Part 3:
VDM-RT for Co-simulation

John Fitzgerald
Peter Gorm Larsen
Ken Pierce

Background: VDM

- 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

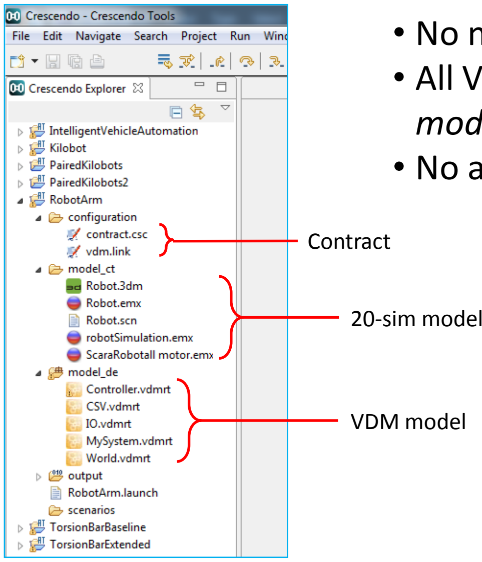
Overview

- VDM use in Crescendo
- VDM-RT (Real-Time)
  - Classes, instance variables, functions, operations, values (constants), threads, synchronisation
  - Real-time features
- Types in VDM
  - Comparison with Java
  - Collections, operators, union types, invariants
- Concurrent in VDM-RT
  - Threads
  - Synchronisation
- DE-first modelling in Crescendo
  - Modelling approximations

Vienna Development Method (VDM)

- Is a formal method for specification of software
  - but (a subset) can be executed for simulation
- Three flavours
  - VDM-SL (Specification Language); created at IBM labs Vienna in the 1970s
  - VDM++ adds object-orientation
  - **VDM-RT adds internal clock and deployment (used in Crescendo)**
- Model-oriented specification
  - Simple, abstract data types
  - Invariants to restrict membership
  - Functional specification:
    - Implicit specification (pre/post)
    - Explicit specification (functional or **imperative**)

Crescendo Workspace



- No namespace or packages
- All VDM-RT classes under *model_de* checked
- No auto-completion (sorry!)

Debugging

- `IO`print(“a string”)`
- `IO`println(“a string plus newline”)`
- `IO`printf(“%s: value of x is %s”, [1, x])`
  - Only `%s` is supported currently!
- String concatenation is ^ (usually Shift-6)
- The symbol: ` is next to the 1 key (top left)
  - Used to access static members of classes (not `.` as in Java)
- Setting breakpoints / Debug perspective

A Simple Controller Class

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

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

values

Kp = 2.0

thread

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

end Controller
```

Co-simulation engine can sync these to 20-sim model

- 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: real := 0;
public setpoint: real := 0;
protected err: real := 0;
output: real := 0;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := 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
- More on types (`real` etc.) later
```

Functions

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := 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

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := P(err);
);

functions

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

values

Kp = 2.0

thread

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

end Controller
```

Like **void**

- 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

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := 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;`
- Are static, can be accessed from other classes (if public)
  - `Controller`Kp`

Threads

```
class Controller

instance variables

measured: real;
setpoint: real;
err: real;
output: real;

operations

public Step: () ==> ()
Step() == (
  err := setpoint - measured;
  output := 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)

VDM-RT Important Features (1)

- VDM-RT (Real Time) 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)`

VDM-RT Important Features (2)

- The internal clock is synchronised with 20-sim (see semantics on earlier lecture notes)
- Also models of CPUs and buses to try to model real code execution
  - objects are "deployed" to CPU with a given speed
  - the time take for execution depends on the modelled CPU speed
  - also a virtual CPU that doesn't advance the clock (if objects aren't deployed)

