INTO-CPS

VDM Tutorial

John Fitzgerald, Newcastle University

NII 大学共同利用機関法人 情報・システム研究機構 国立情報学研究所 National Institute of Informatics
AARHUS UNIVERSITY
Newcastle University
Horizon 2020 Programme

VDM Background

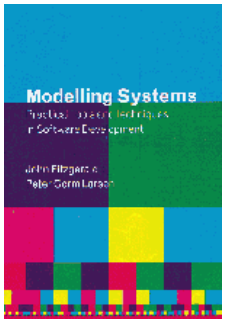
- Our goal: well-founded but accessible modelling & analysis technology
- VDMTools → Overture → Crescendo → Symphony
  - Pragmatic development methodologies
  - Industry applications
- VDM: Model-oriented specification language
  - Extended with objects and real time.
  - Basic tools for static analysis
  - Strong simulation support
  - Model-based test

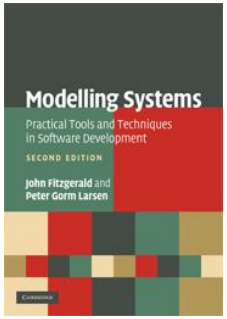
Horizon 2020 Programme

VDM (Vienna Development Method)

- A formal method for specification of software
- Three flavours
  - VDM-SL (Specification Language)
  - VDM++ adds object-orientation
  - VDM-RT adds real-time features (clock and deployment)
- Model-oriented specification language
  - Simple, abstract data types
  - Invariants to restrict membership
  - Functional specification:
    - Implicit specification (pre/post)
    - Explicit specification (functional or imperative)

VDM-SL Module Outline



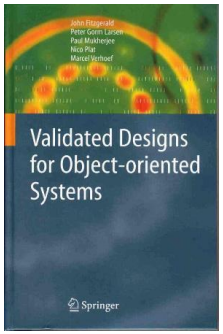



```
module <module-name>
  imports
  exports
    ...

definitions
  state
  types
  values
  functions
  operations
    ...

end <module-name>
```

Interface

Definitions

VDM++ Class Outline




```
class <class-name>
  instance variables
    ...
  types
  values
  functions
  operations
  thread
    ...
  sync
    ...
  traces
    ...
end <class-name>
```

- instance variables, ... — Internal object state
- types, values, functions, operations — Definitions
- thread, ... — Dynamic behaviour
- sync, ... — Synchronization control
- traces, ... — Test automation support

INTO-CPS

Data Types

- Basic types
  - Boolean
  - Numeric
  - Tokens
  - Quote types
  - Characters / String
- Compound types
  - Set types
  - Sequence types
  - Map types
  - Product types
  - Record types
  - Union types
  - Optional types

Type	Values
nat1	1, 2, 3, ...
nat	0, 1, 2, ...
int	..., -2, -1, 0, 1, ...
real	-12.78356, ..., 0, ..., 3, ..., 1726.34, ...

```
flag: bool := false;
flag := true
```

```
id: token := mk_token(5);
id := mk_token("ken")
```

```
<RED>
x = y (equality) and x <> y (inequality)
```

```
letter: char := 'k';
name: seq of char := "ken"
```

```
s: set of int := {1,5,8,3};
1 in set s = true
```

```
s: seq of int := [1,5,5,8,3];
s(4) = 8
```

```
m: map int to real := {1 |-> 3.14};
m(1) = 3.14
```

```
Pair = nat * real;
x: Pair := mk_(1, 3.14);
x.#1 = 1
```

