\rho_1 \cong enrich(\rho, schema\ \rho\ k\ [\![s]\!]) \quad \rho_1, k{+}1 \vdash t :: a}{\rho, k \vdash \{s \bullet t\} :: \mathbb{P}\ a}$$

- Schema designator

$$\frac{\rho, k \vdash \{ sd \bullet \theta sd \} :: a}{\rho, k \vdash sd :: a}$$

- Power-set

$$\frac{\rho, k \vdash t :: \mathbb{P}\, a}{\rho, k \vdash \mathbb{P}\, t :: \mathbb{P}(\mathbb{P}\, a)}$$

- Tuple

$$\frac{\rho, k \vdash t_i :: a_i \quad (1 \leq i \leq n)}{\rho, k \vdash (t_1, \ldots, t_n) :: a_1 \times \cdots \times a_n}$$

- Cartesian product

$$\frac{\rho, k \vdash t_i :: \mathbb{P}\, a_i \quad (1 \leq i \leq n)}{\rho, k \vdash t_1 \times \cdots \times t_n :: \mathbb{P}(a_1 \times \cdots \times a_n)}$$

- θ-term

$$\frac{locids((\rho.sdict\ A).local.sig.vars) \cong \{x_1, \ldots, x_n\} \quad \rho, k \vdash x_i(') :: a_i \quad (1 \leq i \leq n)}{\rho, k \vdash \theta A(') :: \langle x_1 : a_1;\ \ldots;\ x_n : a_n \rangle}$$

- Selection

$$\frac{\rho, k \vdash t :: \langle x_1 : a_1;\ \ldots;\ x_n : a_n \rangle}{\rho, k \vdash t.x_j :: a_j \quad (1 \leq j \leq n)}$$

- Function application

$$\frac{\rho, k \vdash t_1 :: \mathbb{P}(a \times a') \quad \rho, k \vdash t_2 :: a}{\rho, k \vdash t_1(t_2) :: a'}$$

- λ-term

$$\frac{t_1 = char\_tuple[\![s]\!] \quad \rho, k \vdash \{ s \bullet (t_1, t) \} :: a}{\rho, k \vdash \lambda s \bullet t :: a}$$

- μ-term

$$\frac{\rho, k \vdash \{ s \bullet t \} :: \mathbb{P}\ a}{\rho, k \vdash \mu s \bullet t :: a}$$

Environment rules

- Identifiers

$$\frac{\mathit{find}\,(\rho.\mathit{global.sig}, k)\,x \cong \mathit{vref}(v) \qquad M.\mathit{val}(v) = u}{\rho, k, M \vdash x \Rrightarrow u}$$

$$\frac{\mathit{find}\,(\rho.\mathit{global.sig}, k)\,x \cong \mathit{gref}(G) \qquad M.\mathit{gset}(G) = u}{\rho, k, M \vdash x \Rrightarrow u}$$

- Null set

$$\rho, k, M \vdash \varnothing[t] \Rrightarrow \mathit{null}$$

- Extensive set

$$\frac{\rho, k, M \vdash t_i \Rrightarrow u_i \quad (1 \le i \le n)}{\rho, k, M \vdash \{t_1, \ldots, t_n\} \Rrightarrow \mathit{rep}\,\{u_1, \ldots, u_n\}}$$

- Comprehension

$$\frac{\begin{array}{l}\rho_1 \cong \mathit{enrich}(\rho, \mathit{schema}\,\rho\,k\,[\![s]\!]) \\ \rho_1, k{+}1 \vdash t :: a \\ \mathrm{dom}\,uu = \mathit{extend}\,(\rho.\mathit{global.sig}, \rho_1.\mathit{global})\,M \\ \forall M' : \mathrm{dom}\,uu \bullet (\rho_1, k{+}1, M' \vdash t \Rrightarrow uu(M'))\end{array}}{\rho, k, M \vdash \{s \bullet t\} \Rrightarrow \mathit{filter}(\mathit{Carrier}\,M.\mathit{gset}\,a, \mathrm{ran}\,uu)}$$

