# WEEK 7 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 $P_1, P_2, \ldots, P_n$, yields another proposition Q.
- An argument provides support or evidence in favor of one of the others.
- An argument $P_1, P_2, \ldots, P_n \vdash Q$ is said to be valid if Q is TRUE whenever all the premises $P_1, P_2, \ldots, P_n$ 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 |

- ➢ 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.

| $p$ | $q$ | $[(p$ | $\rightarrow$ | $q)$ | $\wedge$ | $\neg$ | $q]$ | $\rightarrow$ | $\neg$ | $p$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

- ➢ Lets determine the validity of the following argument:
- ➢ **p → q, ¬ q ⊢ ¬p.**

An argument which is not valid is called fallacy.

Eg: fallacy argument

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

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 |

| $p$ | $q$ | $p \rightarrow q$ |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

## 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

| Inputs | | Output |
|---|---|---|
| A | B | $Y = A \cdot B$ |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### OR Operator

| Inputs | | Outputs |
|---|---|---|
| A | B | $Y = A + B$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

### IMPLICATION Operator

| p | q | p ↔ q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | T |

### Equivalence Operator

| p | q | p ⇔ q |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Exclusive OR Operator**

AND

| Inputs | | Output |
|---|---|---|
| A | B | Z |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Inclusive OR

| Inputs | | Output |
|---|---|---|
| A | B | Z |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Exclusive OR

| Inputs | | Output |
|---|---|---|
| A | B | Z |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Negation Operator**

| P | ¬P |
|---|---|
| T | F |
| F | T |
| UNDEFINED | UNDEFINED |

**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
- 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.
- – 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:
- ∃n ∈ ℕ ● 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.
- –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
- ☐ $\exists! x \in Cats \bullet G(x)$
- ☐ This would mean There is one and only one cat that is green.

**WEEK 8**

**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

**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.

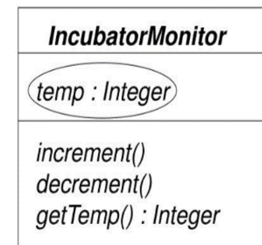
**The UML Specification**

- 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.

**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 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.

**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** $temp > MIN$

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

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

### Implicit Function

- In this method, we use a pre- and post-condition 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** $z = x + y$

### 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: **inv**: $State \rightarrow \mathbb{B}$
- 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:

**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.

### 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:

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

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

**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:
  **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**.

**The nil Value**

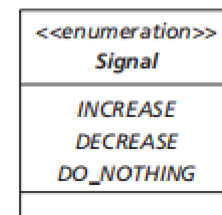
- It is common in the programming world for a value to be undefined. VDM-SL allows for this concept by including the possibility of a term or expression having the value **nil**, meaning that it is undefined.
- Of course, if we want to allow for this possibility, then we need to slightly modify the type 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.

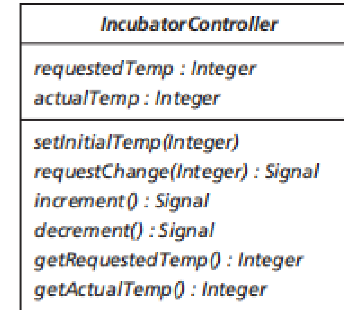
**Improving the Incubator System**

- 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**

**UML specification of the Signal type**




**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**

**state** *IncubatorController* **of**
*requestedTemp* : $[\mathbb{Z}]$
*actualTemp* : $[\mathbb{Z}]$

**Invariant comprising two conjuncts**

$$\textbf{inv mk-}IncubatorController\ (r, a)\ \underline{\Delta}\ (MIN \leq r \leq MAX \lor r = \textbf{nil}) \land (MIN \leq a \leq MAX \lor a = \textbf{nil})$$

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.