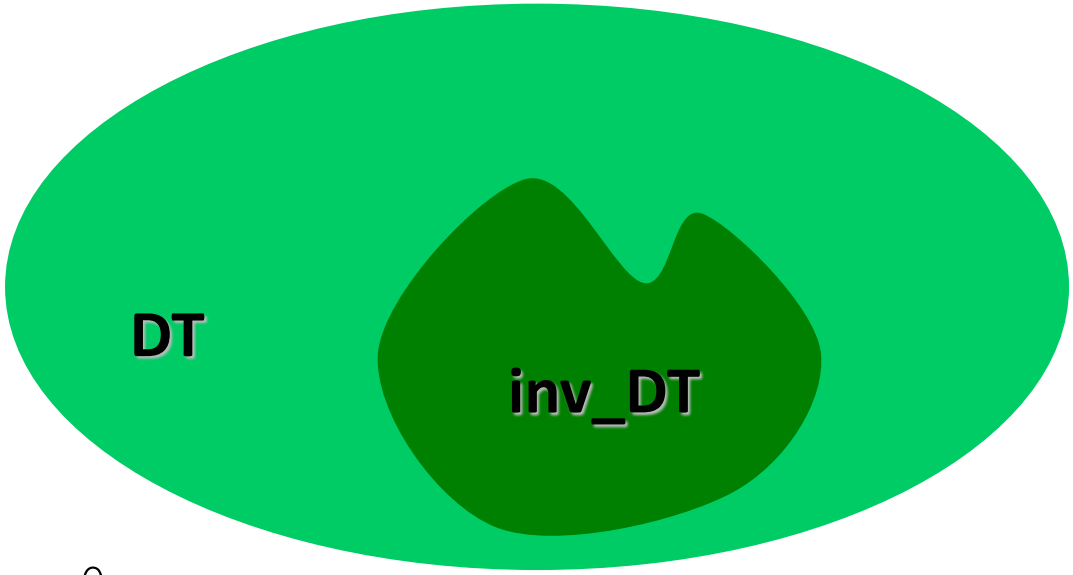
```
Pair :: a : nat
        b : real;
mk_Pair(1, 3.14)
```

```
Colour = <RED> | <GREEN> | <BLUE>;
x: Colour := <RED>
```

```
Type = [nat]   equivalent to
Type = nat | nil
```

Horizon 2020 Programme

Type Invariants




```
Even = nat
inv n == n mod 2 = 0

SpecialPair = nat * real - the first is smallest
inv mk_(n,r) == n < r

DisjointSets = set of set of  A
inv ss == forall s1, s2 in set ss &
                s1 <> s2 => s1 inter s2 = {}
```

Set Types

- Unordered collections of elements
- One copy of each element
- The elements themselves can any type
- e.g.
  - `set of int`
  - `{1,5,8,3};`
  - `{}`

Overview of Set Operators

`e in set s1`	Membership (∈)	`A * set of A -> bool`
`e not in set s1`	Not membership (∉)	`A * set of A -> bool`
`s1 union s2`	Union (∪)	`set of A * set of A -> set of A`
`s1 inter s2`	Intersection (∩)	`set of A * set of A -> set of A`
`s1 \ s2`	Difference (\)	`set of A * set of A -> set of A`
`s1 subset s2`	Subset (⊆)	`set of A * set of A -> bool`
`s1 psubset s2`	Proper subset (⊂)	`set of A * set of A -> bool`
`s1 = s2`	Equality (=)	`set of A * set of A -> bool`
`s1 <> s2`	Inequality (≠)	`set of A * set of A -> bool`
`card s1`	Cardinality	`set of A -> nat`
`dunion s1`	Distr. Union (∪)	`set of set of A -> set of A`
`dinter s1`	Distr. Intersection (∩)	`set1 of set of A -> set1 of A`
`power s1`	Finite power set (ℙ)	`set of A -> set of set of A`

Sequence Types

- Could also be called lists
  - Not fixed length like Java arrays
- Ordered collections of elements
- Numbered from 1 (not 0 like Java)
  - Access element with () and not [], e.g. `list(1)`
- Multiple copies of each element allowed
- The elements themselves can be any type
- e.g.
  - **`seq of int`**; **`seq1 of int`** (non-empty)
  - `[1,5,5,8,1,3]`; `[]`

Overview of Sequence Operators

`hd l`	Head	`seq1 of A -> A`
`tl l`	Tail	`seq1 of A -> seq of A`
`len l`	Length	`seq of A -> nat`
`elems l`	Elements	`seq of A -> set of A`
`inds l`	Indexes	`seq of A -> set of nat1`
`l1 ^ l2`	Concatenation	`seq of A * seq of A -> seq of A`
`conc ll`	Distr. conc.	`seq of seq of A -> seq of A`
`l(i)`	Seq. application	`seq1 of A * nat1 -> A`
`l ++ m`	Seq. modification	`seq1 of A * map nat1 to A -> seq1 of A`
`l1 = l2`	Equality	`seq of A * seq of A -> bool`
`l1 <> l2`	Inequality	`seq of A * seq of A -> bool`

Mapping Types

- Unordered collections of pairs of elements (maplets) with a unique relationship
  - mapping keys to values
  - like Python dictionary
- The elements themselves can be any type
- e.g.
  - `map int to real`