- Schema designator

$$\frac{\rho, k, M \vdash \{\, sd \bullet \theta sd \,\} \Rrightarrow u}{\rho, k, M \vdash sd \Rrightarrow u}$$

- Power-set

$$\frac{\rho, k, M \vdash t \Rrightarrow u}{\rho, k, M \vdash \mathbb{P}\, t \Rrightarrow power(u)}$$

- Tuple

$$\frac{\rho, k, M \vdash t_i \Rrightarrow u_i \quad (1 \leq i \leq n)}{\rho, k, M \vdash (t_1, \ldots, t_n) \Rrightarrow tuple\, \langle u_1, \ldots, u_n \rangle}$$

- Cartesian product

$$\frac{\rho, k, M \vdash t_i \Rrightarrow u_i \quad (1 \leq i \leq n)}{\rho, k, M \vdash t_1 \times \cdots \times t_n \Rrightarrow cproduct\, \langle u_1, \ldots, u_n \rangle}$$

- θ-term
- Selection
- Function application
- λ-term
- μ-term

$$\frac{locids((\rho.sdict\ A).local.sig.vars) \cong \{x_1, \ldots, x_n\} \quad \rho, k, M \vdash x_i(') \Rightarrow u_i \quad (1 \le i \le n)}{\rho, k, M \vdash \theta A(') \Rightarrow binding\,\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}}$$

$$\frac{\rho, k \vdash t :: \langle x_1 : a_1;\ \ldots;\ x_n : a_n \rangle \quad \rho, k, M \vdash t \Rightarrow binding\,\{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}}{\rho, k, M \vdash t.x_j \Rightarrow u_j \quad (1 \le j \le n)}$$

$$\frac{\begin{array}{l}\rho, k, M \vdash t_1 \Rightarrow u \\ \rho, k, M \vdash t_2 \Rightarrow v \\ tuple\,\langle v, v' \rangle \in u \\ \forall\, w : W \bullet tuple\,\langle v, w \rangle \in u \Rightarrow w = v'\end{array}}{\rho, k, M \vdash t_1(t_2) \Rightarrow v'}$$

$$\frac{t_1 = char\_tuple\,[\![s]\!] \quad \rho, k, M \vdash \{\, s \bullet (t_1, t) \,\} \Rightarrow u}{\rho, k, M \vdash \lambda\, s \bullet t \Rightarrow u}$$

$$\frac{\rho, k, M \vdash \{\, s \bullet t \,\} \Rightarrow rep\,\{u\}}{\rho, k, M \vdash \mu\, s \bullet t \Rightarrow u}$$

Formal Methods

Topic#142

END!

Formal Methods

Topic#143
Predicates

Predicates

- Elementary predicates are of three kinds:

-> equality between terms of the same type

-> the membership predicate between a term of type a and another of type P a

-> the logical constants **true** and **false**

- These may be combined with all the usual logical connectives, and qualified with ‘for all’ and ‘there exists’:

$$
\begin{aligned}
PRED ::= \; & TERM = TERM \\
| \; & TERM \in TERM \\
| \; & \textbf{true} \\
| \; & \textbf{false} \\
| \; & \neg \, PRED \\
| \; & PRED \wedge PRED \\
| \; & PRED \vee PRED \\
| \; & PRED \Rightarrow PRED \\
| \; & \exists \, SCHEMA \bullet PRED \\
| \; & \forall \, SCHEMA \bullet PRED.
\end{aligned}
$$

- The other kinds of predicates--infix relation symbols, unique quantifiers, and so on--can be defined as syntactic sugar for combinations of these few forms.
- Of course, these forms are not themselves independent: for example,

‘$\Rightarrow$’ can be defined in terms of ‘$\neg$ ’ and ‘$\vee$’:

$$P \Rightarrow Q \;\widehat{=}\; \neg P \vee Q.$$

- A predicate is modelled in the semantics by the set of structures which satisfy it. For the moment, we deal with partially defined terms by fixing a semantics for the equality sign which makes

$$t_1 = t_2$$

true if t1 and t2 are both defined with equal values, and false otherwise, and fixing a similar semantics for the membership sign.

