Software Modelling and Validation Using VDM

Hsin-Hung Lin

September 7, 2017

FLOLAC'17

Based on slides from overturetools.org
Credits: John Fitzgerald, Peter Gorm Larsen, Takahiko Ogino

Introduction

Software Today: why we need to model systems

- Challenges in software development
- Modelling Computing Systems
- Formality
- Formal specification languages
- The structure & content of this lecture

Characteristics of Software

- We build computing systems out of software
  - Engineers in other disciplines use physical materials like steel, electronic devices or advanced materials.

- What makes software different?

Software Today: challenges

- Technological:
  - you can do more in software than before
- Software is often used for critical tasks.
  - Name some safety- or security-critical applications
- For example?

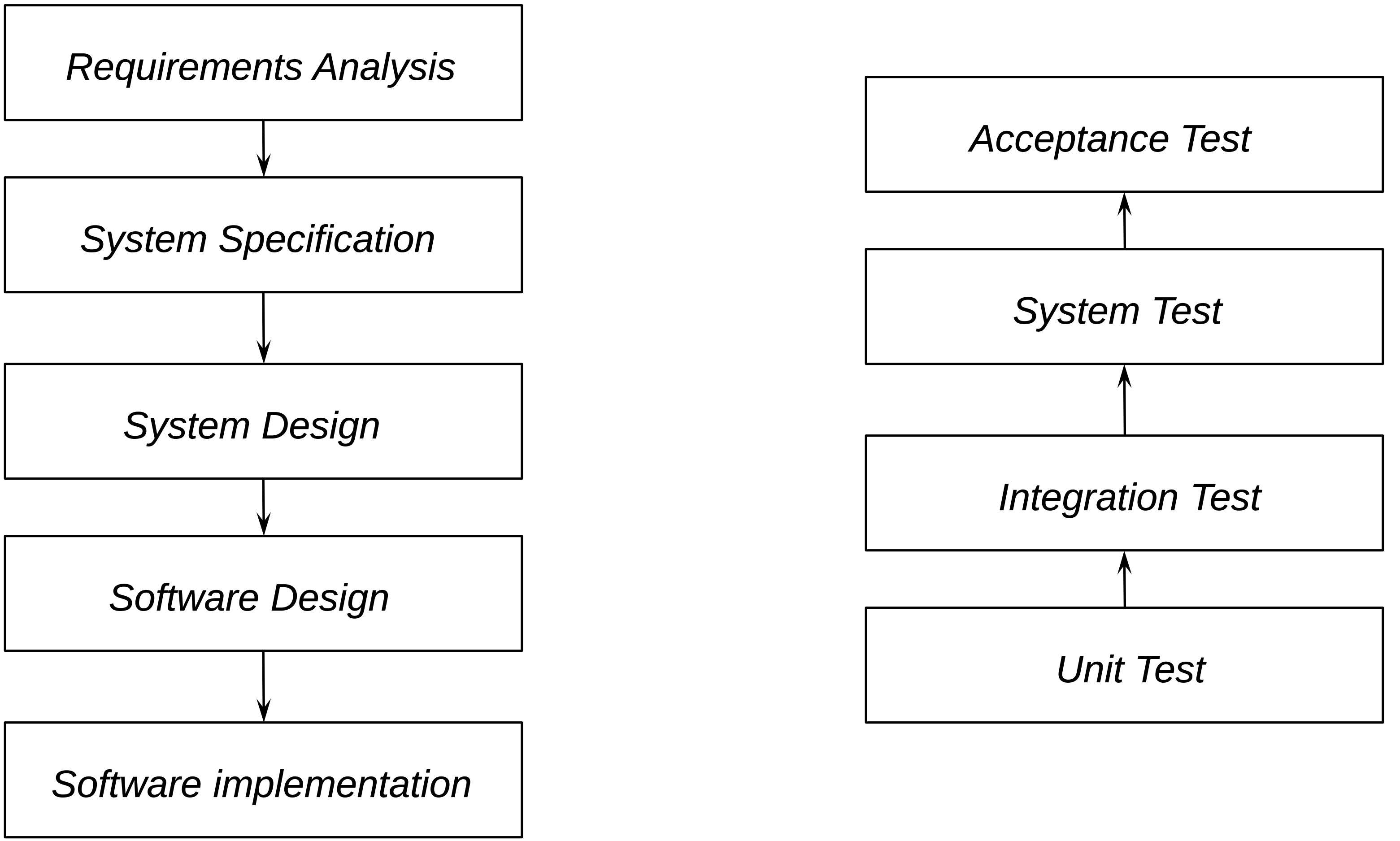
Software Today: challenges

- Economic Challenges: the cost of rework

- Software development takes place on a huge scale, and often goes wrong!

- How much software gets used as delivered?

Software Today: challenges
Rework Costs



Software Today: challenges Rework Costs

- The rework cost to fix a bug is related to “distance” between the commission and the discovery of the error.
- Improved analysis of requirements and designs could reduce the rework costs for some of the most expensive errors.
- This lecture is about a particular class of techniques which help us to do this kind of analysis.

Modelling Computing Systems

- In other engineering disciplines (Mechanical, Electrical, Aeronautical etc.) system models are built to help gain confidence in requirements and designs.
  - For example?

- In this lecture, we will look at how we can build and analyze models of software. There are two characteristics of these models which are crucial to their successful use: abstraction and rigour.

Modelling: Abstraction

- Engineering models omit details which are not relevant to the purpose of the model.

- The omission of detail not relevant to a model’s purpose is called abstraction. The choice of which details to omit is a matter of engineering skill.

Modelling: Abstraction

- Compare these extracts from two descriptions of the same system.

The FlightFinder System is to be used by travel agents and their customers. Details are entered, including point of departure, destination, preferred dates and times. The system will respond with a range of itineraries and fares, along with the relevant restrictions.

The system record locations as nodes in a connected graph structure. Each node struct contains an array of pointers to reachable destinations plus, for each pointer, a timetable of flights stored as a hash table. Each record in the hash table has a flight number (8 character string), departure and arrival times (standard time formats) and operating dates (standard date format). To obtain the optimal route, the graph must be traversed using a shortest path algorithm on a modified adjacency matrix ...

Modelling: Rigor

- The most important property of a model of a computing system is its suitability for analysis. The analysis must be **objective** (not down to the opinion of the individual engineers performing it). It should also be **repeatable** and **susceptible to machine support**.
- The **language** in which a model is expressed should be **rigorously defined**: little room for disagreement about what a model actually says; analysis tools reach the same conclusion about the properties of models.

Modelling computing systems

- How do these concepts of system modelling transfer to software development?
- A range of modelling techniques are used in software development:
  - For example?
- Models constructed in early development stages are **specifications**; those developed in later stages are designs. We will generally be concerned with specifications (because of the importance of modelling in early development stages) but we will tend to use the term **model** to refer to the system descriptions that we develop.

Formality

- This lecture concentrates on formal languages for expressing models.
- A language is formal if its syntax rules and its semantics (the meaning of every construct in the language) are so precisely defined that there is no room for disagreement about the meaning of a model. Models expressed in a formal language are susceptible to a wide range of analysis techniques including mathematical proof (we can, in principle, prove that a model embodies a property such as safety or indeed prove that a program is correct with respect to a specification).

Formal Specification Languages

- A formal specification language is a formal language used for expressing models of computing systems. Such languages typically provide support for abstraction and rigour.

General Purpose	Special Purpose
VDM-SL	CCS
Z	CSP
RSL (RAISE)	Real-Time Logic
	Deontic Logics

Formal Specification Languages: VDM-SL

- Vienna Development Method (VDM)
- Spec. Language is VDM-SL
- ISO Standardized: fully formal
- Support Tools are available
- Good record of industrial use
- Support for abstraction of data and functionality

Structure of Lecture

- Introduction
- Guided tour through a formal model
- Basic abstractions: data types
- Principal abstractions: sets, sequences, mappings
- State, function, and operation
- Validation
- Case studies

Principle of Lecture

- **Formal Methods are part of practical Systems Engineering, not theoretical Computing Science!**

- All examples are based on real formal models developed in a commercial context.

- Practice is the key to master the modelling skills used in this lecture.

References and Reading

- Fitzgerald & Larsen, “Modelling Systems: Practical Tools and Techniques in Software Development”, Cambridge Univ. Press 1998, ISBN 0-521-62348-0
- Documents on http://overturetool.org/
    - Language manual
    - Guides of Overture tool

Constructing a Model

A guided tour through a model in VDM-SL

Deriving a Model

- Chemical Plant Alarm System

- Requirements
- Data Types and invariants
- Functions and pre-conditions

The example is derived from a subcomponent of a large alarm and callout system developed by IFAD, a Danish high-technology firm for the local telephone company Tele Danmark Process.

The contents of a model in VDM-SL

Type Definitions, e.g.

```
Altitude = real
inv alt == alt >= 0

Position :: lat  : Latitude
            long : Longitude
            alt  : Altitude
```

Function definitions, e.g.

```
Move: Id * Position * ATCSystem  ->  ATCSystem
Move(id, pos, sys) == … expression …
pre … expression …
```

The contents of a model in VDM-SL

Data Types built from basic types (`int`, `real`, `char`, `bool` etc.) using type constructors (sets, sequences, mappings, records).

Newly constructed types can be named and used throughout the model.

A **data type invariant** is a Boolean expression that is used to restrict a data type to contain only those values that satisfy the expression.

Functions define the functionality of the system. Functions are referentially transparent - no side-effects and no global variables. In cases where it is intuitive to have global variables, a different **operational style** of modelling is used.

A **pre-condition** is a Boolean expression over the input variables that is used to record restrictions assumed to hold on the inputs.

The contents of a model in VDM-SL

Data abstraction is provided by the unconstrained nature of the data types in VDM-SL. Sets, sequences and mappings, although finite, are unbounded.

Function abstraction, when required, is provided by **implicit specification**.

```
SquareRoot(x:nat)r:real
pre  x >= 0
post r*r = x
```

Post-conditions are Boolean expressions relating inputs and outputs. Post-conditions are used when we do not wish to explicitly define which output is to be returned, or where the explicit definition would be too concrete.

Deriving a Formal Model from Scratch

- No right or wrong way to construct a formal model from a requirements description.
- Always begin by considering a model's **purpose**, as this guides abstraction decisions during development.
- Following steps:
  1. *Read the requirements.*
  2. *Extract a list of possible data types (often from nouns) and functions (often from verbs/actions).*
  3. *Sketch out representations for the data types.*
  4. *Sketch out signatures for the functions.*
  5. *Complete type definitions by determining invariants.*
  6. *Complete the function definitions, modifying data type definitions if required.*
  7. *Review the requirements, noting how each clause has been treated in the model.*

Requirements for the Alarm Example

A chemical plant has monitors which can raise alarms in response to conditions in the plant. When an alarm is raised, an expert must be called to the scene. Experts have different qualifications for coping with different kinds of alarm.

R1*: A computer-based system is to be developed to manage expert call-out in response to alarms.*

R2*: Four qualifications: electrical, mechanical, biological and chemical.*

R3*: There must be experts on duty at all times.*

R4*: Each expert can have a whole list of qualifications, not just one.*

R5*: Each alarm has a description (text for the expert) and a qualification.*

R6*: When an alarm is raised, the system should output the name of a qualified and available expert who can then be called in.*

R7*: It shall be possible to check when a given expert is available.*

R8*: It shall be possible to assess the number of experts on duty at a given period*

Purpose of the model ...

- To clarify the rules governing the duty rota and the calling out of experts in response to alarms.

Aside: *We often find in professional practice that the purpose for which a model is to be developed is only rarely made clear. Yet it is this purpose which should govern the choice of abstractions made in the development of the model and hence the success, ease of use etc. of the model itself.*

Possible data types and functions

- Types
  - Plant
  - Qualification
  - Alarm
  - Period
  - Expert
  - Description
- Functions
  - ExpertToPage
  - ExpertIsOnDuty
  - NumberOfExperts

Sketching type representations:
Enumerated types

R2: Four qualifications: electrical, mechanical, biological and chemical.

```
Qualification = <Elec> | <Mech> | <Bio> | <Chem>
```

- The | constructs the union of several types or quote literals
- The individual quoted values are put in angle brackets <...>
- This type has four elements corresponding to the four kinds of alarm and qualification.
- Just like an enumerated type in a programming language.

Sketching type representations: **Record types**

R5: Each alarm has a description (text for the expert) and ***a qualification****.*

It is always worth asking clients whether they mean "a" or "some" or "at least one".

```
Alarm :: alarmtext : seq of char
         quali     : Qualification
```

```
Alarm :: alarmtext : seq of char
         quali     : Qualification
```

To say that a value v has type T, we write

```
v : T
```

So, to state that a is an alarm, we write

```
a : Alarm
```

Record types

To extract the fields from a record, we use a dot notation:

```
a.alarmtext
```

To say that a is made up from some values, we use a record constructor "mk_":

```
a = mk_Alarm("Disaster - get here fast!", <Elec>)
```

This constructor builds a record from the values for its fields

Sketching type representations:
Mapping types

R4: Each expert can have a whole list of qualifications, not just one.

*Ask the client "Did you really mean a **list**, i.e. the order in which they are presented is important?*

```
Expert :: expertId : ExpertId
          quali    : set of Qualification
```

Sometimes requirements given in natural language do not mean exactly what they say. If in doubt, consult an authority or the client! Hence a set here rather than a sequence.

We try to keep the formal model as abstract as possible - we only record the information that we need for the **purpose** of the model. The choice of what is relevant and what is not relevant is a matter of serious engineering judgement, especially where safety is concerned.

Sketching type representations: **Token types**

The informal requirements give us little indication that we will need to look inside the experts' identifiers. When we need a type, but no detailed representation, we use the special symbol `token`.

```
ExpertId = token
```

The same is also true for the periods into which the plant's timetable is split:

```
Period = token
```

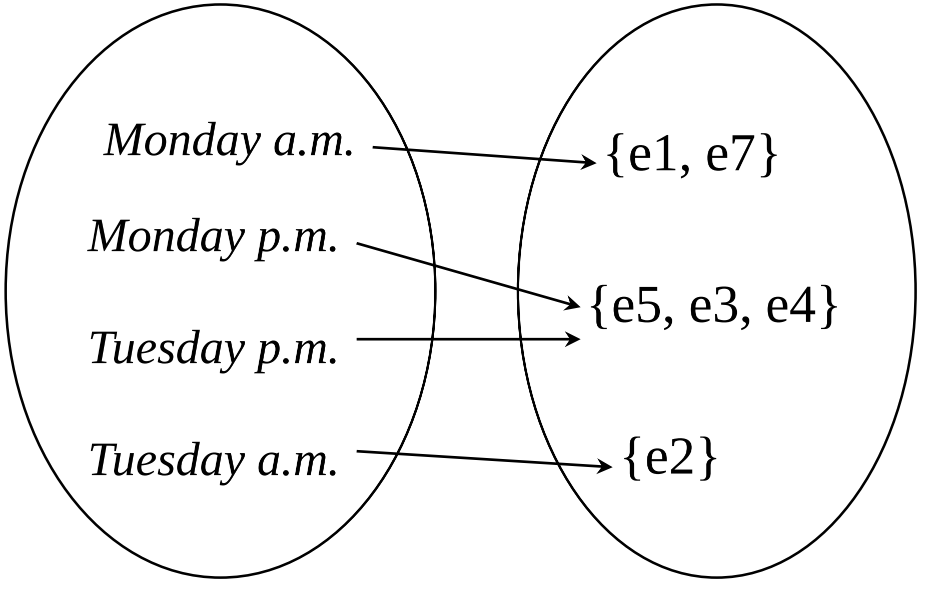
Sketching type representations: **Mapping types**

R3: There must be experts on duty at all times.