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

**Use of inRange Function in the invariant**

$$\textbf{inv mk-}IncubatorController\ (r, a)\ \underline{\Delta}\ (inRange(r) \lor r = \textbf{nil}) \land (inRange(a) \lor a = \textbf{nil})$$

**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:

$$\textbf{init mk-}IncubatorController\ (r, a)\ \underline{\Delta}\ r = \textbf{nil} \land a = \textbf{nil}$$

**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 : ℤ)
ext wr  actualTemp :  [ℤ]
pre     inRange(tempIn) ∧ actualTemp = nil
post    actualTemp = tempIn
```

### requestChange operation

- The operation is specified as follows:

```
requestChange(tempIn : ℤ) signalOut : Signal
ext wr requestedTemp : [ℤ]
    rd actualTemp : [ℤ]
pre     inRange(tempIn) ∧ actualTemp ≠ nil
post    requestedTemp = tempIn ∧
                 (tempIn > actualTemp ∧ signalOut = <INCREASE>
                 ∨ tempIn < actualTemp ∧ signalOut = <DECREASE>
                 ∨ tempIn = actualTemp ∧ signalOut = <DO_NOTHING>)
```

### Increment operation

- The operation is specified as follows:
- The decrement operation is similar and does not require further explanation.

```
increment () signalOut : Signal
ext rd requestedTemp : [ℤ]
    wr actualTemp : [ℤ]
pre     actualTemp < requestedTemp ∧ actualTemp ≠ nil ∧ requestedTemp ≠ nil
post    actualTemp = actualTemp~ + 1 ∧
        (actualTemp < requestedTemp ∧ signalOut = <INCREASE>
           ∨ actualTemp = requestedTemp ∧ signalOut = <DO_NOTHING>)
```

### Read Operations

#### requested temperature

```
getRequestedTemp() currentRequested : [ℤ]
ext rd    requestedTemp : [ℤ]
pre   TRUE
post  currentRequested = requestedTemp
```

#### actual temperature

```
getActualTemp() currentActual : [ℤ]
ext rd    actualTemp : [ℤ]
pre   TRUE
post  currentActual = actualTemp
```

### Template for VDM-SL Specifications ★

- Generalized template for a VDM-SL specification:
- Not every clause would necessarily appear in every specification.

### 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) Δ (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) Δ r = nil ∧ a = nil
end
```

## WEEK 10

### Sets for System Modelling

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

### Sets in VDM-SL

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

### Set Declaration

**types**

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

**importantDays: Day-set**

**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}

**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}
- 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} = { }
- Comprehension: A set is defined by means of an expression and/or a test that each element in the set must satisfy.
- someNumbers = {x | x ∈ {2,…,6} ● isEven(x)}
- 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.
- 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 $\{x \mid x: \mathbb{N} \bullet 1 \le x \le 10\}$

  **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 = $\{x \mid x: \mathbb{Z} \bullet x < 0\}$
- The above set is not permissible in VDM-SL

**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.

**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>}

**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>}

**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>}

**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}

**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

  $J \cap K = K \cap J$

  $J \cup K = K \cup J$
- **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 \ k = {<WED>, <SUN>}

  k \ j = {<FRI>}

**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

**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} ⊆ {a, b, c, d, e, f } **… TRUE**

  {a, b, c, d, e, f } ⊆ {a, d, e} … **FALSE**
- **Returns TRUE if both sets are equal**

  {a, d, e} ⊆ {d, a, e} … **TRUE**

**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**

**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

**UML representation**

| *PatientRegister* |
|---|
| *reg: Patient [*]* |
| *addPatient(Patient)*<br>*removePatient(Patient)*<br>*getPatients( ): Patient[*]*<br>*isRegistered (Patient): Boolean*<br>*numberRegistered():Integer* |

**types**

Patient = TOKEN

**values=**

LIMIT: ℕ = 200

**state** PatientRegister **of**

reg: Patient-**set**

**inv mk**-PatientRegister (r) ≜ **card** r ≤ LIMIT

**inv mk**-PatientRegister (r) ≜ r = { }

**end**

*1.* addPatient (patientIn: Patient)

**ext wr** reg: Patient-**set**

**pre** patientIn ∉ reg ∧ **card** reg < LIMIT

**post** reg = $\overline{reg}$ ∪ {patientIn}

*2.* removePatient (patientIn: Patient)