Overview of Mapping Operators

`dom m`	Domain	`(map A to B) -> set of A`
`rng m`	Range	`(map A to B) -> set of B`
`m1 munion m2`	Merge	`(map A to B) * (map A to B) -> (map A to B)`
`m1 ++ m2`	Override	`(map A to B) * (map A to B) -> (map A to B)`
`merge ms`	Distr. merge	`set of (map A to B) -> map A to B`
`s <: m`	Dom. restr. to	`set of A * (map A to B) -> map A to B`
`s <-: m`	Dom. restr. by	`set of A * (map A to B) -> map A to B`
`m :> s`	Rng. restr. to	`(map A to B) * set of A -> map A to B`
`m :-> s`	Rng. restr. by	`(map A to B) * set of A -> map A to B`
`m(d)`	Map apply	`(map A to B) * A -> B`
`inverse m`	Map inverse	`inmap A to B -> inmap B to A`
`m1 = m2`	Equality	`(map A to B) * (map A to B) -> bool`
`m1 <> m2`	Inequality	`(map A to B) * (map A to B) -> bool`

Specifying Behaviour

- Specifications in terms of post-conditions define a contract

```
sqrt(x: nat) r: real
post x = r * r
```

Implicit definition, not executable

- Explicit version

```
sqrt: nat -> real
sqrt(x) == Math`sqrt(x)
```

Explicit definition can be executed

- Pre-condition and post-conditions

```
sqrt: int -> real
sqrt(x) == Math`sqrt(x)
pre x > 0
post x = RESULT * RESULT
```

A Simple Controller Class

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

COE synchronises these to other models

- Divided into sections (e.g. instance variables, operations, etc.)
- Inheritance supported
  - `class Controller is subclass of Parent`
- Objects created with
  - `new Controller`
- Constructors also similar to Java
  - `public Controller: real * real ==> Controller`
    `Controller(a,b) == (`
    `x:= a;`
    `y := b`
    `);`
- Sections can be repeated and mixed
- Comments are
  - Two dashes: `-- comment`
  - Or: `/* block comment */`

Instance Variables

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- Give the state of the object
- Note syntax for giving the type
  - `private double measured;`
  - `private measured: real;`
- Visibility similar to Java (added here for illustration only)
  - Defaults is private is no visibility given
- Can be assigned when defined

Functions

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- Are pure
  - No side effects
  - Cannot access instance variables
- No return keyword, defined with expressions that return the correct type
- Useful for auxiliary / helper calculations
- Note signature above definition
  - `real * int * bool -> real`
- No loops, must use functional programming techniques
  - Can call other functions

Operations

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- Similar to functions, but...
  - Can access instance variables / have side effects
  - Are imperative like Java
  - Can use while, for loops etc.
  - Must use **`return`** keyword when returning a value
- Can call other operations and functions
- Can define local variables but only at the start
  - ```
    Step() == (
        dcl x: real := 0;
    ```
- Note parentheses () not {}
- Note different arrow to function
  - ```
    real * int * bool ==> real
    ```

Values

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- Used to define constants
- Note = is used, not :=
- Do not need a type
  - but can have one

    ```
    Kp: real = 1.24;
    ```
- Can be set as *Shared Design Parameters*
- Are static, can be accessed from other classes (if public)
  - `Controller`Kp`

Threads

```
class Controller

instance variables

private measured: RealPort;
public setpoint: real;
protected err: real;
output: RealPort;

operations

public Step: () ==> ()
Step() == (
  m := measured.getValue();
  err := setpoint - m;
  output.setValue(P(err));
);

functions

private P: real -> real
P(err) == err * Kp

values

Kp = 2.0

thread

periodic(2E7, 0 , 0 , 0)(Step);

end Controller
```

- Threads are defined in the class
- Definition could be operation call; will run once
  - `thread`
    `Step();`
- Or a loop
  - `thread`
    `while true do Step();`
- Starting
  - `ctrl: Controller := new Controller();`
    `start(ctrl)`
- Or a special, periodic definition (as on the left)
  - will call `Step` operation once every 2e7 nanoseconds (20 milliseconds; 0.02 seconds; 50Hz)

The HardwareInterface Class


```
class HardwareInterface

values
-- @ interface: type = parameter, name="minlevel";
public minlevel : RealPort = new RealPort(1.0);
-- @ interface: type = parameter, name="maxlevel";
public maxlevel : RealPort = new RealPort(2.0);

instance variables
-- @ interface: type = input, name="level";
public level : RealPort := new RealPort(0.0);

instance variables
-- @ interface: type = output, name="valveState";
public valveState : BoolPort := new BoolPort(false);

end HardwareInterface
```