R7: It shall be possible to check when a given expert is available.

These requirements imply that there must be some sort of schedule relating each period of time to the set of experts who are on duty during that period:

```
Schedule = map Period to (set of Expert)
```



Sketching type representations:
The whole plant

R1: A computer-based system is to be developed to manage expert call-out in response to alarms.

```
Plant :: sch    : Schedule
         alarms : set of Alarm
```

The model so far - type definitions

```
Plant :: sch    : Schedule
         alarms : set of Alarm

Schedule = map Period to set of Expert

Period = token

Expert :: expertid : ExpertId
          quali    : set of Qualification

ExpertId = token

Qualification = <Elec> | <Mech> | <Bio> | <Chem>

Alarm :: alarmtext : seq of char
         quali     : Qualification
```

Sketching function signatures

Possible functions were: `ExpertToPage`
`ExpertIsOnDuty`
`NumberOfExperts`

A function definition shows the types of the input parameters and the result in a signature:

```
ExpertToPage: Alarm * Period * Plant -> Expert

ExpertIsOnDuty: Expert * plant -> set of Period

NumberOfExperts: Period * Plant -> nat
```

Complete type definition:
Data type invariants

Additional constraints on the values in the system which must hold at all times are called *data type invariants*.

Example: suppose we agree with the client that experts should always have at least one qualification. This is a restriction on the type `Expert`. To state the restriction, consider a typical value `ex` of type `Expert`

```
ex.quali <> {}
```

We attach invariants to the definition of the relevant data type:

```
Expert :: expertid : ExpertId
          quali    : set of Qualification
inv ex == ex.quali <> {}
```

This is a formal parameter standing for a typical element of the type.

The body of the invariant is a Boolean expression recording the restriction on the formal parameter which represents a typical element of the type.

Complete type definition:
Data type invariants

R3: There must be experts on duty at all times.

This is a restriction on the schedule to make sure that, for all periods, the set of experts is not empty.

Again, we state this formally. Consider a typical schedule, called `sch`

```
forall exs in set rng sch & exs <> {}
```

Attaching this to the relevant type definition:

```
Schedule = map Period to set of Expert
inv sch == forall exs in set rng sch & exs <> {}
```

Complete function definitions

A function definition contains:

A signature

```
NumberOfExperts: Period * Plant -> nat
```

A parameter list

```
NumberOfExperts(per,pl) ==
```

A body

```
card pl.sch(per)
```

A pre-condition (optional)

```
pre per in set dom pl.sch
```

If omitted, the pre-condition is assumed to be `true` so the function can be applied to any inputs of the correct type.

Complete function definitions

R7: It shall be possible to check when a given expert is available.

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex,pl) ==
    {per | per in set dom pl.sch &
           ex in set pl.sch(per)}
```

For convenience, we can use the record constructor in the input parameter to make the fields of the record `pl` available in the body of the function without having to use the selectors:

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex,mk_Plant(sch,alarms)) ==
   {per | per in set dom sch & ex in set sch(per)}
```

Complete Function Definitions

The `alarms` component of the `mk_Plant(sch,alarms)` parameter is not actually used in the body of the function and so may be replaced by a `-`. The final version of the function is:

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex, mk_Plant(sch,-)) ==
     {per | per in set dom sch & ex in set sch(per)}
```

Complete Function Definitions

R6: When an alarm is raised, the system should output the name of a qualified and available expert who can then be called in.

```
ExpertToPage: Alarm * Period * Plant -> Expert
ExpertToPage(al,per,pl) == ???
```

Can we specify what result has to be returned without worrying about how we find it? Use an *implicit definition:*

```
ExpertToPage(al:Alarm, per:Period, pl:Plant) r:Expert
pre   …
post  r in set pl.sch(per) and
      al.quali in set r.quali
```

Have you spotted a problem with the system?

The requirements were silent about ensuring that there is always an expert with the correct qualifications available. After consulting with the client, it appears to be necessary to ensure that there is always at least one expert with each kind of qualification available. How could we record this in the model?

```
Plant :: sch    : Schedule
         alarms : set of Alarm
```

```
inv mk_Plant(sch,alarms) ==
     forall a in set alarms &
          forall per in set dom sch &
               exists ex in set sch(per) &
                    a.quali in set ex.quali
```

Finally, review the requirements

R1: system to manage expert call-out in response to alarms.

R2: Four qualifications.

R3: experts on duty at all times.

R4: expert can have list of qualifications.

R5: Each alarm has description & qualification.

R6: output the name of a qualified and available expert

R7: check when a given expert is available.

R8: assess the number of experts on duty at a given period

Finally, review the requirements

Recall the original requirements.

***R1**: A computer-based system is to be developed to manage expert call-out in response to alarms.*

***R2**: Four qualifications: electrical, mechanical, biological and chemical.*

***R3**: There must be experts on duty at all times.*

***R4**: Each expert can have a whole list of qualifications, not just one.*

***R5**: Each alarm has a description (text for the expert) and a qualification.*

***R6**: When an alarm is raised, the system should output the name of a qualified and available expert who can then be called in.*

***R7**: It shall be possible to check when a given expert is available.*

***R8**: It shall be possible to assess the number of experts on duty at a given period*

Weaknesses in the requirements

- Silence on ensuring that at least one suitable expert is available.
- Use of identifiers for experts was implicit.
- “List” really meant “set”.
- Silence on the fact that experts without qualifications are useless.
- “A qualification” meant “several qualifications”.

Summary

Process of developing a model depends crucially on the statement of the model's purpose.

VDM-SL models are based round type definitions and functions. Abstraction provided by the basic data types and type constructors and the ability to give implicit function definitions.

Basic types:

Type constructors:

Invariants:

Functions:

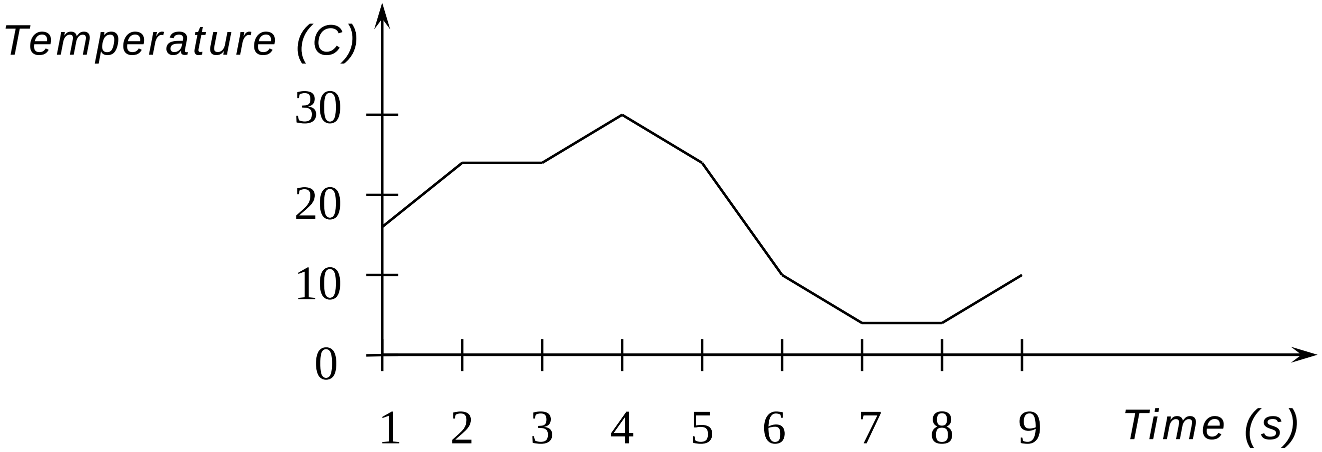
Logic Expressions

Logic Expressions in VDM

Our ability to state invariants, record preconditions and post-conditions, and the ability to reason about a formal model depend on the logic on which the modelling language is based.

- Classical logical propositions and predicates
- Connectives
- Quantifiers
- Handling undefinedness: the logic of partial functions

The temperature monitor example



The monitor records the last five temperature readings

25	*10*	*5*	*5*	*10*

The temperature monitor example

The following conditions are to be detected by the monitor:

Rising: the last reading in the sample is greater than the first

Over limit: there is a reading in the sample in excess of 400 C

Continually over limit: all the readings in the sample exceed 400 C

Safe: If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.

Alarm: The alarm is to be raised if and only if the reactor is not safe

Formal Model of the monitor:

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

Predicates: Propositions

Predicates are simply logical expressions. The simplest kind of logical predicate is a *proposition.*

A proposition is a logical assertion about a particular value or values, usually involving a Boolean operator to compare the values, e.g.

$3 < 27$ $\qquad\qquad\qquad\qquad$ $5 = 9$

Propositions are normally either true or false (but in VDM we also have to handle undefined values - see the later notes on the Logic of Partial Functions).

Propositions have very limited value:

Predicates: General predicates

A predicate is a logical expression that is not specific to particular values but contains variables which can stand for one of a range of possible values, e.g.

```
x < 27

(x**2) + x - 6 = 0
```

The truth or falsehood of a predicate depends on the value taken by the variables.

Predicates in the monitor example

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

Consider a monitor `m`. `m` is a sequence so we can index into it:

First reading in `m`:

Last reading in `m`:

Predicate stating that the first reading in m is strictly less than the last reading:

The truth of the predicate depends on the value of `m`.

Predicates: The rising condition

The last reading in the sample is greater than the first

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

We can express the rising condition as a Boolean function:

```
Rising: Monitor -> bool

Rising(m) == m.temps(1) < m.temps(5)
```

For any monitor `m`, the expression `Rising(m)` evaluates to true iff the last reading in the sample in `m` is higher than the first, e.g.

`Rising(` `mk_Monitor([233,45,677,650,900], false)` **`)`**

Basic logical operators

We build more complex logical expressions out of simple ones using logical connectives:

`not`	negation
`and`	conjunction
`or`	disjunction
`=>`	implication (if ... then ...)
`<=>`	biimplication (if and only if)

Basic logical operators: Negation

Negation allows us to state that the opposite of some logical expression is true, e.g.

The temperature in the monitor `mon` *is not rising:*

```
not Rising(mon)
```

Truth table for negation:

A	not A
true	false
false	true

Basic logical operators: Disjunction

Disjunction allows us to express alternatives that are not necessarily exclusive:

Over limit: *There is a reading in the sample in excess of 400 C*

```
OverLimit: Monitor -> bool
OverLimit(m) ==
```

Truth table for disjunction:

A	B	A **or** B
true	true	true
true	false	true
false	true	true
false	false	false

Basic logical operators: Conjunction

Conjunction allows us to express the fact that all of a collection of facts are true.

Continually over limit: *all the readings in the sample exceed 400 C*

```
COverLimit: Monitor -> bool
COverLimit(m) == m.temps(1) > 400 and
                 m.temps(2) > 400 and
                 m.temps(3) > 400 and
                 m.temps(4) > 400 and
                 m.temps(5) > 400
```

Truth table for conjunction:

A	B	A **and** B
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Basic logical operators: Implication

Implication allows us to express facts which are only true under certain conditions ("if … then …"):

Safe: If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.

```
Safe: Monitor -> bool
Safe(m) == temp(3) > 400 => temp(5) < 400
```

A	B	A **=>** B
true	true	true
true	false	false
false	true	true
false	false	true

Basic logical operators: Biimplication

Biimplication allows us to express equivalence (“if and only if”).

Alarm: The alarm is to be raised if and only if the reactor is not safe

This can be recorded as an invariant property:

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5 and not Safe(temps) <=> alarm
```

A	B	A **<=>** B
true	true	true
true	false	false
false	true	false
false	false	true

Quantifiers

For large collections of values, using a variable makes more sense than dealing with each case separately.

inds `m.temps` represents indices (1-5) of the sample

The "over limit" condition can then be expressed more economically as:

```
exists i in set inds m.temps & temps(i) > 400
```

The "continually over limit" condition can then be expressed using "`forall`":

Quantifiers

Syntax:

`forall` *binding* & *predicate*

`exists` *binding* & *predicate*

There are two types of binding:

Type Binding, e.g.

```
x:nat
n: seq of char
```

*A type binding lets the bound variable range over a **type** (a possibly infinite collection of values).*

Set Binding, e.g.

```
i in set inds m
x in set {1,…,20}
```

*A set binding lets the bound variable range over a **finite set of values.***

Quantifiers

Several variables may be bound at once by a single quantifier, e.g.

```
forall x,y in set {1,…,5} &
        not m.temp(x) = m.temp(y)
```

Would this predicate be true for the following value of `m.temp` ?

```
[320, 220, 105, 119, 150]
```

Quantifiers: Exercises

All the readings in the sample are less than 400 and greater than 50.

Each reading in the sample is up to 10 greater than its predecessor.

There are two distinct readings in the sample which are over 400.

Quantifiers: Exercises

All the readings in the sample are less than 400 and greater than 50.

```
forall i in set inds temp & temp(i) < 400 and temp(i) > 50
```

Each reading in the sample is up to 10 greater than its predecessor.

```
forall i in set inds temp\{1} &
      temp(i-1) > temp(i) and temp(i-1) + 10 >= temp(i)
```

There are two distinct readings in the sample which are over 400.

```
exists i,j in set inds temp &
      i <> j and temp(i) > 400 and temp(j) > 400
```

Quantifiers: Exercises

Suppose we have to formalise the following property:

There is a "single minimum" in the sequence of readings, i.e. there is a reading which is strictly smaller than any of the other readings.

Hint: use two quantifiers

Suppose the order of the quantifiers is reversed.

Quantifiers: Exercises

Suppose we have to formalise the following property:

There is a "single minimum" in the sequence of readings, i.e. there is a reading which is strictly smaller than any of the other readings.

Hint: use two quantifiers

```
exists min in set {1,...,5} &
       forall i in set {1,...,5} &
              i <> min => temp(i) > temp(min)
```

Suppose the order of the quantifiers is reversed.

```
forall i in set {1,...,5} &
       exists min in set {1,...,5} &
              i <> min => temp(i) > temp(min)
```

Summary

- **Propositions**
- **Predicates involve free variables**
- **Predicates may be combined using connectives**
- **Free variables can range over collections of values, using quantifiers**
- **Quantifiers can be mixed**

LPF: coping with undefinedness

Suppose sensors can fail in such a way that they generate the value **Error** instead of a valid temperature.

In this case we can not make comparisons like

`Error < 400`

The logic in VDM is equipped with facilities for handling undefined applications of operators (e.g. if division by zero could occur).

Truth tables are extended to deal with the possibility that undefined values can occur, e.g.

A	not A
true	false
false	true
*	*

** represents the undefined value*

Disjunction in LPF

A	B	A **or** B
true	true	true
true	false	true
true	*	true
false	true	true
false	false	false
false	*	*
*	true	true
*	false	*
*	*	*

If one disjunct is true, we know that the whole disjunction is true, regardless of whether the other disjunct is true, false or undefined.