**ext wr** reg: Patient-**set**

**pre** patientIn ∈ reg

**post** reg = $\overline{reg}$ \ {patientIn}

*3.* getPatients ( ) output: Patient-**set**

**ext rd** reg: Patient-**set**

**pre** TRUE

**post** output = reg

*4.* isRegistered (patientIn: Patient) query: 𝔹

**ext rd** reg: Patient-**set**

**pre** TRUE

**post** query ⇔ patientIn ∈ reg

*5.* numberRegistered ( ) total: ℕ

**ext rd** reg: Patient-**set**

**pre** TRUE

**post** total = **card** reg

**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.
- Give Permission: records the fact that an aircraft has been granted permission to land at the airport.
- Record Landing: records an aircraft as having landed at the airport.
- Record TakeOff: records an aircraft as having taken off from the airport.

- Get Permission: returns the aircrafts currently recorded as having permission to land
- Get Landed: returns the aircrafts currently recorded as having landed
- Number Waiting: returns the number of aircrafts granted permission to land but not yet landed.

| *Airport* |
|---|
| *permission: Aircraft* [*]<br>*landed: Aircraft* [*] |
| *givePermission(Aircraft)*<br>*recordLanding(Aircraft)*<br>*recordTakeOff(Aircraft)*<br>*getPermission( ): Aircraft* [*]<br>*getLanded( ): Aircraft* [*]<br>*numberWaiting(): Integer* |

➢ **types**

Aircraft = TOKEN
**state** Airport **of**
permission: Aircraft**-set**
landed: Aircraft **-set**
**inv mk-**Airport(p, l) $\triangleq$ l ⊆ p
**init mk-**Airport (p, l) $\triangleq$ p = { } ∧ l = { }
**end**

➢ givePermission (craftIn: Aircraft)

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

➢ recordLanding (craftIn: Aircraft)

**ext rd** permission: Aircraft **-set**
**wr** landed: Aircraft **-set**
**pre** craftIn ∈ permission ∧ craftIn ∉ landed
**post** landed = $\overline{\text{landed}}$ ∪ {craftIn}

➢ recordTakeOff (craftIn: Aircraft)

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

➢ getPermission ( ) out: Aircraft **-set**

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

➢ getLanded ( ) out: Aircraft **-set**

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

➢ numberWaiting( ) total: ℕ

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

WEEK 11

**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]

- 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

**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}

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]

- 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 ]

- 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.

**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]

**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]

**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.
- 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].

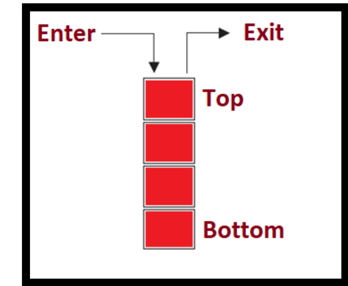
**Sequence Type in VDM-SL**

- 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 : ℤ*
  - convoy : SpaceCraft*

**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)*<br>*pop(): Element*<br>*isEmpty(): Boolean* |




- **types**
  - Element = TOKEN

  **state** Stack **of**
  stack : Element*
  **init mk-**Stack(s) △ s = []
  **end**
- push(itemIn : Element)
  **ext wr** stack : Element*
  **pre** TRUE
  **post** stack = [itemIn] ^ $\overline{stack}$
- pop() itemRemoved : Element
  **ext wr** stack : Element*
  **pre** stack ≠ []
  **post** stack = **tl** $\overline{stack}$ ^ itemRemoved = **hd** $\overline{stack}$
- isEmpty() query : 𝔹
  **ext rd** stack : Element*
  **pre** TRUE
  **post** query⇔stack = []

| *Airport* |
|---|
| *permission: Aircraft [*]*<br>*landed: Aircraft [*]* |
| *givePermission(Aircraft)*<br>*recordLanding(Aircraft)*<br>*recordTakeOff(Aircraft)*<br>*getPermission( ): Aircraft [*]*<br>*getLanded( ): Aircraft [*]*<br>*numberWaiting(): Integer* |