System Class

```
system MySystem

instance variables

-- controller
public static ctrl: Controller;

-- 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"

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
- Here `run()` is like `main()`
- Start threads and wait for end of simulation

Statements

- Loops
  - `while true do (…)`
  - `for i = 1 to 10 do (…)`
  - `for x in xs do (…)` (for sequences)
  - `for all x in set xs do (…)` (for sets)
- Conditionals
  - ```
    if x then …
    elseif y then …
    else …
    ```
  - cases not switch
    ```
    cases x of
        1 -> …,
        2 -> …,
        others -> …
    end
    ```
- Reminder: blocks use ( ) and not { }
- Semi-colons separate; they do not terminate

Primitives Types in Java

- Natural numbers
  - **`byte`** (-128, 127);
  - **`short`** (-32768, 32,767)
  - **`int`** (-2^{31}, 2^{31}-1)
  - **`long`** (-2^{63}, 2^{63}-1)
- Real numbers
  - **`float`** (32-bit IEEE 754 floating point)
  - **`double`** (64-bit IEEE 754 floating point)
- **`boolean`**
- **`char`**
    ```

More Java Types

- Compound types
  - arrays (e.g. `int[]`)
- Everything else is a class…
  - String
  - *List* (e.g. ArrayList)
  - *Set* (e.g. HashSet)
  - *Map* (e.g. HashMap)
  - etc.

Types in VDM – many more!

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

Boolean Types

Type	Values
`bool`	`true`, `false`

Operator	Name	Type
`not b`	Negation	`bool` → `bool`
`a and b`	Conjunction	`bool` * `bool` → `bool`
`a or b`	Disjunction	`bool` * `bool` → `bool`
`a => b`	Implication	`bool` * `bool` → `bool`
`a <=> b`	Biimplication	`bool` * `bool` → `bool`
`a = b`	Equality	`bool` * `bool` → `bool`
`a <> b`	Inequality	`bool` * `bool` → `bool`

Note assignment is:
`:=`

Numeric Types (1)

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

- Fewer than Java
  - e.g. no float vs double
- Restrictions can be added with invariants
  - see later

Numeric Types (2)

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**

Custom Types

- You can define your own types
  - To capture the properties of data you need
  - To clearly specify what is required
  - We can restrict types with invariants…

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

- In Java we would build a class

More Types (1)

- Token types
  - **token**
  - Can only be compared
    - x = y (equality)
    - x <> y (inequality)
  - e.g. **mk_token**(5),**mk_token**("ken")
- Quote types
  - Represent enumerated types
  - <RED>, <BLUE>, <GREEN>
  - Again can only be compared for equality

More Types (2)

- Characters
  - **char**
  - Strings are defined with **seq of char**
  - The tool allows for string literals, e.g. "ken" is equivalent to ['k','e','n']
- Union types
  - Like "or" for types; can be used with quote types
  - Type = **nat** | **bool**
  - For enumeration
    - Colour = <RED> | <GREEN> | <BLUE>
- Optional types
  - Type = [nat] equivalent to Type = nat | **nil**

Compound Types: Product / Record

- Product types, i.e. tuples
  - like a fixed length list, with at least two element
  - e.g. `SpecialPair = nat * real`
  - accessing elements

```
x: SpecialPair := mk_(1, 3.14);
x.#1 -- access first element
```

- Record types
  - Tuple with named elements
  - Like `struct` in C
  - e.g.

```
RecordPair :: a : nat
              b : real
```

  - accessing elements

```
x: RecordPair := mk_RecordPair(1, 3.14);
x.a -- access first element
```


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};`
  - `{}`

Set Operators

Operator	Name	Type
`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 ss`	Distributed union	set of set of A → set of A
`dinter ss`	Distributed intersection	set of set of A → set of A
`power s1`	Finite power set	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]`; `[]`

Sequence Operators

Operator	Name	Type
`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**
`reverse l`	Reverse	**seq of** A → **seq of** A
`l1 ^ l2`	Concatenation	(**seq of** A) * (**seq of** A) → **seq of** A
`conc ll`	Distributed concatenation	**seq of seq of** A → **seq of** A
`l ++ m`	Sequence modification	**seq of** A * **map nat1 to** A → **seq of** A
`l(i)`	Sequence application	**seq of** A * **nat1** → A
`l1 = l2`	Equality	(**seq of** A) * (**seq of** A) → **bool**
`l1 <> l2`	Inequality	(**seq of** A) * (**seq of** A) → **bool**