Conjunction in LPF

A	B	A **and** B
true	true	true
true	false	false
true	*	*
false	true	false
false	false	false
false	*	false
*	true	*
*	false	false
*	*	*

If one conjunct is false, we know that the whole conjunction is false, regardless of whether the other disjunct is true, false or undefined.

Implication and Biimplication in LPF

A	B	A **=>** B
true	true	true
true	false	false
true	*	*
false	true	true
false	false	true
false	*	true
*	true	true
*	false	*
*	*	*

A	B	A **<=>** B
true	true	true
true	false	false
true	*	*
false	true	false
false	false	true
false	*	*
*	true	*
*	false	*
*	*	*

Datatypes in VDM

Type Definitions

- **Basic data types**

  Boolean

  Numeric

  Tokens

  Characters

  Quotations

- **Compound data types**

  Set types

  Sequence types

  Map types

  Product types

  Record types

  Union types

  Operation types

  Function types

Invariants can be added to types

bool

Values: true, false

Operator	Name	Type
`not b`	Negation	$\text{bool} \to \text{bool}$
`a and b`	Conjunction	$\text{bool} * \text{bool} \to \text{bool}$
`a or b`	Disjunction	$\text{bool} * \text{bool} \to \text{bool}$
`a => b`	Implication	$\text{bool} * \text{bool} \to \text{bool}$
`a <=> b`	Biimplication	$\text{bool} * \text{bool} \to \text{bool}$
`a = b`	Equality	$\text{bool} * \text{bool} \to \text{bool}$
`a <> b`	Inequality	$\text{bool} * \text{bool} \to \text{bool}$

Operator	Name	Type
-x	Unary minus	real → real
abs x	Absolute value	real → real
floor x	Floor	real → int
x + y	Sum	real * real → real
x - y	Difference	real * real → real
x * y	Product	real * real → real
x / y	Division	real * real → real
x div y	Integer division	int * int → int
x rem y	Remainder	int * int → int
x mod y	Modulus	int * int → int
x**y	Power	real * real → real
x < y	Less than	real * real → bool
x > y	Greater than	real * real → bool
x <= y	Less or equal	real * real → bool
x >= y	Greater or equal	real * real → bool
x = y	Equal	real * real → bool
x <> y	Not equal	real * real → bool

```
int
nat
nat1
real
```

char

Values: ’a’, ’b’, ’1’, ’2’, ’+’, ’-’, ...

Operator	Name	Type
c1 = c2	Equal	char $*$ char $\to$ bool
c1 <> c2	Not equal	char $*$ char $\to$ bool

For a sequence type defined as

```
string = seq of char
```

The following expression is true

```
[ ‘a’, ’b’, ’c’, ’d’, ’e’ ] = “abcde”
```

quote

Values: <RED>, <CAR>, <QuoteLit>, ...

Operator	Name	Type
`q = r`	Equal	$T * T \to \mathsf{bool}$
`q <> r`	Not equal	$T * T \to \mathsf{bool}$

Quote types are usually used with union to represent enumerations
For example:

```
A = <NEG> | <ZERO> | <POS>
```

to represent the abstraction of integers

token

Values: mk_token(5), mk_token({9, 3}), ...

Operator	Name	Type
`s = t`	Equal	token $*$ token $\rightarrow$ bool
`s <> t`	Not equal	token $*$ token $\rightarrow$ bool

Tokens are used for representing types that is not needed to be in detail. Usually used for values that are not accessed or changed by functionalities of a system. There are no ordering between tokens.

set

S = set of A

Operator	Name	Type
e in set s1	Membership	$A *$ set of $A \rightarrow$ bool
e not in set s1	Not membership	$A *$ set of $A \rightarrow$ bool
s1 union s2	Union	set of $A *$ set of $A \rightarrow$ set of A
s1 inter s2	Intersection	set of $A *$ set of $A \rightarrow$ set of A
s1 \ s2	Difference	set of $A *$ set of $A \rightarrow$ set of A
s1 subset s2	Subset	set of $A *$ set of $A \rightarrow$ bool
s1 psubset s2	Proper subset	set of $A *$ set of $A \rightarrow$ bool
s1 = s2	Equality	set of $A *$ set of $A \rightarrow$ bool
s1 <> s2	Inequality	set of $A *$ set of $A \rightarrow$ bool
card s1	Cardinality	set of $A \rightarrow$ nat
dunion ss	Distributed union	set of set of $A \rightarrow$ set of A
dinter ss	Distributed intersection	set of set of $A \rightarrow$ set of A
power s1	Finite power set	set of $A \rightarrow$ set of set of A

Examples: Let s1 = {<France>,<Denmark>,<SouthAfrica>,<SaudiArabia>}, s2 = {2, 4, 6, 8, 11} and s3 = {} then:

<England> in set s1	≡	false
10 not in set s2	≡	true
s2 union s3	≡	{2, 4, 6, 8, 11}
s1 inter s3	≡	{}
(s2 \ {2,4,8,10}) union {2,4,8,10} = s2	≡	false
s1 subset s3	≡	false
s3 subset s1	≡	true
s2 psubset s2	≡	false
s2 <> s2 union {2, 4}	≡	false
card s2 union {2, 4}	≡	5
dunion {s2, {2,4}, {4,5,6}, {0,12}}	≡	{0,2,4,5,6,8,11,12}
dinter {s2, {2,4}, {4,5,6}}	≡	{4}
dunion power {2,4}	≡	{2,4}
dinter power {2,4}	≡	{}

seq

`S = seq of A`

Operator	Name	Type
`hd l`	Head	seq1 of $A \to A$
`tl l`	Tail	seq1 of $A \to$ seq of A
`len l`	Length	seq of $A \to$ nat
`elems l`	Elements	seq of $A \to$ set of A
`inds l`	Indexes	seq of $A \to$ set of nat1
`l1 ^ l2`	Concatenation	(seq of A) $*$ (seq of A) $\to$ seq of A
`conc ll`	Distributed concatenation	seq of seq of $A \to$ seq of A
`l ++ m`	Sequence modification	seq of A $*$ map nat1 to $A \to$ seq of A
`l(i)`	Sequence application	seq of A $*$ nat1 $\to A$
`l1 = l2`	Equality	(seq of A) $*$ (seq of A) $\to$ bool
`l1 <> l2`	Inequality	(seq of A) $*$ (seq of A) $\to$ bool

Examples: Let `l1 = [3,1,4,1,5,9,2]`, `l2 = [2,7,1,8]`,
`l3 = [<England>, <Rumania>, <Colombia>, <Tunisia>]` then:

```
len l1                                    ≡  7
hd (l1^l2)                                ≡  3
tl (l1^l2)                                ≡  [1,4,1,5,9,2,2,7,1,8]
l3(len l3)                                ≡  <Tunisia>
"England"(2)                              ≡  'n'
conc [l1,l2] = l1^l2                      ≡  true
conc [l1,l1,l2] = l1^l2                   ≡  false
elems l3                                  ≡  { <England>, <Rumania>,
                                               <Colombia>,<Tunisia>}
(elems l1) inter (elems l2)               ≡  {1,2}
inds l1                                   ≡  {1,2,3,4,5,6,7}
(inds l1) inter (inds l2)                 ≡  {1,2,3,4}
l3 ++ {2 |-> <Germany>,4 |-> <Nigeria>}   ≡  [ <England>, <Germany>,
                                               <Colombia>, <Nigeria>]
```

map

S = map A to B
S = inmap A to B

Operator	Name	Type
`dom m`	Domain	(map A to B) $\rightarrow$ set of A
`rng m`	Range	(map A to B) $\rightarrow$ set of B
`m1 munion m2`	Merge	(map A to B) $*$ (map A to B) $\rightarrow$ map A to B
`m1 ++ m2`	Override	(map A to B) $*$ (map A to B) $\rightarrow$ map A to B
`merge ms`	Distributed merge	set of (map A to B) $\rightarrow$ map A to B
`s <: m`	Domain restrict to	(set of A) $*$ (map A to B) $\rightarrow$ map A to B
`s <-: m`	Domain restrict by	(set of A) $*$ (map A to B) $\rightarrow$ map A to B
`m :> s`	Range restrict to	(map A to B) $*$ (set of B) $\rightarrow$ map A to B
`m :-> s`	Range restrict by	(map A to B) $*$ (set of B) $\rightarrow$ map A to B
`m(d)`	Map apply	(map A to B) $*$ $A \rightarrow B$
`m1 comp m2`	Map composition	(map B to C) $*$ (map A to B) $\rightarrow$ map A to C
`m ** n`	Map iteration	(map A to A) $*$ nat $\rightarrow$ map A to A
`m1 = m2`	Equality	(map A to B) $*$ (map A to B) $\rightarrow$ bool
`m1 <> m2`	Inequality	(map A to B) $*$ (map A to B) $\rightarrow$ bool
`inverse m`	Map inverse	inmap A to B $\rightarrow$ inmap B to A

Examples: Let

```
m1 = { <France> |-> 9, <Denmark> |-> 4,
       <SouthAfrica> |-> 2, <SaudiArabia> |-> 1},
m2 = { 1 |-> 2, 2 |-> 3, 3 |-> 4, 4 |-> 1 },
Europe = { <France>, <England>, <Denmark>, <Spain> }
```

then:

```
dom m1      ≡   {<France>, <Denmark>,
                 <SouthAfrica>,
                 <SaudiArabia>}

rng m1      ≡   {1,2,4,9}
```

product / tuple

`T = A1 * A2 * ... * An`

Operator	Name	Type
`t.#n`	Select	$T * \mathsf{nat} \rightarrow Ti$
`t1 = t2`	Equality	$T * T \rightarrow \mathsf{bool}$
`t1 <> t2`	Inequality	$T * T \rightarrow \mathsf{bool}$

Examples: Let `a = mk_(1, 4, 8)`, `b = mk_(2, 4, 8)` then:

$$
\begin{array}{lcl}
\mathsf{a\ =\ b} & \equiv & \mathsf{false} \\
\mathsf{a\ <>\ b} & \equiv & \mathsf{true} \\
\mathsf{a\ =\ mk\_(2,4)} & \equiv & \mathsf{false}
\end{array}
$$

record

```
A :: first  : A1
     second : A2
```

Operator	Name	Type
`r.i`	Field select	$A * Id \to Ai$
`r1 = r2`	Equality	$A * A \to \textsf{bool}$
`r1 <> r2`	Inequality	$A * A \to \textsf{bool}$
`is_A(r1)`	Is	$Id * MasterA \to \textsf{bool}$

Examples: Let `Score` be defined as

```
Score :: team : Team
         won : nat
         drawn : nat
         lost : nat
         points : nat;
Team = <Brazil> | <France> | ...
```

and let

`sc1 = mk_Score (<France>, 3, 0, 0, 9)`,
`sc2 = mk_Score (<Denmark>, 1, 1, 1, 4)`,
`sc3 = mk_Score (<SouthAfrica>, 0, 2, 1, 2)` and
`sc4 = mk_Score (<SaudiArabia>, 0, 1, 2, 1)`.

Then

`sc1.team`	$\equiv$	`<France>`
`sc4.points`	$\equiv$	`1`
`sc2.points > sc3.points`	$\equiv$	true
is_Score(`sc4`)	$\equiv$	true
is_bool(`sc3`)	$\equiv$	false
is_int(`sc1.won`)	$\equiv$	true
`sc4 = sc1`	$\equiv$	false
`sc4 <> sc2`	$\equiv$	true

union

`B = A1 | A2 | ... | An`

Operator	Name	Type
`t1 = t2`	Equality	$A * A \to \mathsf{bool}$
`t1 <> t2`	Inequality	$A * A \to \mathsf{bool}$

Examples: In this example `Expr` is a union type whereas `Const`, `Var`, `Infix` and `Cond` are composite types defined using the shorthand `::` notation.

```
Expr  = Const | Var | Infix | Cond;
Const :: nat | bool;
Var   :: id:Id
         tp: [<Bool> | <Nat>];
Infix :: Expr * Op * Expr;
Cond  :: test : Expr
         cons : Expr
         altn : Expr
```

and let `expr = mk_Cond(mk_Var("b",<Bool>),mk_Const(3), mk_Var("v",nil))` then:

```
is_Cond(expr)          ≡  true
is_Const(expr.cons)    ≡  true
is_Var(expr.altn)      ≡  true
is_Infix(expr.test)    ≡  false
```

Useful Expression Styles

let-in

Syntax: `let p1 = e1, ..., pn = en in e`

Example:

```
map_disj : (map nat to nat) * (map nat to nat) -> map nat to nat
map_disj (m1,m2) ==
  let inter_dom = dom m1 inter dom m2
  in
    inter_dom <-: m1 munion
    inter_dom <-: m2
pre forall d in set dom m1 inter dom m2 & m1(d) = m2(d)
```

let-be-such-that

Syntax: `let b be st e1 in e2`

Example:

```
remove : nat * seq of nat -> seq of nat
remove (x,l) ==
  let i in set inds l be st l(i) = x
  in l(1,...,i-1)^l(i+1,...,len l)
pre  x in set elems l;
```

Define Expression

Syntax:
```
def pb1 = e1;
    ...
    pbn = en
in
    e
```

Example:

```
def user = lib(copy)
in
    if user = <OUT> then true else false
```

If-Then-Else, Case

Syntax:
```
if e1
then e2
else e3
```

Example:

```
lmerge : seq of nat * seq of nat -> seq of nat
lmerge (s1,s2) ==
  if s1 = [] then s2
  elseif s2 = [] then s1
  elseif (hd s1) < (hd s2)
  then [hd s1]^(lmerge (tl s1, s2))
  else [hd s2]^(lmerge (s1, tl s2));
```

Case

Syntax:
```
cases e :
  p11, p12, ..., p1n -> e1,
  ...                -> ...,
  pm1, pm2, ..., pmk -> em,
  others             -> emplus1
end
```

Example:

```
mergesort : seq of nat -> seq of nat
mergesort (l) ==
  cases l:
    [] -> [],
    [x] -> [x],
    l1^l2 -> lmerge (mergesort(l1), mergesort(l2))
  end
```

Sets

Modelling using sets

- Sets:
  - The finite set type constructor
  - Value definitions: enumeration, subrange, comprehension
  - Operators on sets

To define a type:

- *a type constructor*
- *ways of writing down values*
- *ways of operating on values*

The idea of a set ...

An unordered collection of values:

The order doesn't matter:

Nor do duplicates:

The set type constructor

The finite set type constructor is: `set of _`

What are the types of the following expressions?

```
{1, -3, 12}
```

```
{ {9, 13, 77}, {32, 8}, {}, {77} }
```

The set type constructor

The type `set of X` is the class of all possible finite sets of values drawn from the type X. For example:

`set of nat1`	sets of non-zero Natural numbers
`set of Student`	sets of student records
`set of (seq of char)`	sets of sequences of characters (e.g. sets of names)
`set of (set of int)`	sets of sets of integers, e.g. `{ {3,56,2},{-2},{},{-33,5} }`

Defining sets …

(0) Empty Set: `{}`

(1) Enumeration: `{1,2,3,4,5}`

`{‘a’,’b’,’c’}`

(2) Subrange (integers only): `{integer1,...,integer2}`

e.g. `{12,...,20} =`

`{12,...,12} =`

`{9,...,3} =`

Defining sets …

(3) Comprehension

{ *expression* | *binding* & *predicate* }

The set of values of the expression under each assignment of values to bound variables satisfying the predicate.