- The rest of the semantics of predicates is a completely classical truth-defintion.
- The logical connectives are made to correspond to elementary set-theoretic operations on the sets of structures; $\wedge$ to intersection, $\vee$ to union, and so on.

- The semantics of quantifiers is given in terms of the functions *restrict* and *extend*, which link the environment with the result of enriching it with the bound variables of the quantifier.

$$
\begin{array}{l}
pred : ENV \to LEVEL \to PRED \to \mathbb{P}\, STRUCT \\
\hline
pred\ \rho\ k\ [\![t_1 = t_2]\!] \cong \\
\quad \mu\, tt_1, tt_2 : TMEANING \mid \\
\qquad tt_1 \cong term\ \rho\ k\ [\![t_1]\!] \land \\
\qquad tt_2 \cong term\ \rho\ k\ [\![t_2]\!] \land \\
\qquad tt_1.type = tt_2.type \bullet \\
\quad \{ M : \rho.global.models \mid \\
\qquad M \in (\mathrm{dom}\ tt_1.eval) \cap (\mathrm{dom}\ tt_2.eval) \land \\
\qquad tt_1.eval(M) = tt_2.eval(M) \} \\
pred\ \rho\ k\ [\![t_1 \in t_2]\!] \cong \\
\quad \mu\, tt_1, tt_2 : TMEANING \mid \\
\qquad tt_1 \cong term\ \rho\ k\ [\![t_1]\!] \land \\
\qquad tt_2 \cong term\ \rho\ k\ [\![t_2]\!] \land \\
\qquad powerT\ tt_1.type = tt_2.type \bullet \\
\quad \{ M : \rho.global.models \mid \\
\qquad M \in (\mathrm{dom}\ tt_1.eval) \cap (\mathrm{dom}\ tt_2.eval) \land \\
\qquad tt_1.eval(M) \in tt_2.eval(M) \} \\
pred\ \rho\ k\ [\![\mathbf{true}]\!] \cong \rho.global.models \\
pred\ \rho\ k\ [\![\mathbf{false}]\!] \cong \varnothing \\
pred\ \rho\ k\ [\![\neg\ p]\!] \cong \rho.global.models \setminus pred\ \rho\ k\ [\![p]\!] \\
pred\ \rho k [\![p_1 \land p_2]\!] \cong pred\ \rho\ k\ [\![p_1]\!] \cap pred\ \rho\ k\ [\![p_2]\!] \\
pred\ \rho\ k\ [\![p_1 \lor p_2]\!] \cong pred\ \rho\ k\ [\![p_1]\!] \cup pred\ \rho\ k\ [\![p_2]\!] \\
pred\ \rho\ k\ [\![p_1 \Rightarrow p_2]\!] \cong \\
\quad \rho.global.models \setminus (pred\ \rho\ k\ [\![p_1]\!] \setminus pred\ \rho\ k\ [\![p_2]\!])
\end{array}
$$

$$
\begin{array}{l}
pred\ \rho\ k\ [\![\exists\ s \bullet p]\!] \cong \\
\quad \mu\, \rho_1 : ENV \mid \rho_1 \cong enrich(\rho, schema\ \rho\ k\ [\![s]\!]) \bullet \\
\quad restrict\ \rho.global.sig\ (pred\ \rho_1\ (k{+}1)\ [\![p]\!]) \\
pred\ \rho\ k\ [\![\forall\ s \bullet p]\!] \cong \\
\quad \mu\, \rho_1 : ENV \mid \rho_1 \cong enrich(\rho, schema\ \rho\ k\ [\![s]\!]) \bullet \\
\quad \{ M : \rho.global.models \mid \\
\qquad extend\ (\rho.global.sig, \rho_1.global)\ M \subseteq pred\ \rho_1\ (k{+}1)\ [\![p]\!] \}.
\end{array}
$$

Formal Methods

Topic#143

END!