The System Class

```
system MySystem

instance variables

-- controller
public static ctrl: Controller;

-- Hardware interface variable required
-- by FMU Import/Export
public static hwi: HardwareInterface :=
       new HardwareInterface();
-- CPU
private cpu: CPU; := new CPU(<FP>, 1E6)

operations

public MySystem: () ==> MySystem
MySystem() == (
    ctrl := new Controller();
    cpu.deploy(ctrl)
)

end MySystem
```

- Special class for CPU and deployment
- Can only define instance variables and a constructor
- CPU speed in (simulated) MIPS
  - getting a model within ~20% of the real thing is typically “good enough”

The World Class

```
class World

operations

-- run a simulation
public run: () ==> ()
run() == (
    start(System`ctrl);
    block();
);

-- wait for simulation to finish
block: () ==> ()
block() == skip;
sync per block => false;

end World
```

- Entry point for code execution
- Start threads and wait for end of simulation

Concurrency

- Concurrency in VDM-RT is based on threads
- Threads communicate using shared objects
- Synchronization on shared objects is specified using permission predicates
  - `sync`
    `per <operation name> => predicate`
  - Operation is blocked when the predicate is false
  - `mutex(A, B)`
  - History counters

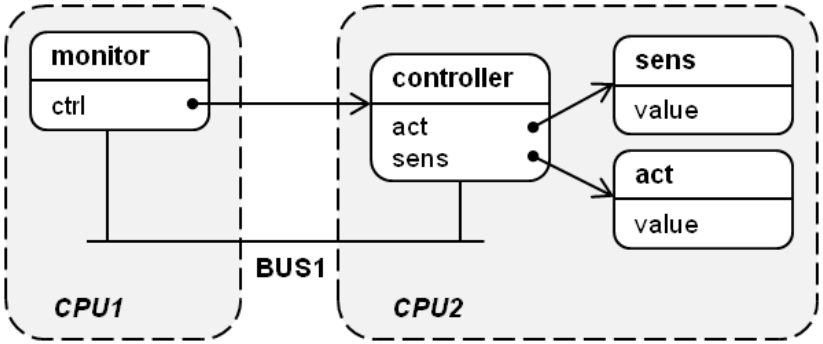
`#req` `op`	The number of times that `op` has been requested
`#act` `op`	The number of times that `op` has been activated
`#fin` `op`	The number of times that `op` has been completed
`#active` `op`	The number of active executions of `op`
`#waiting` `op`	The number of waiting executions of `op`

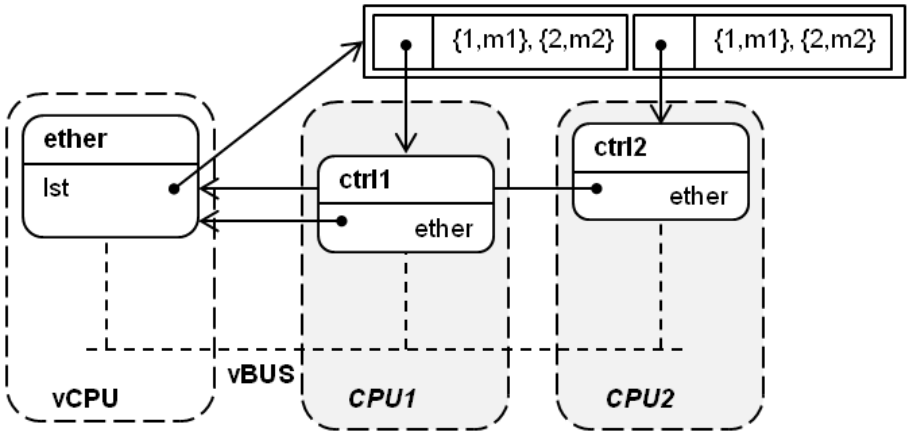
VDM-RT Features (1)

- VDM-RT has extensions for modelling real-time systems
- An internal clock
  - in nanoseconds from simulation start
  - accessible with the **`time`** keyword, e.g.
    - `dcl now: real := time/1e9 -- time in seconds`
- **All** expressions advance the clock
  - default is two simulated cycles
  - Can be altered with `cycles(number)(expression)` or `duration(number)(expression)`
  - Cycles used to compute duration

VDM-RT Features (2)