Consider all the values that can be taken by the variables in the binding.

Restrict this to just those combinations of values which satisfy the predicate.

Evaluate the expression for each combination. This gives you the values in the set.

e.g. `{ x**2 | x:nat & x < 5 }`

Defining sets …

Examples of Comprehensions:

```
{x | x:nat & x < 5}

{x | x in set {0,...,4} }

{x | x in set {1,...,15} & x < 5}

{y | y:nat & y < 0}

{x+y | x,y:nat & x<3 and y<4}

{y | y in set {1,...,20} &
                  exist x in set {1,...,3} & x*2=y}
```

Defining sets

Finiteness

In VDM-SL, sets should be finite, so be careful when writing comprehensions that you don't define a predicate that could be satisfied by an infinite number of values.

Example: `{x | x:nat & x > 10}`

Define a type with invariant instead if you need infinite values

Example:

```
BigNat = nat
inv x == x > 10
```

Operators on Sets

There are plenty of built-in operators on sets.

Each one has a *signature* defining the number and types of operand expected, e.g. the set union operator:

```
_ union _ : set of A * set of A -> set of A
```

Name of the operator. The underscores show where the arguments go when the operator is used.

Types of the inputs, in order. The "*" separates each input type.

The type of the result.

What can you tell about the union operator from this signature?

Operators on Sets

```
_ union _ : set of A * set of A -> set of A
```

Are the following expressions legal, according to the signature?

```
union({4, 7, 9} {23, 6})
3 union {7, 1, 12}
{12,…,15} union {x-y | x,y:nat & x<4 and y<10}
{} union {}
{12} union {x**y | x,y:nat & x<4 and y>2}
```

Operators on Sets

```
_ union _   : set of A * set of A -> set of A
_ inter _   : set of A * set of A -> set of A
_ \ _       : set of A * set of A -> set of A
dunion      : set of (set of A)   -> set of A
dinter      : set of (set of A)   -> set of A
card        : set of A            -> nat
_ in set _ : A * set of A         -> bool
_ subset _ : set of A * set of A -> bool
```

Note: *we don't show the underscores when the operator is normally used in a prefix form, e.g.*

```
card {12, 45, 12, 3} = 4
```

Operators on Sets

distributed operators

The most common operators have special forms in which they are extended to a whole set of arguments, not just two.

```
dunion    : set of (set of A)   -> set of A

dinter    : set of (set of A)   -> set of A
```

Operators on Sets

In side a function definition, we may need to select an *arbitrary* element from a set, not caring how it is selected. We can do this by using a local definition, i.e. in the body of the function say

```
let x in set S in …
```

(now x *stands for some arbitrary member of* S*)*

Alternatively, we could just define a general function for selecting an element from a set. Since we are not interested in the means of selection, we could do this by an implicit function definition:

```
Select (s:set of X) result:X

pre s <> {}

post result in set s
```

... and now we can use `Select(_)` whenever we want to select an element of a set.

Sequences

Modelling using Sequences

- Sequences
  - The finite sequence constructor
  - Value definitions: enumeration, subsequence
  - Operators on Sequences

The finite sequence type constructor

In VDM-SL, a **sequence** is a finite ordered collection of values. The presence of duplicates and the order in which elements are presented is significant.

The finite sequence type constructor is:

```
seq of X
```

where `X` is an arbitrary type. The type `seq of X` is the class of all possible finite sequences of values drawn from the type `X`. For example:

```
seq of nat1

seq of (seq of char)
```

Finite sequence value definitions

Sequence values can be represented in various ways:

- **Enumeration**, e.g. `[3, 5, 2, 5, 45]`
  `[ {34}, {34,7}, “Fred” ]`
  empty sequence `[]`

- Sequences of characters may be given as strings in quotation marks, e.g.

  `[‘l’,’i’,’n’,’u’,’x’] = “linux”`

- **Subsequence**: If we have a sequence `q` then we can take an extract from `q`, e.g. `q(3,…,5)= [q(3), q(4), q(5)]`

- **Comprehension**: The sequence comprehension notation is not often used and is described in the text.

- Note that sequences, like sets, are *finite.*

Operators on finite sequences

`hd: seq of X -> X` — Partial operator: `s <> []`
First element

`tl: seq of X -> seq of X` — Partial operator: `s <> []`
Tail (NB: a sequence!)

`len: seq of X -> nat` — length of sequence

`elems: seq of X -> set of X` — elements in the sequence (reduced to a set)

`inds: seq of X -> set of nat` — indices of the sequence
{1,…,len s}

`_ ^ _ : seq of X * seq of X -> seq of X`
sequence concatenation

`conc: seq of (seq of X) -> seq of X`
`conc s =` the concatenation of all the sequences in `s`

Mappings

Modelling using Mappings

- Mappings:
  - The finite mapping type constructor
  - Value definitions: enumeration, comprehension
  - Operators on mappings

The finite mapping type constructor

A mapping is a functional relationship between two sets of values: a domain and a range. Mappings are common in many models, e.g.

“Each bank account has exactly one balance”

“Each reactor has an input, an output and an operating temperature.”

Mappings represent one-to-one or many-to-one relationships, *but not one-to-many!*

The finite mapping type constructor

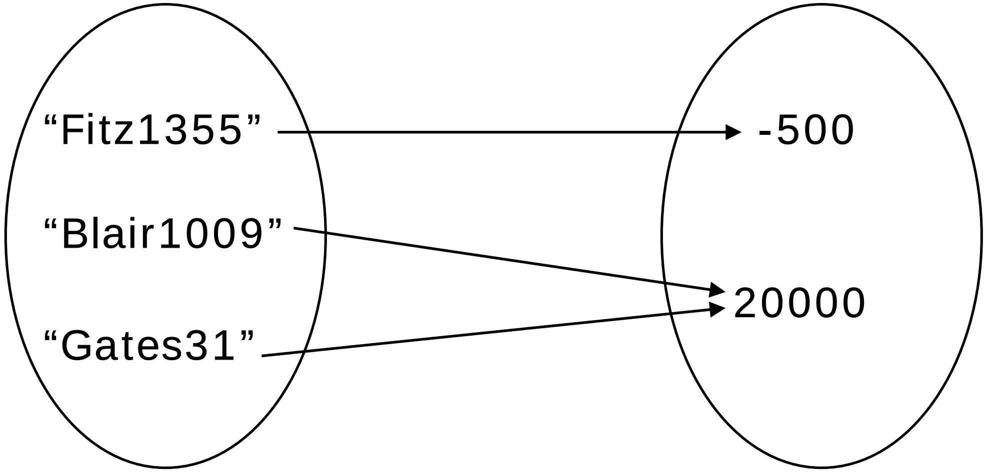
The mapping type constructor is **map** X **to** Y

where X and Y are data types

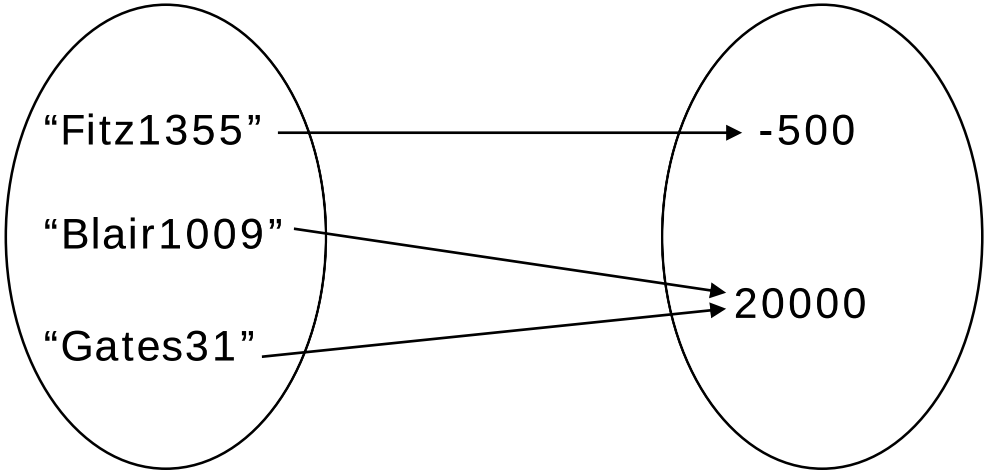
e.g. “each bank account has exactly one bank balance”:

```
AccountNumber = seq of char
Balance = int
Accounts = map AccountNumber to Balance
```

An example mapping:



Value definitions: enumeration, comprehension



To enumerate a mapping, we present the related domain element-range element pairs (called *maplets).* For the mapping illustrated above, the enumeration would be:

```
{“Fitz1355“ |-> -500, “Blair1009” |-> 20000, “Gates31” |-> 20000}
```

A maplet relating domain element **x** to range element **y** is written

```
x |-> y
```

Value definitions: enumeration, comprehension

A mapping comprehension has the following form:

{ *expression* |-> *expression* | *binding* & *predicate* }

The mapping consisting of the maplets formed by evaluating the expressions under each assignment of values to bound variables satisfying the predicate.

Consider all the values that can be taken by the variables in the binding.

Restrict this to just those combinations of values which satisfy the predicate.

Evaluate the expressions for each combination. This gives you the maplets in the mapping.

e.g. `{ x |-> x/2 | x:nat & x < 5 }`

Value definitions: enumeration, comprehension

Like sets and sequences, mappings are finite. Are the following mappings defined?

```
{ x |-> x**2 | x:nat1 & x**2 > 3 }
```

```
{ x |-> y | x,y:nat1 & x<4 and y<3 }
```

```
{ x |-> x**2 | x:int & x < 10 }
```

Operators on mappings

```
dom: map A to B  ->  set of A
```

Domain

```
rng: map A to B  ->  set of A
```

Range

Evaluate the following:

```
dom { n |-> 3*n | n:nat & n<50 }

rng { n |-> 3*n | n:nat & n<50 }
```

Operators on mappings

```
_(_) : map A to B * A  ->  B
```

Mapping Lookup

For a mapping `m` and a domain element `a`, the expression

```
m(a)
```

denotes the range element pointed to by `a`.

*Is this a total or a partial operator?

Operators on mappings

Example of mapping lookup:

```
Accounts = map AccountNumber to Balance
```

Define a function with the following signature which returns the names of overdrawn account holders:

```
overdrawn : Accounts -> set of AccountNumber
```

```
overdrawn(acs) ==
       {a | a in dom acs & acs(a) < 0 }
```

Operators on mappings

The mapping merge or mapping union operator joins two mappings together:

```
_ munion _ : (map A to B) * (map A to B) -> (map A to B)
```

Example:

```
{ “John” |-> -500, “Tony” |-> 20000 }
munion { “Cherie” |-> 150 }
= { “John” |-> -500, “Tony” |-> 20000, “Cherie” |-> 150 }
```

This operator is partial. Can you see why?

Operators on mappings

Mapping union is only defined on inputs that are *compatible.* We can define a function to check for mapping compatibility:

```
compatible: (map A to B) * (map A to B) -> bool
compatible(m1,m2) ==
    forall x in set dom m1 inter dom m2 & m1(x) = m2(x)
```

Operators on mappings

An alternative operator is the mapping override operator:

```
_ ++ _ : (map A to B) * (map A to B) -> (map A to B)
```

This operator is defined just like `munion`, except that where `m1` and `m2` are not compatible, `m2` wins, e.g.

```
{ “John” |-> -500, “Tony” |-> 20000 }
++ { “Tony” |-> 300, “Cherie” |-> 150 }
= { “John” |-> -500, “Tony” |-> 300, “Cherie” |-> 150 }
```

A very common use of this operator is to update a mapping at a point, e.g.

```
m ++ {x |-> e}
```

updates the mapping `m` so that `x` now points to `e`.

Operators on mappings

There are some operators to modify mappings by restricting the domain or range:

```
_ <-: _ : (set of A) * (map A to B) -> (map A to B)
```

The expression `s <-: m` is the same as `m` except that the elements of `s` have been removed from its domain (and any unattached range elements are removed too).

```
_ <: _ : (set of A) * (map A to B) -> (map A to B)
```

The expression `s <: m` is the same as `m` except that the domain is restricted down to just the elements of `s` (and any unattached range elements are removed too).

```
_ :-> _ : (map A to B) * (set of B) -> (map A to B)
```

The expression `m :-> s` is the same as `m` except that the elements of `s` have been removed from its range (and any unattached domain elements are removed too).

```
_ :> _ : (map A to B) * (set of B) -> (map A to B)
```

The expression `m :> s` is the same as `m` except that the range is restricted down to just the elements of `s` (and any unattached domain elements are removed too).

Details are in the text.

Operators on mappings

Example:

Define a function returning the accounts mapping for those account holders who are not overdrawn.

```
AccountNumber = seq of char

Balance = int

Accounts = map AccountNumber to Balance


credit-map : Accounts -> Accounts

credit-map (acs) ==
```

```
credit-map(acs) ==
      {a |-> acs(a) | a in dom acs & acs(a) >= 0 }
```

Modelling State

State, Functions, and Operations

Explicit Function Definitions

- VDM features a (functional/procedural) programming language
- Function definitions include a signature and the expression
  - Syntax of explicit function

```
f: X1 * ... * Xn -> R
f(x1, ..., xn) == ...
```

  - Example

```
mult: nat * nat -> nat
mult(x, y) ==
      if y = 1 then x else mult(x, y - 1) + y
```

Implicit Function Definitions

- Sometimes one does not want / know how to define a function Implicit function definitions allow to express what is to be computed, not how
  - Syntax of implicit function

```
f(x1: X1, ..., xn: Xn) res: R
pre P(x1, ..., xn)
post Q(x1, ..., xn, res)
```

  - Example

```
mult(x: nat , y: nat) res: R
pre true
post res = x * y
```

Implicit+Explicit Function Definitions

- Both implicit and explicit can be used at the same time
  - Syntax

```
f: X1 * ... * Xn -> R
f(x1, ..., xn) == ...
pre P(x1, ..., xn)
post Q(x1, ..., xn, RESULT)
```

  - Example

```
mult: nat * nat -> nat
mult(x, y) ==
       if y = 1 then x else mult(x, y - 1) + y
pre true
post RESULT = x * y
```

Limitations of functional style

- So far, the models we have looked at have used a high level of abstraction.
- Functionality has been largely modelled by *explicit functions*, e.g.

```
Update: System * Input -> System
Update(oldsys, val) == mk_System( … )
```

- Few computing systems are implemented using only pure functions

Persistent state

- More often, “real” systems have variables holding data, which may be modified by operations invoked by a user
- VDM-SL provides facilities for state-based modelling:
  - state definition
  - operations
  - auxiliary definitions (types, functions)

Example: Alarm Clock

- An alarm clock keeps track of current time and allows user to set an alarm time
- The alarm could be represented as a record type:

```
Time = nat
Clock :: now : Time
         alarm: Time
         alarmOn: bool
```

- Instead, we will use a state-based model

State definition

- State is defined with the following syntax:

```
state Name of
      component-name : type
      component-name : type
      …
      component-name : type
inv optional-invariant
init initial-state-definition
end
```

- Definition introduces a new type (`Name`) treated as a record type, with state variables as fields

State definition

- Variables which represent the state of the system are collected into a state definition
- This represents the *persistent data*, to be read or modified by operations

```
state Clock of
  now : Time
  alarm : Time
  alarmOn : bool
init cl == cl = mk_Clock(0,0,false)
end
```