Formal Methods

Topic#144
Schema Bodies

Schema bodies

- A schema body consists of a declaration constrained by a predicate.
- Schema bodies may appear in many places in a Z specification: after quantifiers, after λ, and so on, as well as in the definition of schemas.
- The meaning of a schema body is a variety--it is the one which results from selecting those models of the declaration which also satisfy the predicate:

$$SCHEMA ::= DECL \mid PRED$$

$$schema : ENV \to LEVEL \to SCHEMA \nrightarrow VARIETY$$

$$\begin{array}{l} schema\ \rho\ k\ [\![d \mid p]\!] \cong \\ \quad \mu\, \rho_1 : ENV \mid \rho_1 \cong enrich(\rho, decl\ \rho\ k\ [\![d]\!]) \bullet \\ \quad \mu\, VARIETY \mid \\ \qquad sig = \rho_1.global.sig \land \\ \qquad models \cong pred\ \rho_1\ (k+1)\ [\![p]\!]. \end{array}$$

- The predicate part of a schema body may be omitted in any context; the default is the logical constant **true**.
- Also, a list of predicates may be written in place of a single predicate: this is equivalent to the conjunction of the predicates.

Formal Methods

Topic#144

END!

Formal Methods

Topic#145
Schema Designators

Schema designators

- A schema designator is an applied occurrence of a schema name.
- The components of the named schema can be subjected to systematic decoration, and actual parameters are substituted for the formal generic parameters of the definition.
- So, syntactically, a schema designator consists of a word naming the schema, a decoration--possibly blank--to be applied to the component names, and a list of set-valued terms which are the actual generic parameters:

$$SDES ::= WORD\ DECOR[TERM, \ldots, TERM].$$

- the meaning of a schema designator is a variety: it has all the given-set names and variables of the global part of the environment, together with the local varaiables of the schema.
- The model class represents information from three sources:

-> First, the global part of the specification

-> Second, the axioms of the schema itself

-> Third, the actual parameters of the schema designator

Formal Methods

Topic#145

END!

Formal Methods

Topic#146
Schema Expressions

Schema expressions

- Schema expressions are built up from schema bodies and schema designators by various combining operators:

$$
\begin{aligned}
SEXP ::= & \ \textbf{schema}\ SCHEMA\ \textbf{end} \\
| \ & \ SDES \\
| \ & \ \neg\ SEXP \\
| \ & \ SEXP \wedge SEXP \\
| \ & \ SEXP \vee SEXP \\
| \ & \ SEXP \Rightarrow SEXP \\
| \ & \ SEXP \upharpoonright SEXP \\
| \ & \ SEXP \setminus (IDENT, \ldots, IDENT) \\
| \ & \ \exists\, SCHEMA \bullet SEXP \\
| \ & \ \forall\, SCHEMA \bullet SEXP.
\end{aligned}
$$

- Schema expressions can be combined with any of the connectives of the propositional calculus, and here we show '$\neg$', '$\wedge$' and '$\vee$' as examples.
- The negation operator takes the complement of the class of models of its argument:

$$
\begin{array}{|l}
\hline
negate : VARIETY \rightarrow VARIETY \\
\hline
negate(\theta VARIETY) = \\
\quad \mu\, VARIETY' \mid \\
\qquad sig' = sig \wedge \\
\qquad models' = Struct(sig) \setminus models.
\end{array}
$$

- Implication is defined in terms of the function *imply*, consistency with the global part of the specification again being obtained by using *combine*.