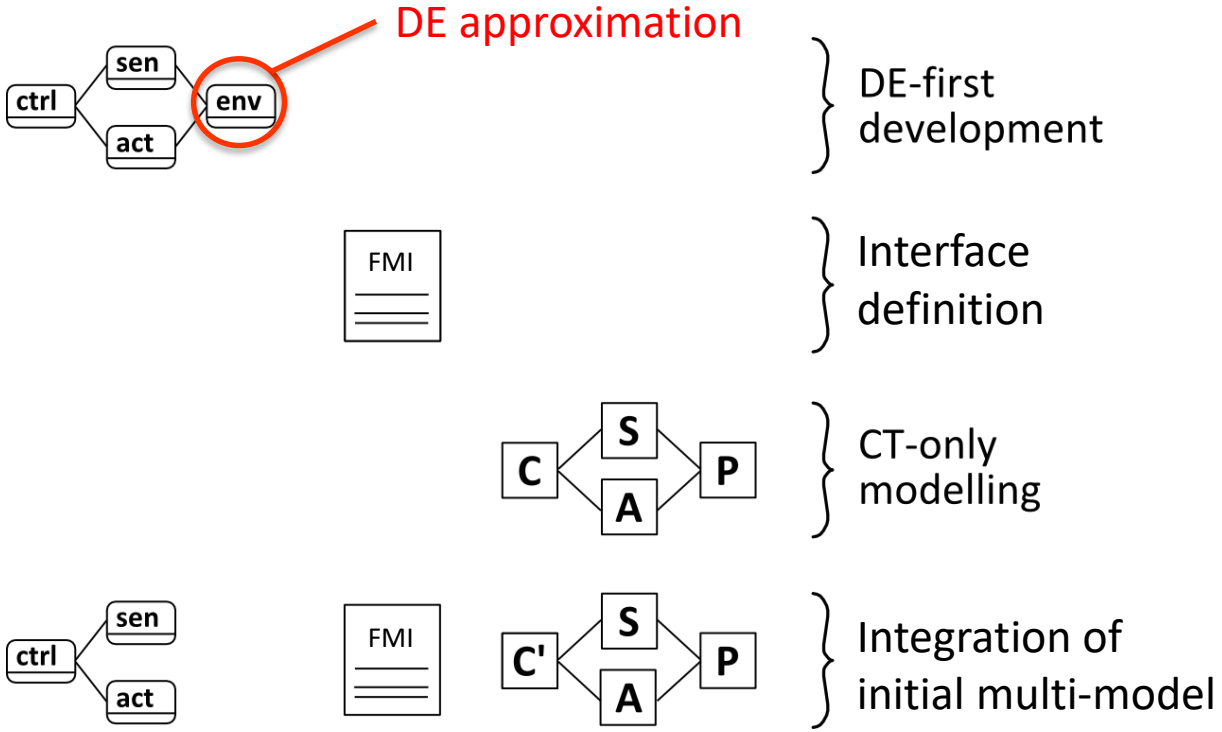
- Also models of CPUs and BUSes to try to model real code execution
  - objects are “deployed” to CPU with a given speed
  - execution duration depends on the modelled CPU speed
  - also a virtual CPU that doesn’t advance the clock





DE-first Modelling

- DE-first (DE-only) model:
  - Controller, sensor and actuator classes
  - *Environment model*

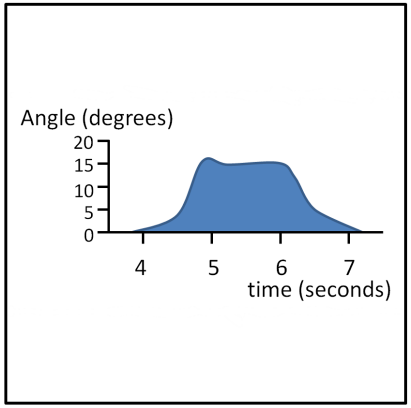


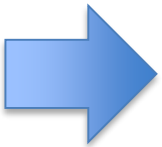
Approximating CT Behaviour

- Linear approximations
- Simple integration:

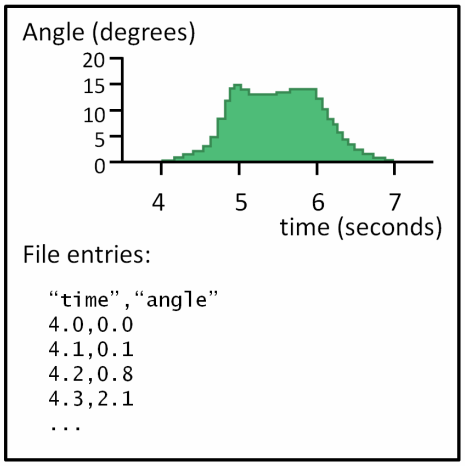
```
position = position + velocity * dt;
velocity = velocity + acceleration * dt;
```