- A model has only one state definition
- `init` clause sets initial values for persistent data

Operations

- Procedures which can be invoked by the system's users – human or other systems – are *operations*
- Operations (can) take input and generate output
- Operations have *side effects* – can read, and modify, state variables

- Operations may be *implicit* or *explicit*, just as with functions

Explicit Operations

- An explicit operation has signature and body just like an explicit function, but the body need not return a value:
- e.g. alarm is set to a given time, which must be in the future:

```
SetAlarm: Time ==> ()
SetAlarm(t) == (alarm :=t ; alarmOn := true)
pre t > now
```

- Note features:
  - no return value (in this case)
  - sequence of assignments to state variables

Implicit Operations

- Explicit operations produce a relatively concrete model
- Normally in state-based model, begin with implicit operations
  - better abstraction
  - not executable
- e.g. implicit version of SetAlarm operation:

```
SetAlarm(t: Time)
ext wr alarm: Time
    wr alarmOn: bool
    rd now: Time
pre t > now
post alarm = t and alarmOn
```

Implicit operations

- Implicit operations have the following components:

  header with operation name, names and types of input and any result parameters

  externals clause lists the state components used by the operation, and whether they are read-only or may be modified by the operation

  pre-condition recording the conditions assumed to hold when the operation is applied

  post-condition relates state and result value after the operation is completed, to the initial state and input values. Post-condition must respect restrictions in the externals clause, and define “after” values of all writeable state components

Implicit operation syntax

```
OpName(param:type, param:type, …) result:type
ext wr/rd state-variable:type
    wr/rd state-variable:type
    ...
    wr/rd state-variable:type
pre logical-expression
post logical-expression
```

- Operation definition is a specification for a piece of code with input/output parameters and side effects on state components

Alarm clock: modelling time

- We might also model the passing of time:
- Implicit operation:

```
Tick()
ext wr now: Time
post now = now~ + 1
```

- Explicit operation:

```
Tick: () ==> ()
Tick() == now := now + 1;
```

State-based modelling

- Development usually proceeds from abstract to concrete
    - identify state and persistent state variables
    - define implicit operations in terms of their access to state variables, pre and postconditions
    - complete definitions of types, functions noting any restrictions to be captured as invariants and preconditions, check internal consistency
    - transform implicit operations to explicit (and to code) by provably correct steps

Validation

The Idea of Validation

How confident can you be that a formal model accurately describes the system that the customer wanted?

- Requirements are often incomplete and ambiguous: modellers have to resolve these in unambiguous models.
- Requirements often state the client's intention incorrectly.

Validation is the process of increasing confidence that a model is an accurate representation of the system under consideration. Two aspects of this:

1. Checking internal consistency of a model.
2. Checking that the model describes the required behaviour of the system being modelled.

Internal Consistency

If a modelling language is formal then it must have:

• a formal syntax: rules restricting the symbols in the language and saying where they can be used.

• a formal semantics: rules for determining the meaning of a model written in accordance with the formal syntax.

If the syntax is formal, then we can check it with the aid of a tool (c.f. *syntax checker* in a programming language compiler).

If the semantics is formal, then we can check at least some aspects with the aid of a tool (c.f. *type checker* in a programming language compiler).

But we can't check everything!

Internal Consistency: Behaviour

The other aspect of validation is checking the accuracy with which the model records the desired system behaviour.

We will look at three approaches:

- **Animating** the model - *works well with clients unfamiliar with the modelling notation but requires a good interface.*

- **Testing** the model - *can assess coverage but limited to the quality of the tests and the model must be executable.*

- **Proving** properties of the model - *provides excellent coverage and does not require executability, but not well supported by tools.*

Internal Consistency: Type checking

A simple form of internal consistency checking is type checking. Consider a type checking tool working on the following extracts from function definitions:

```
Student :: …

Sid = token

Dbase = map Sid to Student

newStudent1: Sid * Student * Dbase -> Dbase
newStudent1(sid,s,db) == db ++ { sid |-> s }


newStudent2: Sid * Student * Dbase -> Dbase
newStudent2(sid,s,db) == db ^ { sid |-> s }


newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
```

Internal Consistency: Type checking

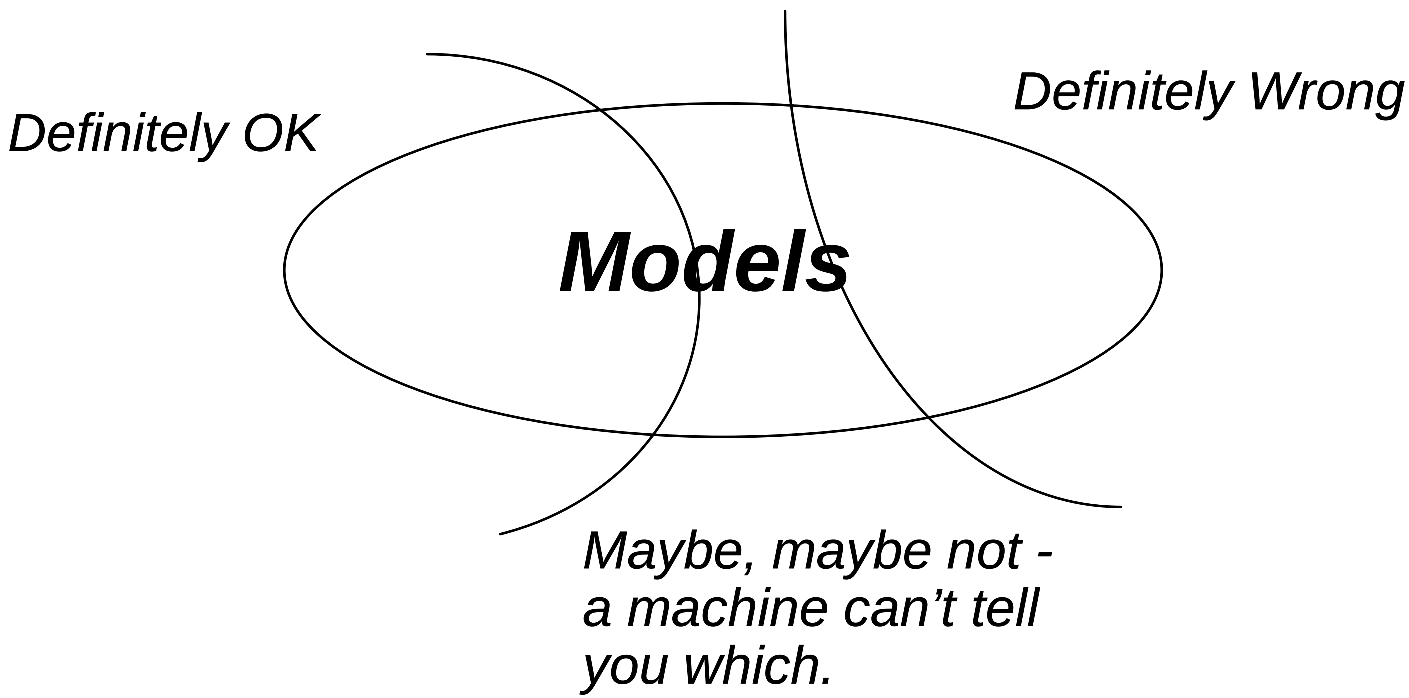
```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid in set dom db
```

We know that this is OK, but could a machine work it out? What about …

```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid in set {s1 | s1 : Sid &
    exists y in set rng db & db(s1) = y}
```

We can't provide a completely general tool that can automatically check that all uses of operators are properly protected (programming languages have the same problem - you can't produce a general tool that can automatically statically check whether division by zero will occur unless the language is very restricted and inexpressive).

Internal consistency: Type checking



Much of the current research in formal modelling aims to develop techniques and tools to reduce the size of the middle area by performing more and more checks automatically.

Internal consistency: Proof obligations

If a check cannot be performed automatically, the techniques of mathematical proof are required to complete it.

The collection of all checks to be performed on a VDM model are called *proof obligations.* A proof obligation is a logical expression which must be shown to hold before a VDM-SL model can be regarded as formally internally consistent.

We look at three proof obligations on VDM-SL models:

- **Domain Checking**
- **Satisfiability of explicit definitions**
- **Satisfiability of implicit definitions**

Proof Obligations: Domain Checking

Using a partial operator outside its domain of definition is usually an error on the part of the modeller. Two kinds of construct are impossible to check automatically:

- applying a function that has a pre-condition; and
- applying a partial operator.

Some definitions:

```
f:T1 * T2 * … * Tn -> R
f(a1,…,an) == ...
pre ...
```

We can refer to the precondition of `f` as a Boolean function with the following signature:

```
pre_f:T1 * T2 * … * Tn -> Bool
```

Proof Obligations: Domain Checking

Domain Checking for Functions with Pre-conditions

If a function `g` uses a function `f:T1*…*Tn -> R` in its body, occurring as an expression `f(a1,…,an)`, then it is necessary to show

```
pre-f(a1,…,an)
```

for any `a1,…,an` that can arise in this position.

Example:

```
Delete: Tracker * ContainerId * PhaseId -> Tracker
Delete(tkr,cid,source) ==
   mk_Tracker({cid} <-: tkr.containers,
              Remove(tkr,cid,source).phases)
pre pre_Remove(tkr,cid,source)
```

Proof obligation for domain checking:

```
forall tkr:Tracker, cid:ContainerId, source:PhaseId &
   pre_Delete(tkr,cid,source) => pre_Remove(tkr,cid,source)
```

Proof Obligations: Domain Checking

Domain Checking for Partial Operators

Each application of a partial operator must be protected. For example, consider:

```
Introduce: Tracker * ContainerId * real * Material -> Tracker
Introduce(trk,cid,quan,mat)==
   mk_Tracker(trk.containers munion
               {cid |-> mk_Container(quan,mat)},
               trk.phases)
pre cid not in set dom trk.containers
```

The obligation is:

```
forall trk:Tracker, cid:ContainerId, quan:real, mat:Material &
pre_Introduce(trk,cid,quan,mat) =>
  compatible(trk.containers,{cid |-> mk_Container(quan,mat)})
```

Proof Obligations: Domain Checking

Partial operators can be protected by pre-conditions (as in the Permission example) or by including an explicit check in the body of the function, e.g.

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest) ==
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity and
  containers(cid).material in set
     phases(dest).expected_materials
```

Proof obligation

```
forall mk_Tracker(containers,phases):Tracker,
         cid:ContainerId, dest:PhaseId &
         (cid in set dom containers and
          dest in set dom phases) => dest in set dom phases
```

Proof Obligations: Domain Checking

Exercise: What is the proof obligation generated by the **highlighted** expression below?

```
Move: Tracker * ContainerId * PhaseId * PhaseId -> Tracker

Move(trk,cid,ptoid,pfromid) ==

  let pha = mk_Phase(trk.phases(ptoid).contents union {cid},

                     trk.phases(ptoid).capacity)

  in

  mk_Tracker(trk.containers,

             Remove(trk,cid,pfromid).phases ++ {ptoid |-> pha})

pre Permission(trk,cid,ptoid) and pre_Remove(trk,cid,pfromid)
```

```
forall trk:Tracker, cid:ContainerId, ptoid:PhaseId &
       pre_Move(trk,cid,ptoid) =>
           ptoid in set dom trk.phases
```

Proof Obligations: Domain Checking

It can be difficult to decide what to include in a pre-condition.

- Some conditions are determined by the requirements.
- Many conditions are there to guard applications of partial operators and functions.

When you write a function definition, read through it systematically, highlighting each application of a partial operator, and ensure that you have guarded against misapplication of that operator by adding a suitable conjunct to the precondition.

Proof Obligations: Satisfiability

An explicit function *without* a pre-condition defined

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

is said to be ***satisfiable*** if, for all inputs, the result defined by the function body is of the correct type. Formally,

```
forall p1:T1,…,pn:Tn & f(p1,…,pn) : R
```

An explicit function *with* a pre-condition:

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

is said to be ***satisfiable*** if, for all inputs satisfying the pre-condition, the result defined by the function body is of the correct type. Formally,

```
forall p1:T1,…,pn:Tn &
  pre_f(p1,…,pn) => f(p1,…,pn) : R
```

Proof Obligations: Satisfiability

For example, consider

```
Introduce: Tracker * ContainerId * real * Material ->
           Tracker
Introduce(trk,cid,quan,mat)==
   mk_Tracker(trk.containers munion
               {cid |-> mk_Container(quan,mat)},
               trk.phases)
pre cid not in set dom trk.containers
```

The satisfiability proof obligation is:

```
forall mk_Tracker(containers, phases): Tracker,
       cid: ContainerId, quan: real, mat: Material &
       pre_Introduce(trk, cid, quan, mat) =>
           Introduce(trk,cid,quan,mat): Tracker
```

Proof Obligations: Satisfiability

- Most of the work in showing satisfiability comes in showing, not that the result returned belongs to the correct general type, but that it respects the invariant on that type.

Proof Obligations: Satisfiability

Satisfiability of implicit function definitions

A function `f` defined implicitly as follows

```
f(a1:T1,…,an:Tn) r:R
pre ...
post ...
```

is said to be ***satisfiable*** if, for all inputs satisfying the pre-condition, there exists a result of the correct type satisfying the post-condition. Formally,

```
forall p1:T1,…,pn:Tn &
   pre_f(p1,…,pn) =>
      exists x:R & post_f(p1,…,pn,x)
```

Proof Obligations: Satisfiability

Example:

```
Find(trk:Tracker,cid:ContainerId) p:(PhaseId|<NotAllocated>)
pre cid in set dom trk.containers
post if exists pid in set dom trk.phases &
               cid in set trk.phases(pid).contents
     then p in set dom trk.phases and
          cid in set trk.phases(p).contents
     else p = <NotAllocated>
```

The satisfiability proof obligation is as follows:

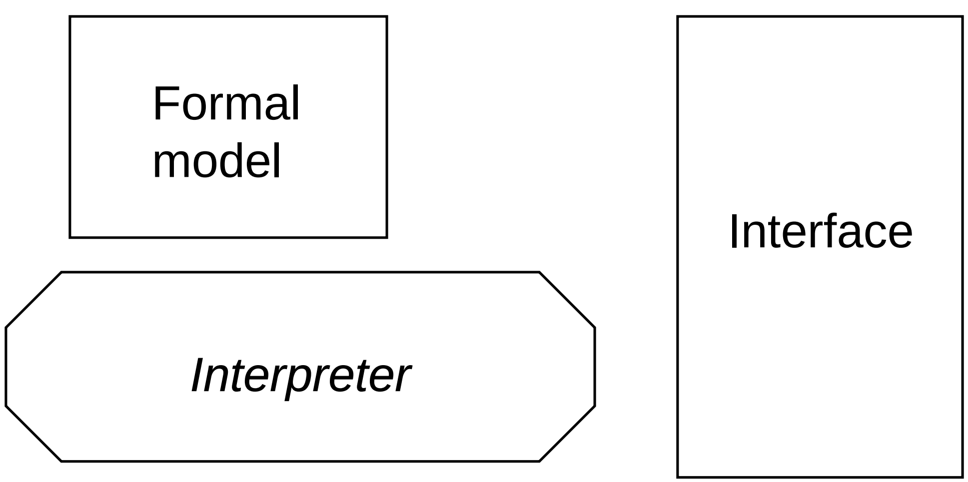
```
forall trk:Traceker, cid:ContainerId &
     pre_Find(trk,cid) =>
        exists p:(PhaseId|<NotAllocated>) &
             post_Find(trk,cid,p)
```

Animation

The goal of validation is to increase confidence that a model accurately reflects the customer's intentions.

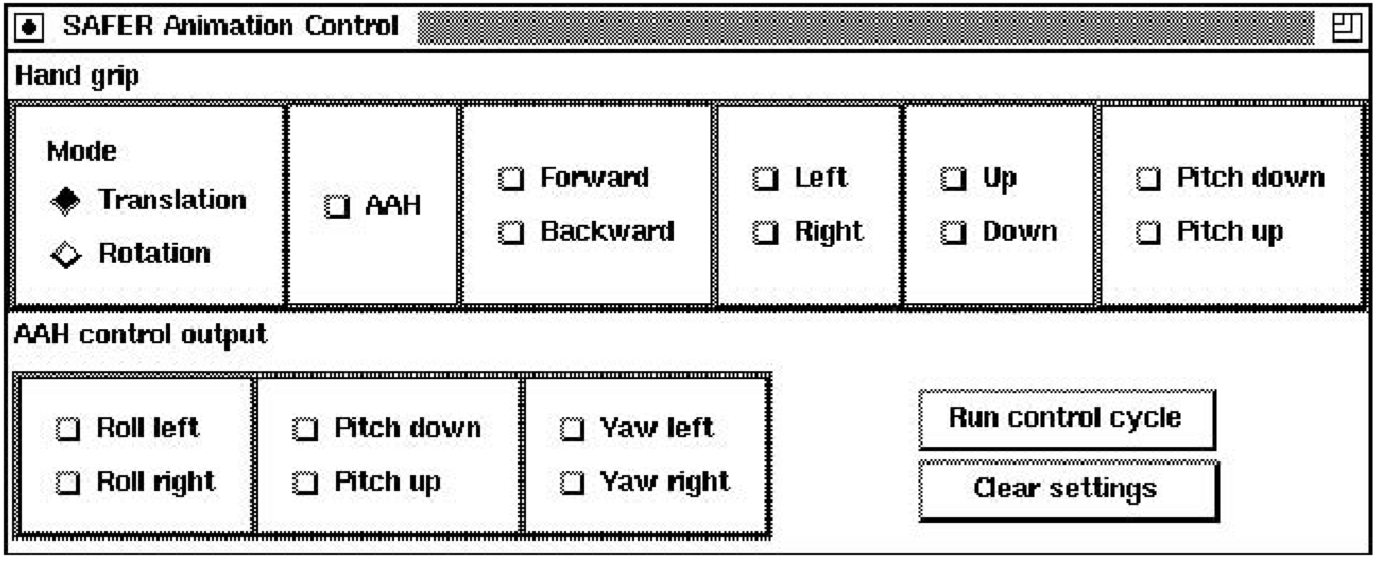
However, customers rarely understand the modelling language used, whether it is formal or not.

Animation is the execution of the model through an interface. The interface can be coded in a programming language of choice so long as a *dynamic link* facility exists for linking the interface code to the model.



Animation

The interface functions (in C++) have to be made known to the VDM-SL layer, This can be done in a dynamic link module which also provides a file name reference to the compiled C++ code.



Systematic Testing

The level of confidence gained through an animation is only as good as the particular choice of scenarios executed by the user through the interface.

More systematic testing is also possible:

define a collection of test cases

execute each test case on the formal model

compare with expectation

Test cases can be generated by hand or automatically. Automatic generation can however produce a vast number of individual test cases.

Techniques for test generation in functional programs carry over to formal models.

Dynamic type checking in executing a formal model can help validating proof obligations (but not proving them)

Systematic Testing

Executing the test:

```
Permission(mk_Tracker({|->},{|->}),mk_token(1),mk_token(2))
```

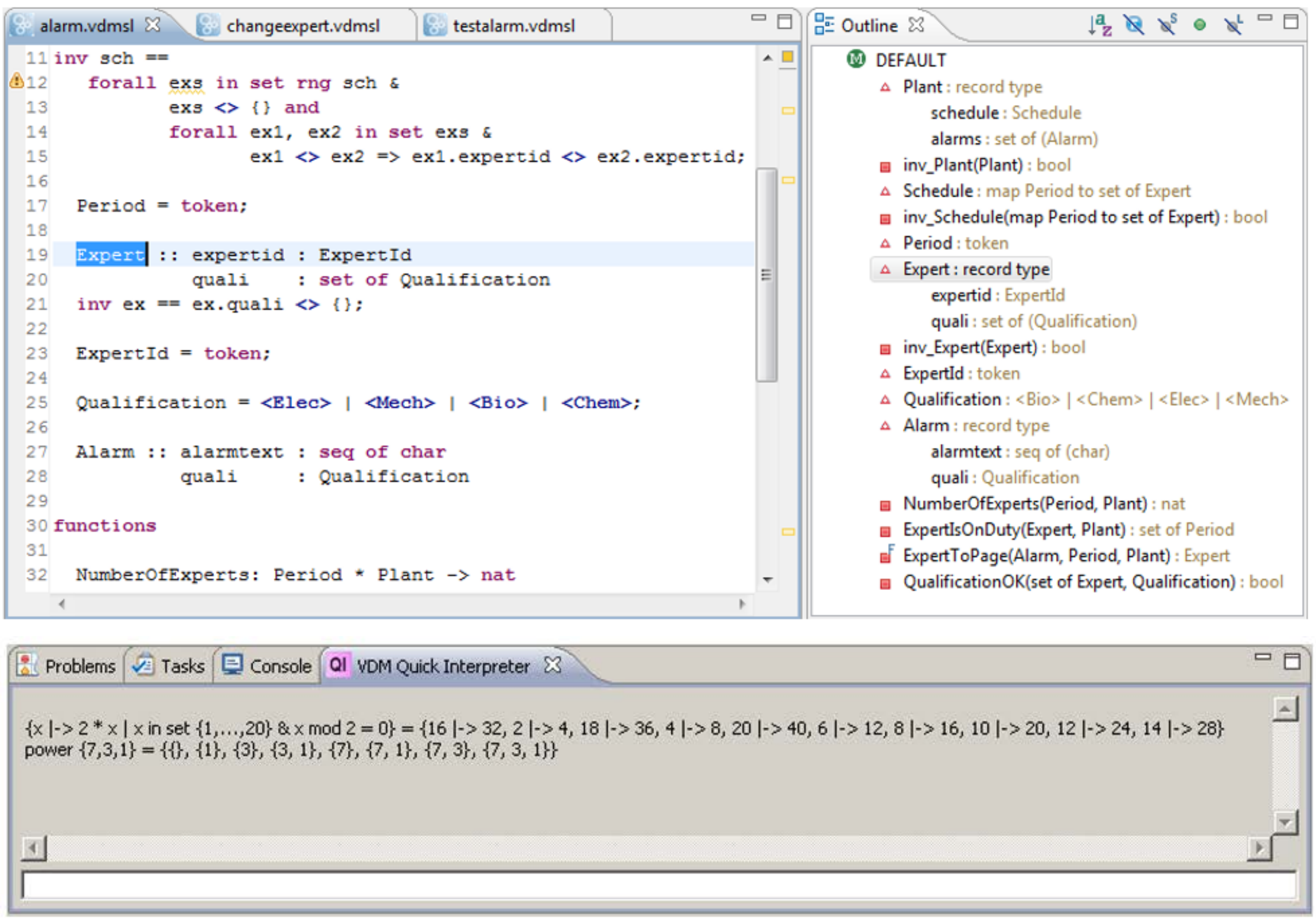
yields false. We can also tell which parts of the permission function have been exercised ("covered") by the test:

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest) ==
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity and
  containers(cid).material in set
     phases(dest).expected_materials
```

It is possible to have a tool highlight parts of the model that are not exercised by a test and use this information to devise other tests.

Systematic Testing with Tool Support

The Overture Tool



Validation by Proof

Systematic testing and animation are only as good as the tests and scenarios used. *Proof* allows the modeller to assess the behaviour of a the model for whole classes of inputs in one analysis.

In order to prove a property of a model, the property has to be formulated as a logical expression (like a proof obligation). A logical expression describing a property which is expected to hold in a model is called a *validation conjecture*.

Proofs can be time-consuming. Machine support is much more limited: it is not possible to build a machine that can automatically construct proofs of conjectures in general, but it is possible to build a tool that can check a proof once the proof itself is constructed. Considerable skill is required to construct a proof - but a successful proof gives high assurance of the truth of the conjecture about the model.

Summary

Validation: the process of increasing confidence that a model accurately reflects the client requirements.

- Internal consistency:
  - domain checking: partial ops and functions with precondition
  - satisfiability of explicit and implicit function
- Checking accuracy:
  - animation
  - testing
  - proof

Case Study:
the explosives storage example

Case Study: the explosives storage example

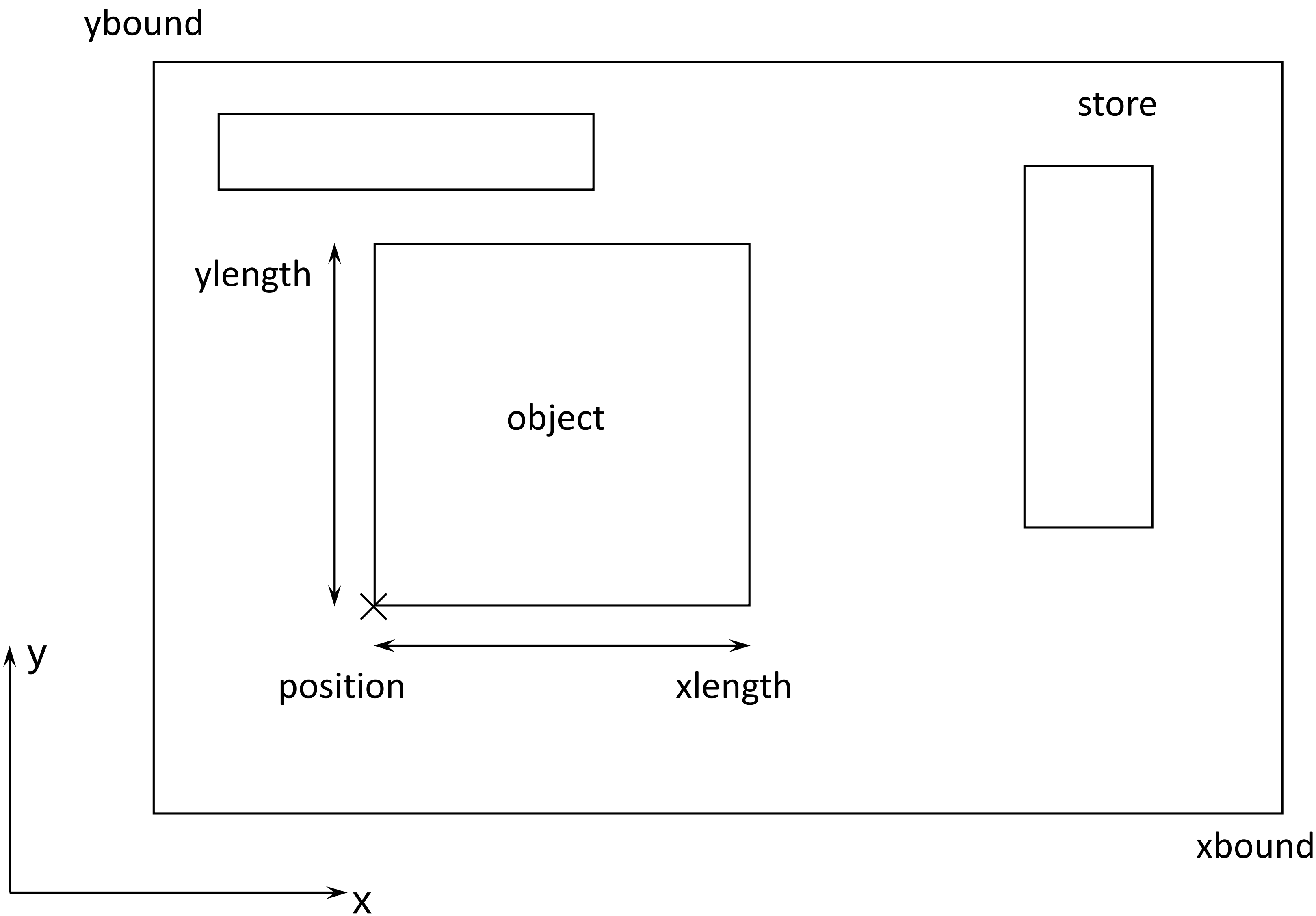
- The system to be modelled is part of a controller for a robot that positions explosives such as dynamite and detonators in a store.
- The store is a rectangular building. Positions within the building are represented as coordinates with respect to one corner designated the origin. The store's dimensions are represented as maximum x and y coordinates.
- Objects in the store are rectangular packages, aligned with the walls of the store. Each object has dimensions in the x and y directions. The position of an object is represented as the coordinates of its lower left corner. All objects must fit within the store and there must be no overlap between objects.

Case Study:
the explosives storage example

The positioning controller must provide functions to:

1. return the number of objects in a given store;
2. suggest a position where a given object may be accommodated in a given store;
3. update a store record to note that a given object has been placed in a given position;
4. update a store record to note that all the objects at a given set of positions have been removed.

Purpose of the model: to clarify the rules for the storage of explosives.


ybound
store
ylength
object
y
position
xlength
xbound
x

Case Study:
the explosives storage example

```
Store  :: contents :
          xbound   :
          ybound   :

Object :: position :
          xlength  :
          ylength  :
```

Case Study:
the explosives storage example

```
Store  :: contents :
          xbound   :
          ybound   :

inv mk_Store(contents,xbound,ybound) ==
```

Case Study:
the explosives storage example

```
Store  :: contents : set of Object
          xbound   : nat
          ybound   : nat

inv mk_Store(contents,xbound,ybound) ==
All objects in the store is within the bounds of the store
and
All objects in the store are not overlapped with each other
```

```
Point :: x : nat
         y : nat

Object :: position : Point
          xlength  : nat
          ylength  : nat
```

Case Study:
the explosives storage example

```
Store  :: contents :
          xbound   :
          ybound   :

inv mk_Store(contents,xbound,ybound) ==

     forall o in set contents &
          InBounds(o,xbound,ybound) and
     not exists o1, o2 in set contents &
          o1 <> o2 and Overlap(o1,o2)

InBounds: Object * nat * nat -> bool
InBounds(o,xb,yb) == ???

Overlap: Object * Object -> bool
Overlap(o1,o2) == ???
```

Case Study:
the explosives storage example

```
Store  :: contents :  set of Object
          xbound   :  nat
          ybound   :  nat

inv mk_Store(contents,xbound,ybound) ==

     forall o in set contents &
          InBounds(o,xbound,ybound) and
     not exists o1, o2 in set contents &
          o1 <> o2 and Overlap(o1,o2)

InBounds: Object * nat * nat -> bool
InBounds(o,xb,yb) == ???

Overlap: Object * Object -> bool
Overlap(o1,o2) == ???
```

```
InBounds(o,xb,yb) ==
       o.position.x + o.xlength <= xb and
       o.position.y + o.ylength <= yb
```

```
Overlap(o1,o2) == Points(o1) inter Points(o2) <> {}

Points: Object -> set of Point
Points(o) ==
       let o=mk_Object(pos,xlen,ylen) in
       {mk_Point(x,y) | x in set {pos.x,...,pos.x+xlen},
                        y in set {pos.y,...,pos.y+ylen} }
```

Case Study:
the explosives storage example

1. return the number of objects in a given store;

```
NumObjects: Store -> nat
```

2. suggest a position where a given object may be accommodated in a given store;

```
SuggestPos: nat * nat * Store -> Store
```

3. update a store record to note that a given object has been placed in a given position;

```
Place: Object * Store Point -> Store
```

4. update a store record to note that all the objects at a given set of positions have been removed.

```
Remove: Store * set of Point -> Store
```

Case Study:
the explosives storage example

```
NumObjects: Store -> nat

NumObject(s) == ???
```

Case Study:
the explosives storage example

```
NumObjects: Store -> nat

NumObject(s) ==  card s.contents
```

Case Study:
the explosives storage example

```
SuggestPos: nat * nat * Store -> Store

SuggestPos(xlength,ylength,s) == ???
```

There might be any number of viable positions, but the requirements are not specific about which one ought to be returned - any point with sufficient space will do.