Map 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`**
  - `{1 |-> 3.14, 2 |-> 6.28}`

Map Operators

Operator	Name	Type
`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`	Distributed merge	set of (map A to B) → map A to B
`s <: m`	Domain restrict to	(set of A) * (map A to B) → map A to B
`s <-: m`	Domain restrict by	(set of A) * (map A to B) → map A to B
`m :> s`	Range restrict to	(map A to B) * (set of B) → map A to B
`m :-> s`	Range restrict by	(map A to B) * (set of B) → map A to B
`m(d)`	Map apply	(map A to B) * A → B
`m1 comp m2`	Map composition	(map B to C) * (map A to B) → map A to C
`m ** n`	Map iteration	(map A to A) * nat → map A 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
`inverse m`	Map inverse	inmap A to B → inmap B to A

Concurrency in VDM-RT

- Concurrency in VDM-RT is based on threads
- Threads communicate using shared objects
- Synchronization on shared objects is specified using permission predicates
- Class may have a thread section:

```
class SimpleThread
thread
  while true do skip;
end SimpleThread
```

- Thread execution begins by using the **`start`** statement
  - on an object whose class defines a thread

Producer / Consumer Example

- Concurrent threads must be synchronized to avoid race conditions
- Illustrated here with a simple producer-consumer example
- Assume a single producer, single consumer
- Producer has a thread which repeatedly places data in a buffer
- Consumer has a thread which repeatedly fetches data from the buffer
- For simplicity, single-item buffer
  - optional value: `nil` means empty

Producer / Consumer Classes

```
class Producer

instance variables

b: Buffer

operations

Produce: () ==> nat
Produce() == ...

thread
  while true do
    b.Put(Produce())

end Producer
```

```
class Consumer

instance variables

b: Buffer

operations

Consume: nat ==> ()
Consume(d) == ...

thread
  while true do
    Consume(b.Get())

end Consumer
```

```
class Buffer

instance variables

data : [nat] := nil

operations

public Put: nat ==> ()
Put(newData) ==
  data := newData;

public Get: () ==> nat
Get() ==
  let oldData = data in (
    data := nil;
    return oldData
  )

end Buffer
```

Permission Predicates (1)

- What if the producer thread generates values faster than the consumer thread can consume them?
- Shared objects require *synchronisation*
- Synchronisation is achieved in VDM++ using *permission predicates*
- A permission predicate describes when an operation call may be executed
- If a permission predicate is not satisfied, the operation call blocks

Permission Predicates (2)

- Permission predicates are described in the **sync** section of a class:

```
sync
per <operation name> => predicate
```

- Operation is blocked when the predicate is false
- The predicate may refer to the class's instance variables
- The predicate may also refer to special variables known as *history counters*

History Counters

- Allow permission predicates to refer to current and historical information about operations

Counter	Description
`#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

Synchronised Buffer

- Assuming the buffer does not lose data, there are two requirements:
  - It should only be possible to *get* data, when the producer has placed data in the buffer.
  - It should only be possible to *put* data when the consumer has fetched data from the buffer.
- The following permission predicates could model these requirements:

```
per Put => data = nil
per Get => data <> nil
```

- Can also be written using history counters:

```
per Put => #fin(Put) - #fin(Get) = 0
per Get => #fin(Put) - #fin(Get) = 1
```