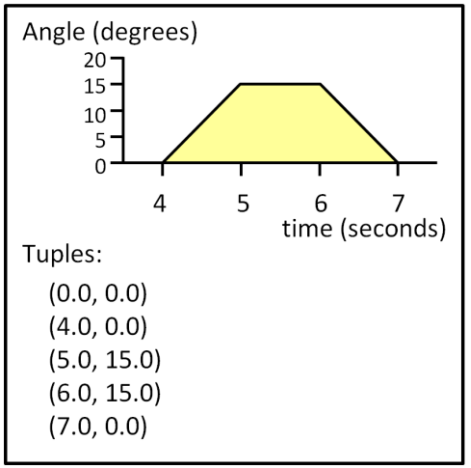
- Approximation of non-linear behaviour





Data input

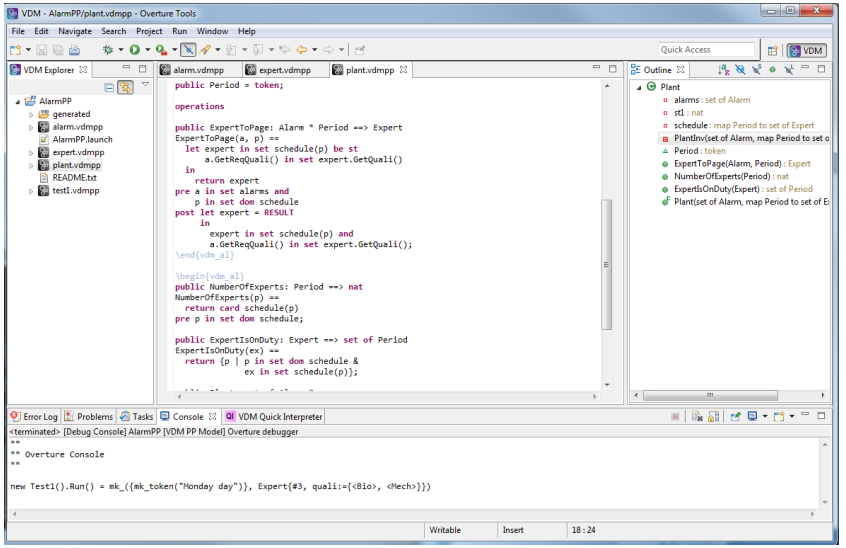




Tuples

The Overture Tool

- Open-source tool for analysing VDM models
- Eclipse/Java based
- Current stable version 2.5.4 (November 2017)
- Visit us at http://overturetool.org/
  - Useful references
  - Examples can be imported
  - Language manual
  - Tool users manual
  - Install the FMI Exporter
  - Install the C code generator

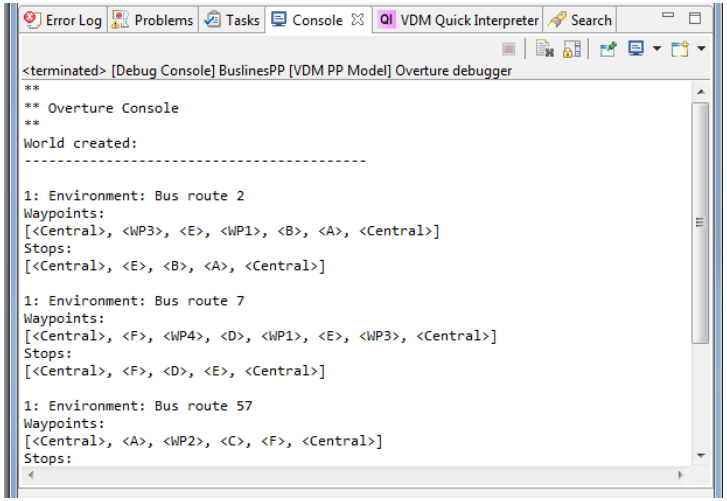


Overview of the Tool Features (1)

- Simulation of VDM models
  - Debugging
  - Combining VDM with executable code
- Model validation
  - Static analysis (e.g. type-checking)
  - Unit testing
  - Adding visualisation to a VDM model
  - System-level timing constraints (VDM-RT trace viewer)
- Realising a VDM model
  - Generate Java, C from subsets of VDM dialects