Since we do not have to give a specific point, there is not need to give an algorithm for finding a suitable point: we can use an *implicit function definition* instead.

Case Study:
the explosives storage example

An implicit definition does not have a body, but does describe the result by means of a postcondition.

functionName (input vars & types) result & type

pre *precondition*

post *postcondition*

```
sqrt(x:real) r:real

pre   x >= 0

post  r*r = x
```

Case Study:
the explosives storage example

```
SuggestPos: nat * nat * Store -> Store

SuggestPos(xlength,ylength,s) == ???


SuggestPos(xlength:nat, ylength:nat, s:Store) p:[Point]

post  -- if there is a point with enough room
      -- then return some point where there is
      --      enough room
      -- else return nil

     if exists poss:Point &
               RoomAt(xlength,ylength,s,poss)

     then RoomAt(xlength,ylength,s,p)

     else p = nil
```

Case Study:
the explosives storage example

```
RoomAt: nat * nat * Store * Point -> bool

RoomAt(xlength,ylength,s,p) ==

      let new_o = mk_Object(p,xlength,ylength) in
```

Case Study: the explosives storage example

```
RoomAt: nat * nat * Store * Point -> bool

RoomAt(xlength,ylength,s,p) ==

      let new_o = mk_Object(p,xlength,ylength) in

      InBounds(new_o,s.xbound,s.ybound) and
      not exists o1 in set s.contents &
             Overlap(o1,new_o)
```

Case Study:
the explosives storage example

3. update a store record to note that a given object has been placed in a given position;

```
Place: Object * Store Point -> Store

Place(o,s,p) ==

    let new_o = mk_Object(p,o.xlength,o.ylength) in

    mk_Store (




              )

    pre
```

Case Study: the explosives storage example

3. update a store record to note that a given object has been placed in a given position;

```
Place: Object * Store * Point -> Store

Place(o,s,p) ==

    let new_o = mk_Object(p,o.xlength,o.ylength) in

    mk_Store (
               s.contents union {new_o},
               s.xbound,
               s.ybound
             )

    pre  RoomAt(o.xlength, o.ylength, s, p)
```

Case Study:
the explosives storage example

An extension - Suppose we have a site which consists of a collection of stores:

```
Store :: name : token
         ...

Site = set of Store
inv site ==
  forall store1, store 2 in set site &
      store1. name = store2.name => store1 = store2
```

and we need to take an inventory of the site:

```
Inventory = set of inventoryItem
InventoryItem :: store : token
                 item  : Object
```

Case Study: the explosives storage example

We could take the union of the individual inventories of each store:

```
SiteInventory: Site -> Inventory
SiteInventory(site) ==
 dunion{StoreInventory(store) | store in set site}


StoreInventory: Store -> Inventory
StoreInventory(store) ==
 {mk_InventoryItem(store.name,o) |
       o in set store.contents}
```

Case Study:
the explosives storage example

- Summary

- Use implicit specification (postcondition) when there is no need to give a particular result;

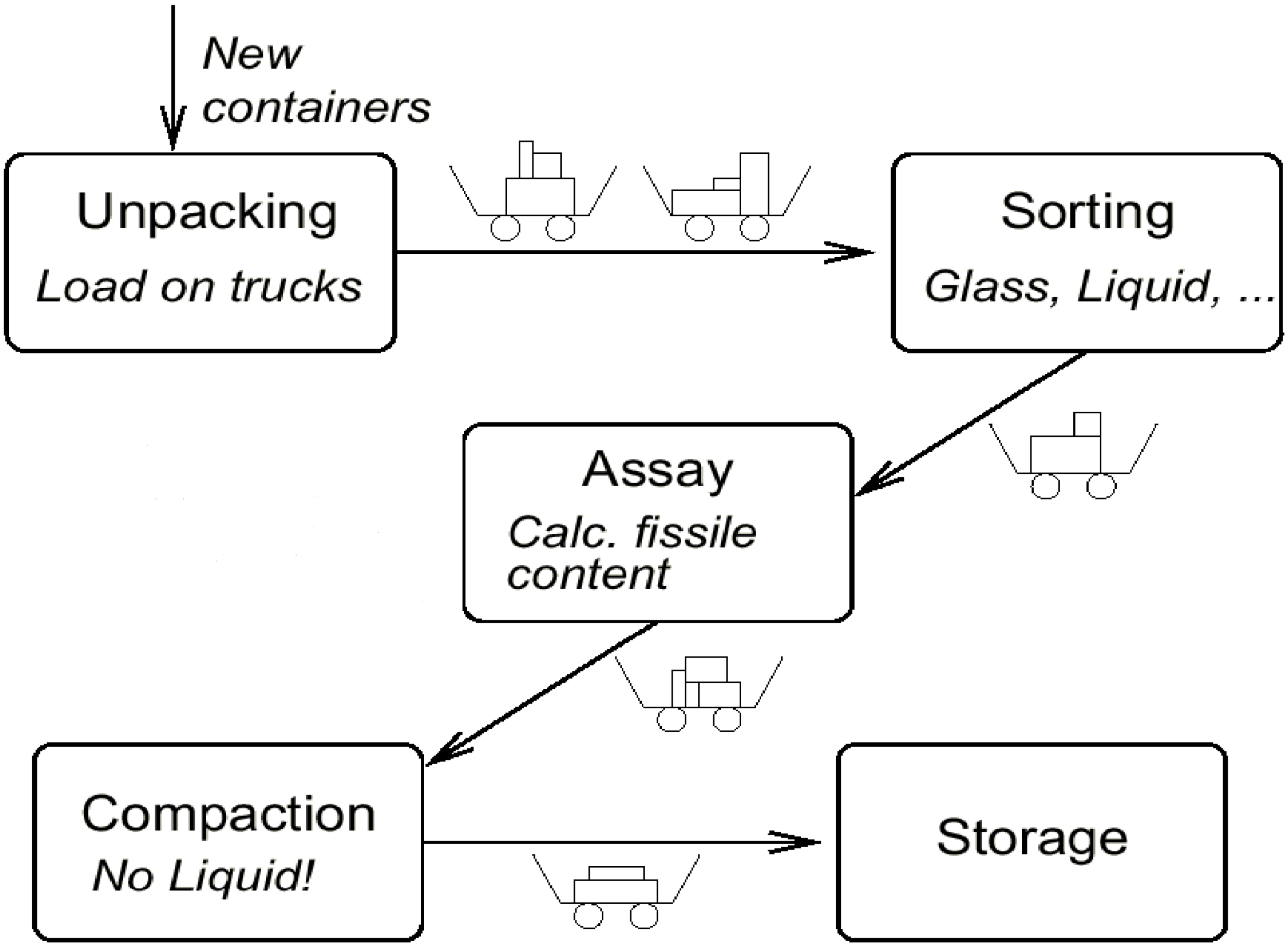
- Use auxiliary function definitions to break down and simplify the task of building the model.

The tracking manager example

The tracking manager example

A model of an architecture for tracking the movement of containers of hazardous waste as they go through reprocessing was developed by a team in Manchester Informatics with BNFL (British Nuclear Fuels Limited) in 1995.

The **purpose** of the model was to establish the rules governing the movement of containers of waste which the tracking manager would have to enforce. The model was safety-related, but note that the model was built simply in order to understand the problem better, not as a basis for software development. *Models don't just have to serve as specifications.*



New containers
Unpacking
Load on trucks
Sorting
Glass, Liquid, ...
Assay
Calc. fissile content
Compaction
No Liquid!
Storage

The tracking manager example

At the top level, the tracker holds information about containers and the phases of the plant:

```
Tracker :: containers : ContainerInfo
           phases     : PhaseInfo
```

The container and phase information is modelled as a mapping from identifiers to details *(this is a very common use of mappings, with identifiers in the domain and data types defining details in the range)*

```
ContainerInfo = map ContainerId to Container
PhaseInfo = map PhaseId to Phase
```

The tracking manager example

The details of how identifiers are represented are immaterial:

```
ContainerId =

PhaseId =
```

For each container, we record the fissile mass of its contents and the kind of material it contains.

```
Container =


Material
```

The tracking manager example

The details of how identifiers are represented are immaterial:

```
ContainerId = token
PhaseId = token
```

For each container, we record the fissile mass of its contents and the kind of material it contains.

```
Container :: fiss_mas : real
             material : Material
Material = token
```

The tracking manager example

Each phase houses a number of containers, expects certain material types and has a maximum capacity.

Try modelling this yourself:

The tracking manager example

Each phase houses a number of containers, expects certain material types and has a maximum capacity.

Try modelling this yourself:

```
Phase  ::     contents : set of ContainerId
              expected_materials : set of Material
              capacity = nat
```

The tracking manager example

In the real tracking manager project, domain experts from BNFL were closely involved with the development of the formal model. We relied on the domain experts to point out the safety properties that had to be respected by the tracker. For example, the number of containers in a phase should not exceed the phase's capacity:

```
Phase ::



inv p ==
```

The domain experts from BNFL often commented that this ability to record constraints formally as invariants was extremely valuable.

The tracking manager example

In the real tracking manager project, domain experts from BNFL were closely involved with the development of the formal model. We relied on the domain experts to point out the safety properties that had to be respected by the tracker. For example, the number of containers in a phase should not exceed the phase's capacity:

```
Phase  ::     contents : set of ContainerId
              expected_materials : set of Material
              capacity = nat

inv p = card p.contents <= p.capacity
```

The domain experts from BNFL often commented that this ability to record constraints formally as invariants was extremely valuable.

The tracking manager example

```
Tracker :: containers : ContainerInfo
           phases     : PhaseInfo
inv mk_Tracker(containers,phases) ==
      Consistent(containers,phases) and
      PhasesDistinguished(phases) and
      MaterialSafe(containers,phases)
```

Invariant:

1. All of the containers present in phases are known about in the `containers` mapping.
2. No two distinct phases may have any containers in common.
3. In each phase, all its containers contain materials as expected by the phase.

The tracking manager example

```
Consistent: ContainerInfo * PhaseInfo -> bool
Consistent(containers,phases) ==
 -- all of the containers present in phases are known
 -- about in the containers mapping.
    forall ph in set rng phases &
       ph.contents subset dom containers
```

The tracking manager example

```
PhasesDistinguished: PhaseInfo -> bool
PhaseDistinguished(phases) ==
 -- no two distinct phases may have any containers
 -- in common
   not exists p1, p2 in set dom phases &
      p1 <> p2 and
      phases(p1).contents inter phases(p2).contents <> {}
```

The tracking manager example

```
MaterialSafe: ContainerInfo * PhaseInfo -> bool
MaterialSafe(containers,phases) ==
 -- In each phase, all its containers contain materials
 -- as expected by the phase
   forall ph in set rng phases &
     forall cid in ph.contents &
       cid in set dom containers and
       containers(cid).material in set ph.expected_materials
```

The tracking manager example

- introduce a new container to the tracker, giving its identifier and contents;
- give permission for a container to move into a given phase;
- remove a container from a phase;
- delete a container from the plant.

```
Introduce: Tracker * ContainerId * real * Material
           -> Tracker
Introduce(trk, cid, quan, mat) ==
  mk_Tracker(
    trk.containers munion {cid |-> mk_Container(quan,mat)},
    trk.phases)
  pre cid not in set dom trk.containers
```

The tracking manager example

- give permission for a container to move into a given phase

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases),cid,dest) ==
 -- must check that the tracker invariant will be
 -- maintained by the move
    cid in set dom containers and
    dest in set dom phases and
    card phases(dest).contents < phases(dest).capacity and
    containers(cid).material in set
              phases(dest).expected_materials
```

container consistency

phase capacity

material safety

The tracking manager example

- remove a container from a phase

```
Remove: Tracker * Containerid * PhaseId -> Tracker
Remove(mk_Tracker(containers,phases),cid,pid) ==
 mk_Tracker(containers,
            phases ++ {pid |->

pre pid in set dom phases and
    cid in set phases(pid).contents
```

The tracking manager example

- remove a container from a phase

```
Remove: Tracker * Containerid * PhaseId -> Tracker
Remove(mk_Tracker(containers,phases),cid,pid) ==
 mk_Tracker(containers,
       phases ++ {pid |-> mk_Phase(
                            phases(pid).containers\cid,
                            phases(pid).expected_materials,
                            phases(pid).capacity
                          )
                          }
pre pid in set dom phases and
    cid in set phases(pid).contents
```

The tracking manager example

We can simplify function definitions by using a local declaration given in a **let** expression:

```
Remove: Tracker * Containerid * PhaseId -> Tracker
Remove(mk_Tracker(containers,phases),cid,pid) ==
let pha = mk_Phase(phases(pid).contents \ {cid},
                   phases(pid).expected_materials,
                   phases(pid).capacity)
in
mk_Tracker(containers, phases ++ {pid |-> pha})
pre pid in set dom phases and
    cid in set phases(pid).contents
```

The tracking manager example

To delete a container, two things have to be done:

- we have to remove the container from the `containers` mapping; and
- we have to remove the container from the phase in which it occurs (just as in the `Remove` function).