**UML Representation**

- **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**

- isUnique(seqIn : Aircraft*) query : $\mathbb{B}$

**pre** TRUE
**post** query⇔$\forall i_1, i_2 \in$ **inds** seqIn ● $i_1 \neq i_2 \Rightarrow seqIn(i_1) \neq seqIn(i_2)$

- inv mk-Airport2 (p,l,c) $\triangleq$ l ⊆ p
  ∧ elems c ⊆ p
  ∧ elems c ∩ l = { }
  ∧ isUnique(c)
- 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 = $\overline{\text{circling}}$ ^ [craftIn]

- recordLanding( )

**ext wr** circling : Aircraft*
**wr** landed : Aircraft-set
**pre** circling ≠ []
**post** landed = $\overline{\text{landded}}$ ∪ { **hd** $\overline{\text{circling}}$ } ∧ circling = **tl** $\overline{\text{circling}}$

**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
- 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))
- find(sequenceIn : Element*, element : Element) position : ℕ

**pre** element ∈ **elems** sequenceIn
**post** sequenceIn(position) = element

- findFirst(sequenceIn : Element*, element : Element) position : ℕ

**pre** element ∈ sequenceIn
**post** sequenceIn(position) = element ∧ ∀i ∈ **inds** sequenceIn ● sequenceIn (i) = element ⇒ position ≤ i

WEEK 12

**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

**TypeName :: fieldname1 : Type1**
**fieldname2 : Type2**
**:**

'::' symbol is called "Composed of" Individual values are called fields

**Example:**

**Time:: hour:** ℕ
**minute:** ℕ
**second:** ℕ

**Composite Object Operators**

- **make function: creates new object of given composite type**
  - **mk-CompositeObjectTypeName (parameter list)**
- inv mk-Airport2 (p,l,c) $\triangleq$ l ⊆ p ∧ elems c ⊆ p ∧ elems c ∩ l = { } ∧ isUnique(c)
- **mk**-Airport2: Aircraft-**set** x Aircraft-**set** x Aircraft* →Airport2
- **mk-**Time: ℕ **x** ℕ **x** ℕ → **Time**
- **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) $\triangleq$ h < 24 ∧ m < 60 ∧ s < 60

- ☐ **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
- ☐ **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)

**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

| *DiskScanner* |
|---|
| *damagedBlocks: Block [*]* |
| *addBlock(Integer, Integer)*<br>*removeBlock (Integer, Integer)*<br>*isDamaged(Integer, Integer):Boolean*<br>*getBadSectors(Integer): Integer [*]* |

➢ addBlock (**trackIn:** ℕ , **sectorIn:** ℕ)
**ext wr** damagedBlocks: Block**-set**
**pre mk**-Block (trackIn, sectorIn) ∉ damagedBlocks
**post** damagedBlocks =$\overline{\text{damagedBlocks}}$ ∪ {**mk**-Block (trackIn, sectorIn)}

➢ removeBlock(**trackIn:** ℕ , **sectorIn:** ℕ)
**ext wr** damagedBlocks: Block**-set**
**pre mk**-Block (trackIn, sectorIn) ∈ damagedBlocks
**post** damagedBlocks =$\overline{\text{damagedBlocks}}$ \ {**mk**-Block (trackIn, sectorIn)}

➢ 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}

**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
- ☐

| *<<enumeration>>*<br>*Status* |
|---|
| *READY*<br>*BLOCKED* |

| *Process* |
|---|
| *id: String*<br>*status: Status* |

| *ProcessManagement* |
|---|
| *running: String*<br>*waiting: Process[*]* |
| *admit(String)*<br>*dispatch()*<br>*timeOut()*<br>*block()*<br>*wakeUp(String)*<br>*terminate()* |