Mutual Exclusion

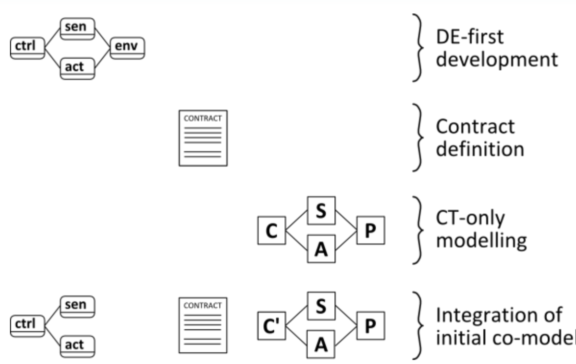
- Another problem could arise with the buffer: what if the producer produces and the consumer consumes at the same time?
- The result could be non-deterministic and/or counter-intuitive.
- VDM++ provides the keyword **`mutex`**

```
mutex(Put, Get)
```

- Shorthand for

```
per Put => #active(Get) = 0
per Get => #active(Put) = 0
```


DE-first Modelling (1)



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

DE-first Modelling (2)

- Development begins with a system model in the DE formalism
- This model contains a controller object (ctrl) and environment object (env)
- Linked by (one or more) sensor and actuator objects (sens and act).
- The environment object is used to mimic the behaviour of the CT world in the DE domain.
- Once sufficient confidence is gained, a contract is defined.
- Alternative implementations of sensor and actuator objects are made
  - that do not interact with the environment object and act simply as locations for shared variables that are updated by the co-simulation engine.

Environment Model

- A simplified model of the plant that will later be replaced by a CT model
- Built an `Environment` class that can act as (or be called by) a thread.
  - Step operation with `dt` (time since last call)
- Two approaches:
  - Data driven: pre-calculated data is read in and provided to the controller model via the sensor objects
  - Integration: simple implementation of a CT-like integrator
  - Or: a combination of both

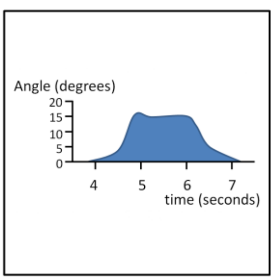
Simple Integration

- Consider a moving object with an acceleration, velocity and position, simulated over some time step, `dt`.
- A simple Euler integration might look like:

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

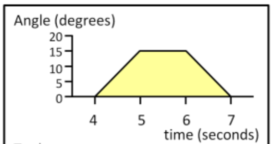
- Simplifying assumptions used, e.g.
  - acceleration is constant, or
  - motors have no acceleration and instantly reach speed

Approximating CT Behaviour



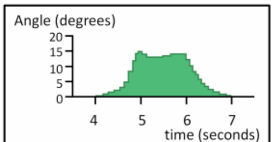
- Linear approximations are okay for the plant model, what about non-linear (e.g. user input)?
- E.g. the plot here might represent user input on the self-balancing scooter
  - it is high fidelity
  - but for testing safety and modes (e.g. start-up), only an approximation will do

Finding Approximations



Tuples:

(0.0, 0.0)
(4.0, 0.0)
(5.0, 15.0)
(6.0, 15.0)
(7.0, 0.0)



File entries:

```
"time","angle"
4.0,0.0
4.1,0.1
4.2,0.8
4.3,2.1
...
```

- Tuples
  - create a sequence of time/value pairs
  - seq of real * real
  - change at the given time, interpolate between times
- Data input
  - use real measured data or generate data
  - Store in CSV and read in at the given time
  - `CSV`freadval[seq of real](filename)`

Summary

- VDM-RT is used to build controllers in Crescendo
  - it is object-oriented, supports inheritance
  - classes are divided into sections
    - instance variables, operations, functions, values, thread, sync
  - there is an internal clock that is synchronised with 20-sim; all expressions take time and increase the internal clock
- Concurrency in VDM-RT
  - threading defined per class
  - asynchronous operations spawn new thread
  - synchronisation mechanisms (permission predicates, mutex)
- DE-first
  - simplified plant model
  - runs as a thread, like a simple simulator
  - approximations of CT behaviour

Practical 2:
Line-following Robot Co-model

John Fitzgerald
Peter Gorm Larsen
Ken Pierce

Instructions

- Extract *Practical\Practical2.zip* from the memory stick
  - this will place a Robot folder on your hard drive
- Navigate to the extracted folder and follow the instructions in *Practical2-Instructions.pdf*