Overview of the Tool Features (2)

- VDM standard libraries
  - IO: For file and console input/output
  - CSV: For working with CSV-based data
  - MATH: Provides commonly used math functions
  - VDMUnit: A unit testing framework for VDM
  - VDMUtil: For converting between VDM values

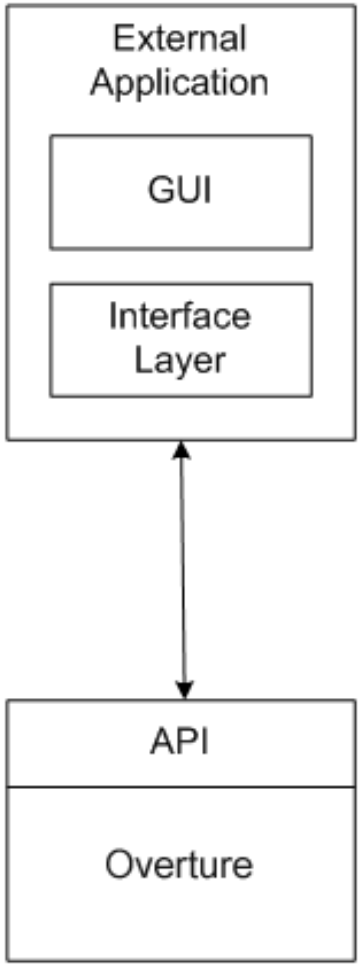


Combining VDM with Executable Code

INTO-CPS

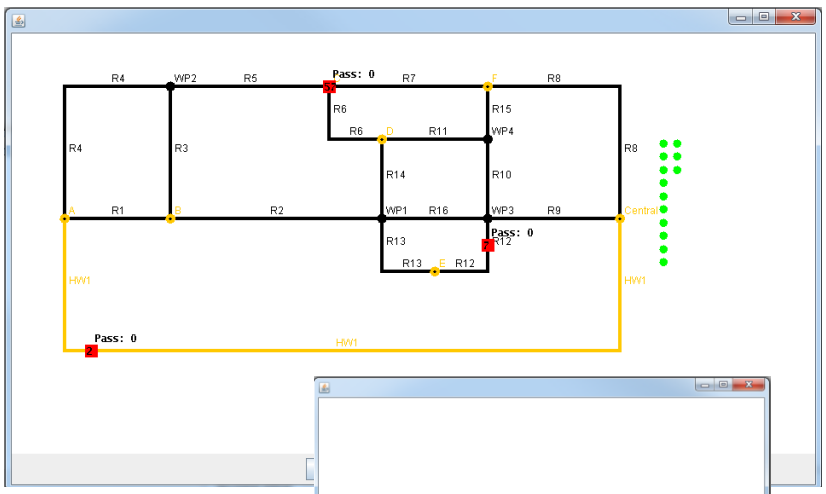
- Two types of external interfacing
  - External Call Interface
    - From the VDM model to an external interface
  - Remote Control Interface
    - Allows for external calls into a VDM Model
- Used to implement the VDM libraries

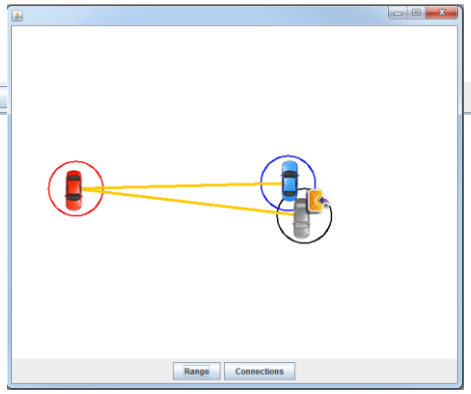
```
public static sqrt: real -> real
sqrt(a) ==
is not yet specified
pre a >= 0;
```