```
Delete: Tracker * Containerid * PhaseId -> Tracker
Delete(tkr, cid, pid) ==
mk_Tracker({cid} <-: containers,
           Remove(tkr, cid, pid).phases)
pre pre_Remove(tkr, cid, source)
```

*delete the element of key `cid`

*precondition of other functions can be used like this

The Overture Tool

The Overture Tool

- http://overturetool.org/download/
- The latest version: 2.5.0
- Interface based on Eclipse
- Functionalities
  - Editor with syntax highlight
  - Type check, Animation (Execution, Testing), Proof obligation generation (Integration check)

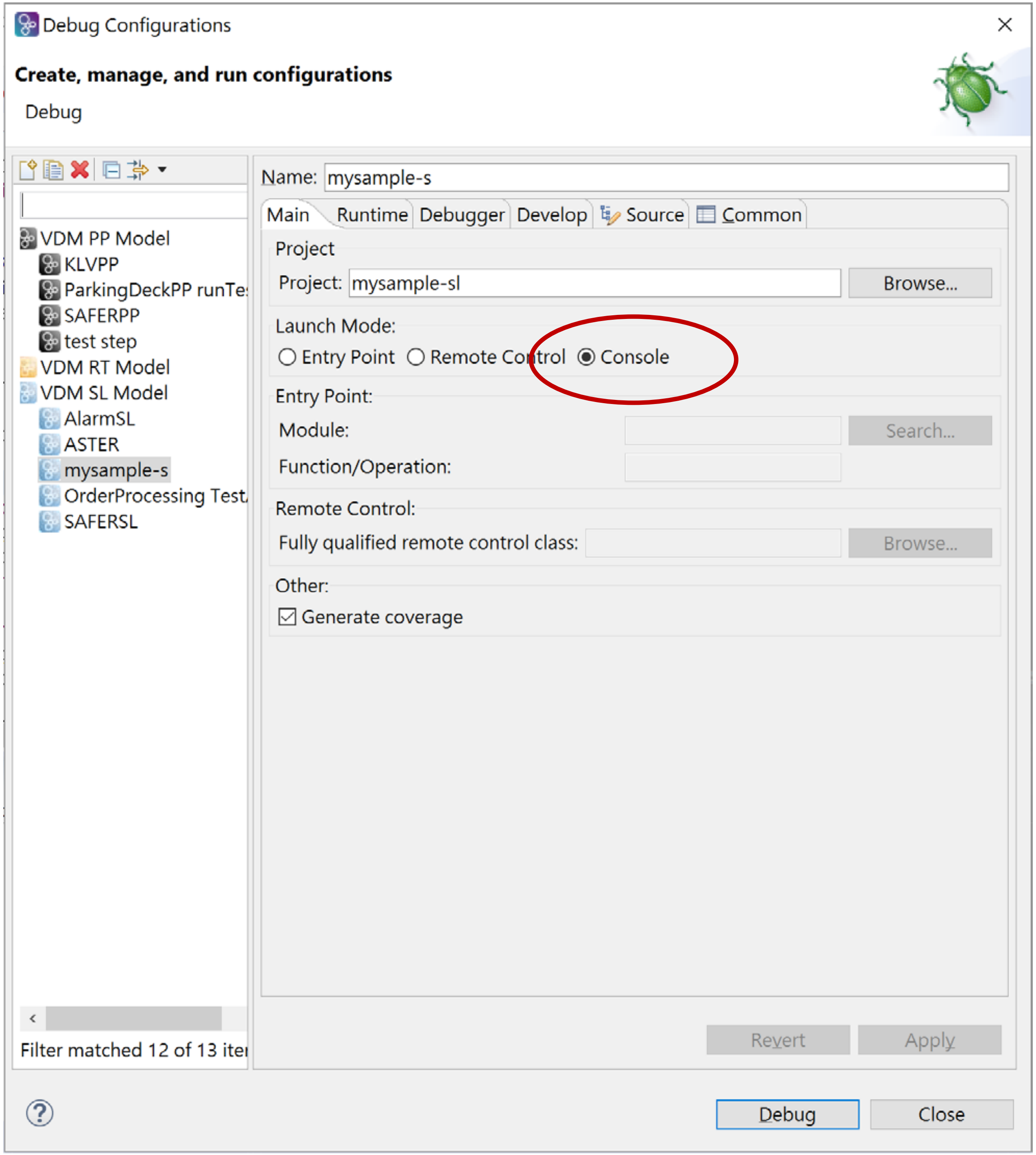
- How to start: unzip and execute

Start a VDM-SL Project

- File -> New -> Project -> VDM-SL Project
  - Set project name
  - …
  - Add file with extension “.vdmsl”

```
module CMDS
definitions

types
CMD = <R> | <L>;
CMD_series = seq of [CMD];
CMD_times = map CMD to nat;

state S of
  commands : CMD_series
  inv s == forall k in set {1,...,len s.commands - 1} &
           s.commands(k) <> s.commands(k+1)
  init p == p = mk_S([])
end

operations
  push_cmd(a:[CMD])
  pre commands = [] or hd commands <> a
  post hd commands = a and tl commands = commands~;

functions
times_count : CMD_series -> CMD_times
times_count(a) == { <R> |-> len [ i | i in set inds a & a(i)=<R> ],
                    <L> |-> len [ i | i in set inds a & a(i)=<L> ] };

end CMDS
```

Debug config.



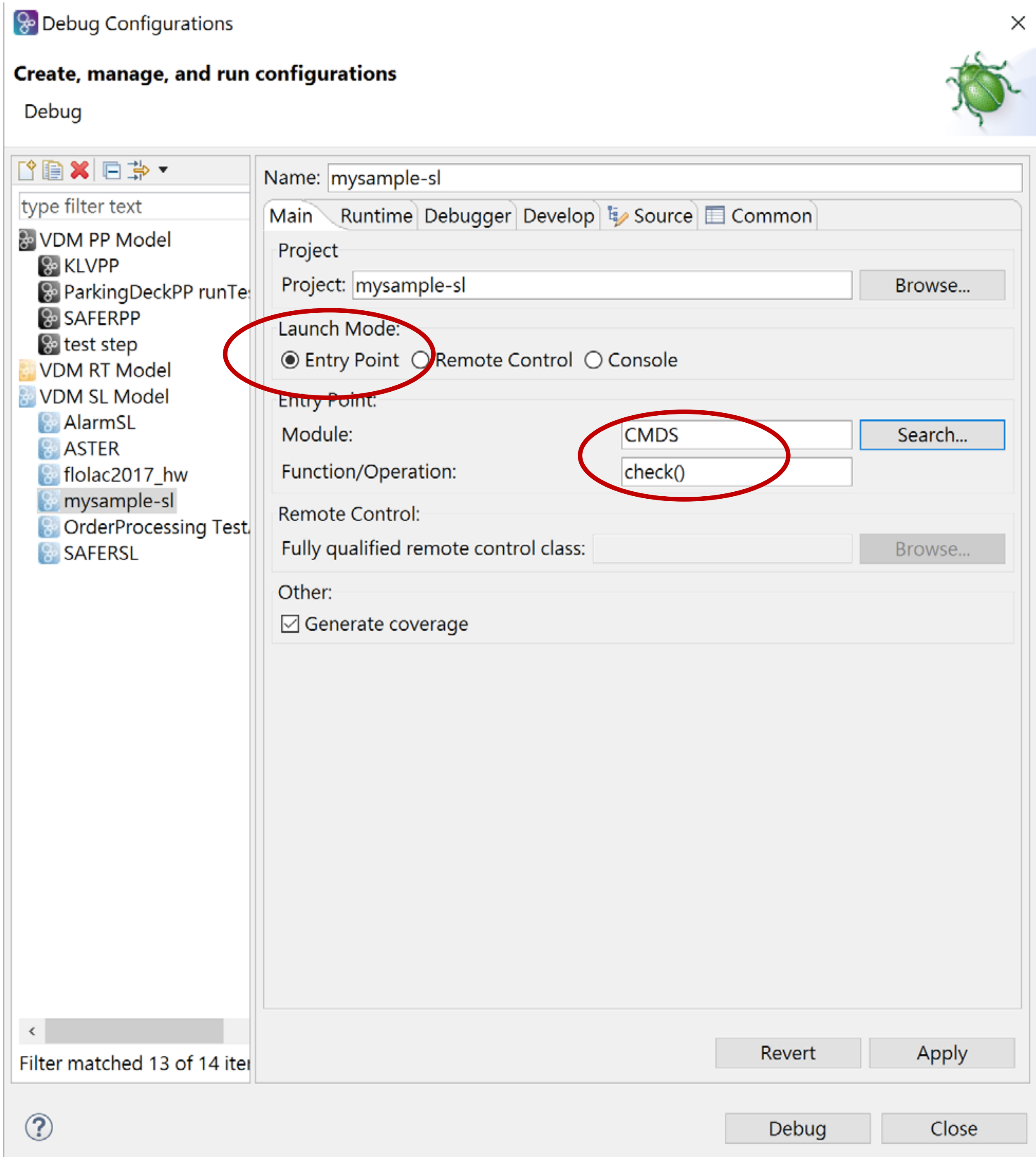
Launch in console

Run some tests

Initialize and set dynamic type check off

'p' means "print"

```
> init
> set dtc off
> p inv_S(mk_S([]))
true
> p inv_S(mk_S([<R>,<R>,nil,<R>]))
false
> p inv_S(mk_S([<R>,<L>,nil,nil]))
false
> p pre_push_cmd(<R>,mk_S([]))
true
> p pre_push_cmd(<R>,mk_S([<R>]))
false
> p pre_push_cmd(<R>,mk_S([<L>]))
true
> p post_push_cmd(<R>,mk_S([]),mk_S([<R>]))
true
> p post_push_cmd(<R>,mk_S([<L>,nil,<L>]),mk_S([<L>,nil,<L>,<R>]))
false
> p post_push_cmd(<R>,mk_S([<L>,nil,<L>]),mk_S([<R>,<L>,nil,<L>]))
true
> p times_count([<R>,<L>,nil,<R>,<L>,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5, <L> |-> 2}
true
> p times_count([<R>,nil,<R>,nil,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5}
false
> p times_count([<R>,nil,<R>,nil,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5, <L> |-> 0}
true
```



Debug config.

Launch from
an entry point
(function or operation)

Test Coverage

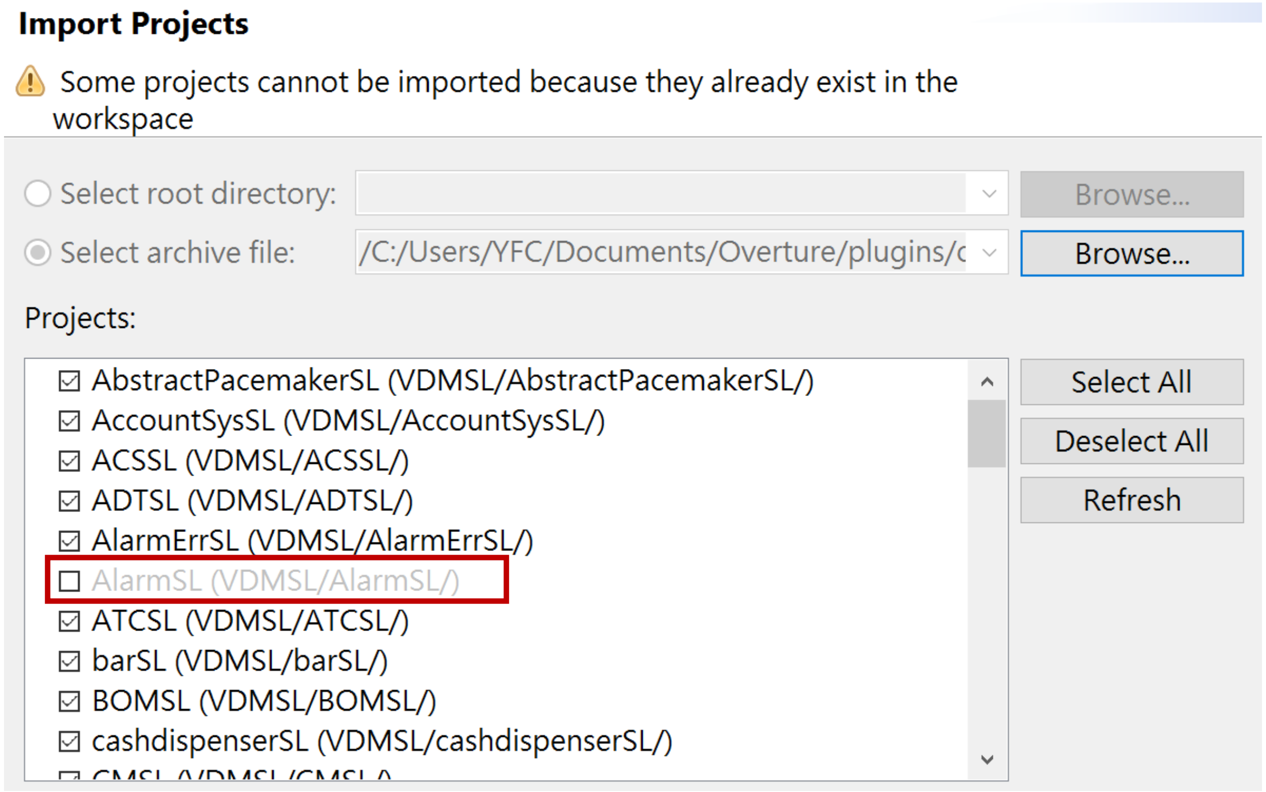
```
> coverage
Test coverage for mysample.vdmsl:

  module CMDS
  definitions

  types
  CMD = <R> | <L>;
  CMD_series = seq of [CMD];
  CMD_times = map CMD to nat;

  state S of
    commands : CMD_series
+   inv s == forall k in set {1,...,len s.commands - 1} & s.commands(k) <> s.commanc
+   init p == p = mk_S([])
  end

  operations
-   push_cmd(a:[CMD])
+   pre commands = [] or hd commands <> a
+   post hd commands = a and tl commands = commands~;

  functions
  times_count : CMD_series -> CMD_times
+ times_count(a) == { <R> |-> len [ i | i in set inds a & a(i)=<R> ], <L> |-> len [

end CMDS

Coverage = 90.0%
```

Test Coverage

```
CMD_times = map CMD to nat;

state S of
  commands : CMD_series
  inv s == forall k in set {1,...,len s.commands - 1} & s.commands(k) <> s.commands(k+1)
  init p == p = mk_S([])
end

operations
  push_cmd(a:[CMD])
  pre commands = [] or hd commands <> a
  post hd commands = a and tl commands = commands~;

functions
times_count : CMD_series -> CMD_times
times_count(a) == { <R> |-> len [ i | i in set inds a & a(i)=<R> ], <L> |-> len [ i | i in set inds a & a(i)=<L> ] };

check : () -> seq of bool
check() == [
  inv_S(mk_S([])), -- true
  inv_S(mk_S([<R>,<R>,nil,<R>])), -- false
  inv_S(mk_S([<R>,<L>,nil,nil])), -- false
  pre_push_cmd(<R>,mk_S([])), -- true
  pre_push_cmd(<R>,mk_S([<R>])), -- false
  pre_push_cmd(<R>,mk_S([<L>])), -- true
  post_push_cmd(<R>,mk_S([]),mk_S([<R>])), -- true
  post_push_cmd(<R>,mk_S([<L>,nil,<L>]),mk_S([<L>,nil,<L>,<R>])), -- false
  post_push_cmd(<R>,mk_S([<L>,nil,<L>]),mk_S([<R>,<L>,nil,<L>])), -- true
  times_count([<R>,<L>,nil,<R>,<L>,nil,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5, <L> |-> 2}, -- true
  times_count([<R>,nil,<R>,nil,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5}, -- false
  times_count([<R>,nil,<R>,nil,<R>,nil,<R>,nil,<R>]) = {<R> |-> 5, <L> |-> 0} -- true
];

end CMDS
```

Try Yourself

- Import a module from Overture examples and run
  - For example, the chemical plant alarm.



Try Yourself

- VDM Quick interpreter
  - Quickly check a VDM expression
  - For example, to know the resulted set from a set comprehension expression

```
> { a | a in set {0, … , 10} & a mod 2 = 0}
{0,2,4,6,8,10}
```

What if ?

```
> { a : nat & a mod 2 = 0}
```

Type binding (unexecutable) vs. Set binding (unexecutable)