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

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

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

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

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

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

**THE LET… IN CLAUSE**

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

| **pre** *waiting(findPos(waiting, idIn)).status* = <BLOCKED> | **pre let** *pos* = *findPos(waiting, idIn)*<br>**in** *waiting(pos).status* = BLOCKED |
|---|---|
| **post** *waiting* = $\overline{waiting}$ † {*findPos*($\overline{waiting}$, *idIn*)<br>↦ **mk**–*Process*(*idIn*, <READY>)} | **post let** *pos* = *findPos*($\overline{waiting}$, *idIn*)<br>**in** *waiting* = $\overline{waiting}$ † {*pos* ↦ **mk**-*Process*(*idIn*, <READY>)} |
| **post** *waiting* = $\overline{waiting}$ † {*findPos*($\overline{waiting}$, *idIn*)<br>↦ **mk**–*Process*(*idIn*, <READY>)} | **post let** *pos* = *findPos*($\overline{waiting}$, *idIn*)<br>**in let** *wakeProcess* = **mk**-*Process*(*idIn*, <READY>)<br>*In waiting* = $\overline{waiting}$ † {*pos* ↦ *wakeProcess*} |

# WEEK 13

## 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.
- sensors = {A ↦ <LOW>, B ↦ <NORMAL>, C ↦ <NORMAL>, D ↦ <HIGH>, E ↦ <NORMAL>, F ↦ <NORMAL>}
- $m = \{a \mapsto y, b \mapsto x, C \mapsto x, d \mapsto z\}$
- 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:

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

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

  Empty map $\{\mapsto\}$

## 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}

### 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 ∪ m2" and "m2 ∪ m3" are undefined

### 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.**

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

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

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

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

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

### 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

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

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

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

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

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

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

### 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} ⩤ $m1 = \{b \mapsto 2, d \mapsto 3\}$

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

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

**Map Application**

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

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

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

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

$m1(d) = 3$

$m2( f ) = 1$

$m3( f ) = 2$

$m3(x)$ is undefined

**Using the Map Type in VDM-SL**

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

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

**Specifying 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.

| *SecuritySys* |
|---|
| *authorized: Employee[*]*<br>*inside: String[*]* |
| *addEmployee(String, String)*<br>*removeEmployee(String)*<br>*enter(String, String): Signal*<br>*leave(String): Signal* |

| *Employee* |
|---|
| *name: String*<br>*password: String* |
| |

| *<<enumeration>>*<br>*Signal* |
|---|
| *OPEN_DOOR*<br>*ACTIVATE_ALARM* |
| |

**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**
- **inv mk**-SecuritySys(a,i) $i \subseteq$ **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 = $\overline{authorized} \cup \{nameIn \mapsto passwordIn\}$

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 = $\{nameIn\} \lhd \overline{authorized}$

enter(nameIn : String, passwordIn : String) signal : Signal

- **ext rd** authorized : String $\xrightarrow{m}$ String
- **wr** inside : String-**set**

**pre** TRUE

**post** (authorized(nameIn) = passwordIn $\wedge$ nameIn $\notin$ $\overline{inside}$)

$\wedge$ (inside = $\overline{inside}$ $\cup$ {nameIn} $\wedge$ signal = <OPEN_DOOR>)

$\vee$ (authorized(nameIn) $\neq$ passwordIn $\vee$ nameIn $\in$ inside)

$\wedge$ (inside = $\overline{inside}$ $\wedge$ signal = <ACTIVATE_ALARM>)