$$imply : VARIETY \times VARIETY \nrightarrow VARIETY$$

$$imply(V_1, V_2) \cong disjoin(negate(V_1), V_2).$$

- Now, some hiding operations, the projection operator ‘|’ and the operator ‘\’, which allows particular variables to be hidden, is defined in terms of the function *hide*:

$$
\begin{array}{|l}
hide : VARIETY \times \mathbb{P}\, NAME \rightarrow VARIETY \\
hide\_sig : SIG \times \mathbb{P}\, NAME \rightarrow SIG \\
\hline
hide(\theta VARIETY, S) = \\
\quad \mu\, VARIETY' \mid \\
\qquad sig' = hide\_sig(sig, S) \wedge \\
\qquad models' = restrict\ sig'\ (\!|models|\!) \\
hide\_sig(\theta SIG, S) = \\
\quad \mu\, SIG' \mid \\
\qquad sig'.given = sig.given \wedge \\
\qquad sig'.vars = sig.vars \setminus S \wedge \\
\qquad sig'.type = sig'.vars \lhd sig.type.
\end{array}
$$

- The universal quantifier likewise corresponds to removing the variables of a schema from the signature of a schema expression and universally quantifying them in the axiom part.
- The classical equivalance between the universal and existential quantifiers continues to hold in the schema calculus.

$$\forall s \bullet se \mathrel{\widehat{=}} \neg (\exists s \bullet \neg se),$$

where the quantifiers and negation signs have their interpretation as schema operations.

- However, we give here an explicit definition of the universal quantifier in terms of a function *univ*:

$$
\begin{array}{|l}
univ : VARIETY \times VARIETY \times \mathbf{P}\, NAME \nrightarrow VARIETY \\
\hline
univ(\theta\, VARIETY, S) \cong \\
\quad \mu\, VARIETY' \mid \\
\qquad sig' = hide\_sig(sig, S) \wedge \\
\qquad models' = \{\, M' : Struct(sig') \mid \\
\qquad\quad \{\, M : Struct(sig) \mid restrict\ sig'\ M = M' \,\} \subseteq models \,\}.
\end{array}
$$

Formal Methods

Topic#146

END!

Formal Methods

Topic#147
Specifications

Specifications

- The smallest specification is a definition, which may introduce some global given-set names, introduce some global variables with some axioms, or define a new schema.
- Definitions may joined with in, and then the names introduced by the first are added to the environment before the second is considered.

$$
\begin{aligned}
SPEC ::= & \ \textbf{given}\ IDENT, \ldots, IDENT \\
| & \ \textbf{let}\ SCHEMA\ \textbf{end} \\
| & \ \textbf{let}\ WORD[IDENT, \ldots, IDENT] = SEXP \\
| & \ SPEC\ \textbf{in}\ SPEC.
\end{aligned}
$$

- A specification is evaluated in an initial environment--this might contain the definitions of the standard mathematical tool-kit, and it produces a new environment, enriched with the new objects defined in the specification.

$$
\begin{array}{l}
spec : ENV \rightarrow SPEC \nrightarrow ENV \\
\hline
spec\ \rho\ [\![\textbf{given}\ x_1, \ldots, x_n]\!] \cong \\
\quad enrich(\rho, new\_givens(\rho.global, tag\ 0\ (\!|\{x_1, \ldots, x_n\}|\!))) \\
spec\ \rho\ [\![\textbf{let}\ s\ \textbf{end}]\!] \cong enrich(\rho, schema\ \rho\ 0\ [\![s]\!]) \\
spec\ \rho\ [\![\textbf{let}\ w[x_1, \ldots, x_n] = se]\!] \cong \\
\quad \mu\ \rho_1 : ENV \mid \rho_1 \cong \\
\qquad enrich(\rho, new\_givens(\rho.global, tag\ 1\ (\!|\{x_1, \ldots, x_n\}|\!))) \bullet \\
\quad \mu\ sm : SMEANING \mid \\
\qquad sm.local \cong sexp\ \rho_1\ 1\ [\![se]\!] \wedge \\
\qquad sm.fparam = \langle x_1, \ldots, x_n \rangle \bullet \\
\quad add\_schema(\rho, w, sm) \\
spec\ \rho\ [\![z_1\ \textbf{in}\ z_2]\!] \cong \mu\ \rho_1 : ENV \mid \rho_1 \cong spec\ \rho\ [\![z_1]\!] \bullet spec\ \rho_1\ [\![z_2]\!].
\end{array}
$$