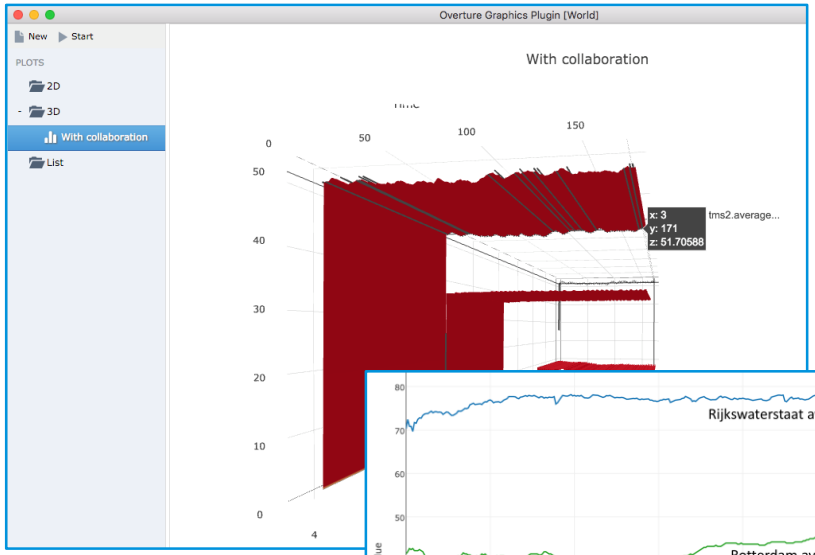


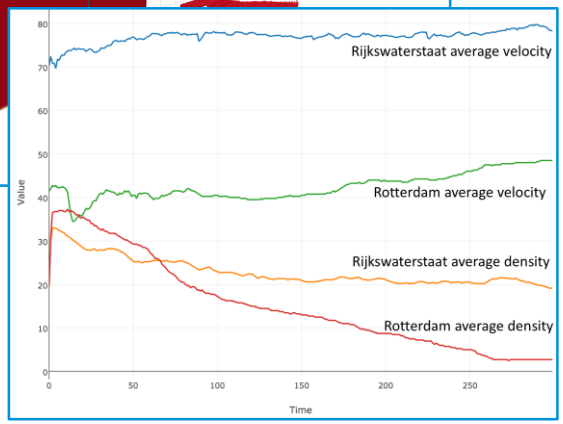
Visualising a VDM model

- Understanding a formal model can itself be difficult
  - Especially for a non-technical stakeholder
- Validating a model with a domain expert



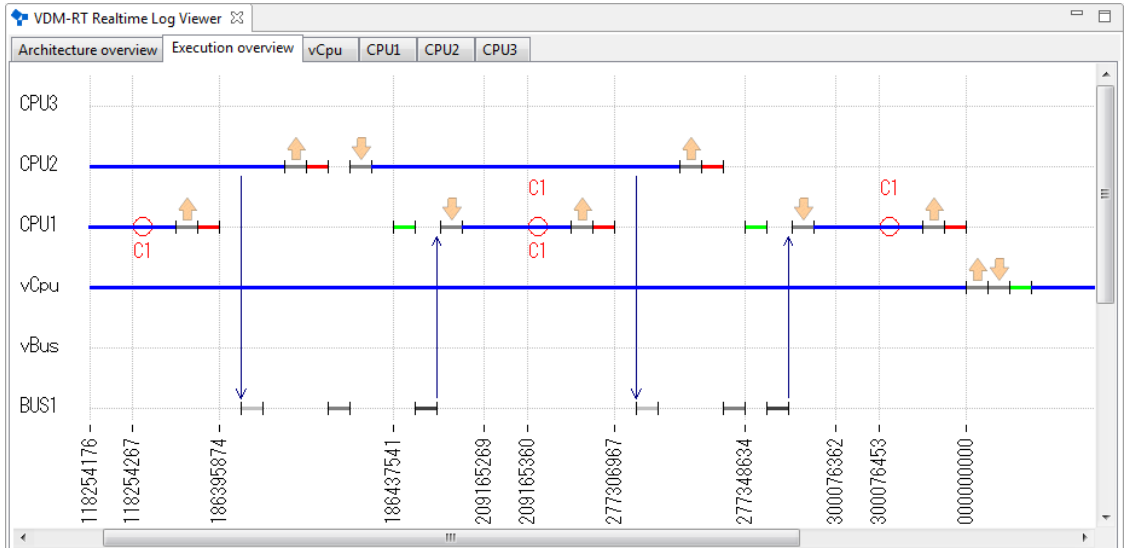






VDM-RT trace viewer

- VDM-RT models a distributed architecture
  - CPUs are connected via buses
  - Objects are deployed on CPUs
- A trace records the VDM-RT model execution
  - Message exchange between objects
- Validation of system-level timing constraints



Functional Mockup Units (FMUs)

- Import model description to make skeleton model
  - Creates system class
  - Creates World class
  - Creates HardwareInterface class
- Export to FMU
  - Tool wrapper
  - Source code (in C)