- leave(nameIn : String) signal : Signal

**ext wr** inside : String-**set**

**pre** TRUE

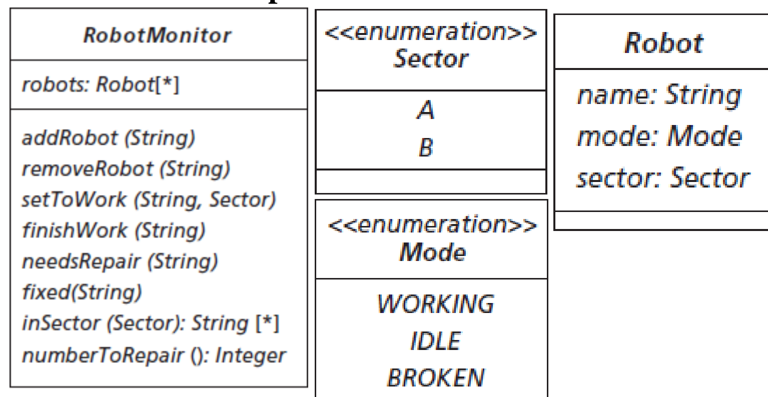
**post** nameIn $\in$ $\overline{inside}$ $\wedge$ inside = $\overline{inside}$ \{nameIn} $\wedge$ signal = <OPEN_DOOR>

$\vee$ nameIn $\notin$ inside $\wedge$ inside = $\overline{inside}$ $\wedge$ signal=<ACTIVATE_ALARM>

**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.
- **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.




addRobot (nameIn: String)

**ext wr** robots: String $\xrightarrow{m}$ Robot
**pre** nameIn ∉ dom robots
**post** robots = $\overline{\text{robots}}$ ∪ {nameIn ↦ mk-Robot (nameIn, <IDLE>, nil)}

removeRobot (nameIn: String)

**ext wr** robots: String $\xrightarrow{m}$ Robot
**pre** nameIn ∈ **dom** robots ∧ robots(nameIn).mode ≠ <WORKING>
**post** robots = {nameIn} ⩤ $\overline{\text{robots}}$

setToWork (nameIn: String,sectorIn:Sector)

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

finishWork (nameIn: String)

**ext wr** robots: String $\xrightarrow{m}$ Robot
**pre** nameIn ∈ **dom** robots ∧ robots(nameIn).mode = <WORKING>
**post** robots = $\overline{\text{robots}}$ † { nameIn ↦ **mk**-Robot(nameIn, <IDLE>, nil ) }

needsRepair (nameIn: String)

**ext wr** robots: String $\xrightarrow{m}$ Robot
**pre** nameIn ∈ **dom** robots
**post** robots = $\overline{\text{robots}}$ † { nameIn ↦ **mk**-Robot(nameIn, <BROKEN>, nil ) }

fixed (nameIn: String)

**ext wr** robots: String $\xrightarrow{m}$ Robot
**pre** nameIn ∈ **dom** robots ∧ robots(nameIn).mode = <BROKEN>
**post** robots = $\overline{\text{robots}}$ † { nameIn ↦ μ($\overline{\text{robots}}$(nameIn), mode ↦ <IDLE>)}

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>}

## WEEK 14

### 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:

$$\begin{array}{l} \underline{PhoneDB} \\ known : \mathbb{P}\ NAME \\ phone : NAME \nrightarrow PHONE \\ \hline known = \mathrm{dom}\ phone \end{array}$$

## Possible state of the system

$$\begin{array}{l} known = \{Smith, Jones, Robinson\} \\ phone = \{Smith \mapsto 01\text{–}325\text{–}4939, \\ \quad Jones \mapsto 0865\text{–}54141, \\ \quad Robinson \mapsto 0865\text{–}54141\} \end{array}$$

- 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.

## 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.
- **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.

## 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 algebraic methods, where the aim is to describe a system in terms of its desired properties, without constructing and explicit model.

## Methods

| Model oriented methods | Algebraic methods |
|---|---|
| • Z | • Clear |
| • VDM (Vienna Development Method) | • 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.
- Algebraic 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

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

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:

**AddPhone operation**

```
AddPhone(name: NAME, number: PHONE)
ext wr known: set of NAME
    wr phone: map NAME to PHONE
pre name ∉ known
post known = known↼ ∪ name ∧
        phone = phone↼ ∪ {name ↦ number}.
```

**FindPhone operation**

```
FindPhone(name: NAME) number: PHONE
ext rd known: set of NAME
    rd phone: map NAME to PHONE
pre name ∈ 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.

**Algebraic specifications**

- Algebraic specifications begin with a world which is far simpler than the rich universe of structured types assumed by Z and VDM.
- Their basic vocabulary 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:
- 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.
- Sorting can be described as a higher-order function which maps ordering relations to functions from sequences to sequences

**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