- The declaration of global given-set names or global variables enriches the global variety of the environment.
- Because of the semantics of declarations and the definition of *new-givens*, it is meaningless to declare the same global variable or given-set name twice.
- A schema definition adds a new schema name to the dictionary.
- The formal generic parameters may be used in the defining schema expression, and their order is preserved in the *fparam* sequence for matching with actual parameters in schema designators.

- The operation *new-givens* adds new given-set names to a variety in much the same way as *new-var*.
- the definition is slightly simpler because no type information is associated with given-set names.

$$
\begin{array}{|l}
new\_givens : VARIETY \times \mathbb{F}\, NAME \rightarrow VARIETY \\
\hline
new\_givens = \\
\quad \lambda\, VARIETY_0; S : \mathbb{F}\, NAME \mid \\
\qquad S \cap sig_0.given = \varnothing \bullet \\
\quad \mu\, VARIETY \mid \\
\qquad sig.given = sig_0.given \cup S \wedge \\
\qquad sig.vars = sig_0.vars \wedge \\
\qquad sig.type = sig_0.type \wedge \\
\qquad models = \\
\qquad\quad \{ M : Struct(sig) \mid restrict\ sig_0\ M \in models_0 \}.
\end{array}
$$

Formal Methods

Topic#147

END!

CS636 Formal Methods

16.1

Topics Covered

Well-Formedness

Well-Formedness

- **The property of a model, specification, or expression adhering to the syntactic rules and constraints defined by a formal language or formalism.**
- ***When a model or specification is well-formed, it means that it is structurally correct and conforms to the syntax and grammar of the formal language being used.***

Well-Formedness

- **Ensuring well-formedness is essential because formal methods rely on rigorous mathematical techniques for analysis and verification, and any errors or ambiguities in the specification could lead to incorrect results or misinterpretations.**
- **Once a model is well-formed, formal analysis techniques can be applied to check properties like consistency, correctness, safety, liveness, and more, thereby enhancing the reliability and correctness of complex systems.**

Completeness

- **Property of a formal model or specification that indicates it includes all the necessary and relevant information to describe the system or problem being modeled.**
- **A formal model is considered complete when it fully captures all the intended behaviors, states, and constraints of the system in question.**

Completeness

- **Completeness is essential because it ensures that no critical aspects or essential requirements of the system are omitted from the formal representation.**
- **If a formal model is not complete, it may lead to incorrect or insufficient conclusions when performing formal analysis or verification.**

Completeness

- **A complete formal model should cover all possible scenarios, edge cases, and interactions between components of the system.**
- **It should also encompass all relevant variables, transitions, and invariants that define the system's behavior accurately.**

Completeness

- **Achieving completeness in formal models can be challenging, especially for complex systems, as it requires a thorough understanding of the system's requirements and potential behaviors.**
- **Additionally, modeling all aspects of a system with complete accuracy can be a time-consuming process.**

Completeness

- **Practitioners often strive to strike a balance between the desired level of completeness and the practical limitations, aiming to model the most critical parts of the system accurately to ensure the effectiveness of formal analysis and verification efforts.**
- **It's important to note that achieving complete formal models is not always feasible or necessary, but making the model as complete as possible within reasonable bounds is crucial for reliable results in formal methods applications.**

CS636 Formal Methods

16.2

Topics Covered

Robustness

Robustness

- **Ability to provide accurate and reliable results in the face of various challenges, uncertainties, and complexity associated with modeling and verifying systems.**
- ***A formal method is considered robust if it can handle different scenarios, edge cases, and potential sources of errors while still producing meaningful and trustworthy outcomes.***