```
const Name =
    enrich Bool by
        sorts name
        opns _==_: name, name → bool
        eqns n == n = true
             n == m = m == n
             n == m ∧ m == p ⇒ n == p = true
    enden
```

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

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

**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:
- 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:
- 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:

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

- A set of x is a subset of another set y if every element of x is also an element of y:
- 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:
- 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 biggest, which must be infinite, for it contains ∅ and is closed under the operation taking x to x U {x}:
- Without this axiom, there is no guarantee 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 ∅ 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
- 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:

**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
- This encoding couple could be defined in terms of the primitive operations on the world of sets:
- 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 ($x_1$, ..., $x_n$).
- 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
- This freedom allows tuple to be defined constructively by iterating the couple operation.

**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
- This corresponds with the more informal notation as follows:

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

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

- 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:

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

**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:

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

- 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:

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

- 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:

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

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

$$\vdash f \in given \rightarrow Type(given') \land g \in given' \rightarrow Type(given'') \Rightarrow (tsubst\ g) \circ (tsubst\ f) = tsubst\ ((tsubst\ g) \circ f).$$

**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:

$$\begin{array}{|l} SIG \\ given : \mathbb{F}\ NAME \\ vars : \mathbb{F}\ NAME \\ type : NAME \nrightarrow TYPE \\ \hline type \in vars \rightarrow Type(given) \end{array}$$

- 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:

$$\begin{array}{|l} A[X, Y] \\ p : X \\ q : X \times Y \\ \hline \exists\, y : Y \bullet q = (p, y) \end{array}$$

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

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

**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

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

Satisfies the axiom, but the structure

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

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.

$$\begin{array}{|l} STRUCT \\ gset : NAME \nrightarrow W \\ val : NAME \nrightarrow W \end{array}$$

- 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:

**Variety**

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

$$\begin{array}{|l} VARIETY \\ sig : SIG \\ models : \mathbb{P}\ STRUCT \\ \hline models \subseteq Struct(sig) \end{array}$$

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

**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.

$$EXP ::= var \langle\langle IDENT \rangle\rangle \mid plus \langle\langle EXP \times EXP \rangle\rangle \mid times \langle\langle EXP \times EXP \rangle\rangle$$

- So instead of giving the syntax of expressions as, say,

We might write the following:

$$EXP ::= IDENT \mid EXP + EXP \mid EXP * EXP.$$

  - 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  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 ≅ an 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.

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

- This means that the specification

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 ∩ 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
- 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.

**The language of schemas**

$$SEXP ::= \cdots \mid SEXP \wedge SEXP \mid SEXP \vee SEXP \mid SEXP \upharpoonright SEXP \mid \cdots$$

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

$$sexp : ENV \to SEXP \nrightarrow VARIETY$$
$$\cdots$$
$$sexp\ \rho\ [\![se_1 \wedge se_2]\!] \cong combine(sexp\ \rho\ [\![se_1]\!], sexp\ \rho\ [\![se_2]\!])$$
$$\cdots$$

- 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.

$$combine : VARIETY \times VARIETY \nrightarrow VARIETY$$
$$combine(\theta VARIETY_1, \theta VARIETY_2) \cong \mu\ VARIETY' \mid sig' \cong join(sig_1, sig_2) \wedge models' = \{ M : Struct(sig') \mid restrict\ sig_1\ M \in models_1 \wedge restrict\ sig_2\ M \in models_2 \}.$$

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

$$\cdots$$
$$sexp\ \rho\ [\![se_1 \vee se_2]\!] \cong disjoin(sexp\ \rho\ [\![se_1]\!], sexp\ \rho\ [\![se_2]\!])$$
$$\cdots$$

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

$$\cdots$$
$$sexp\ \rho\ [\![se_1 \upharpoonright se_2]\!] \cong project(sexp\ \rho\ [\![se_1]\!], sexp\ \rho\ [\![se_2]\!])$$
$$\cdots$$