Robustness Evaluation Aspects

- ***Completeness: A robust formal method should be able to handle complex systems and specifications while remaining as complete as possible, capturing all essential behaviors and requirements.***
- ***Scalability: The method should be scalable, meaning it can handle large and complex systems efficiently, without becoming computationally infeasible or impractical.***
- ***Expressiveness: A robust formal method should support a rich set of constructs and language features, allowing modelers to represent a wide range of system behaviors accurately.***

Robustness Evaluation Aspects

- ***Automation: The level of automation provided by the method is crucial for its robustness. Automated tools for model checking, theorem proving, and other analyses reduce the risk of human error and make the verification process more reliable.***
- ***Soundness and Completeness of Analysis: The method should have sound theoretical foundations, meaning that any conclusion drawn from the analysis is guaranteed to be correct. It should also be as complete as possible, ensuring that it explores all relevant aspects of the model.***
- ***Handling Abstraction and Refinement: A robust formal method should allow users to work with models at various levels of abstraction and support refinement techniques to iteratively enhance the model's precision.***

Robustness Evaluation Aspects

- ***Handling Uncertainty: Real-world systems often have uncertainties and partial knowledge. A robust formal method should handle these uncertainties gracefully and provide insights into the behavior of the system under different conditions.***
- ***Tool Support: The availability and maturity of tools supporting the formal method are crucial for its robustness. Well-maintained and actively developed tools can improve the usability and effectiveness of the method.***
- ***Applicability to Different Domains: A truly robust formal method should be applicable to a wide range of domains, from hardware and software systems to safety-critical and real-time systems.***

CS636 Formal Methods

16.3

Topics Covered

Proofs

Proof

- **A "proof" refers to a rigorous demonstration or argument that establishes the correctness or validity of a particular claim or statement within a formal system.**
- **It involves applying formal rules of logic and reasoning to show that a property, specification, or assertion holds true for a given model or system under consideration.**

Proof

- **Proofs play a central role in formal methods for various purposes, including:**
  - **Verification: Proofs are used to verify that a system meets its desired properties or requirements. For example, in software verification, a proof can demonstrate that a program satisfies a specified safety property, meaning it will never exhibit certain undesirable behaviors.**
  - **Correctness: Proofs are employed to demonstrate the correctness of algorithms, protocols, or systems. By providing a mathematical guarantee of correctness, formal methods can offer strong assurances of system reliability.**
  - **Refinement: Proofs are used in the process of refining abstract models to more concrete designs. These proofs ensure that the implementation correctly refines the higher-level specification.**

Proof

- **Proofs play a central role in formal methods for various purposes, including:**
  - **Consistency: Proofs can be used to check the consistency of a formal model. A consistent model does not contain conflicting statements or logical contradictions.**
  - **Completeness: In formal systems, proofs can establish the completeness of logical systems, showing that all true statements within the system can be proven.**

Consistency of a formal methods

- **Consistency refers to a property of a formal system or a formal model where it does not contain conflicting statements, logical contradictions, or incompatible assumptions.**
- ***If a system is inconsistent, it could produce conflicting or unreliable outcomes when subjected to formal analysis, verification, or reasoning.***

Consistency of a formal methods

- **When inconsistency is detected in a formal system or model, it indicates a flaw in the logic or definitions. In such cases, it is crucial to identify and resolve the conflicting statements to restore the system's consistency.**
- **Often, inconsistencies are uncovered during formal verification or automated theorem proving, and addressing them is an essential step in ensuring the correctness and reliability of the formal methods application.**

Consistency of a formal methods

- **Consistency ensures that the results obtained through formal methods are reliable, and that the system's behavior is well-defined and free from paradoxes or contradictions.**
- **It is one of the key aspects that distinguish formal methods from informal methods of reasoning, providing a solid foundation for mathematically rigorous analysis and verification.**

Thank You