VDM-RT for Co-simulation

John Fitzgerald
Peter Gorm Larsen

Newcastle University
AARHUS UNIVERSITY

Background: VDM

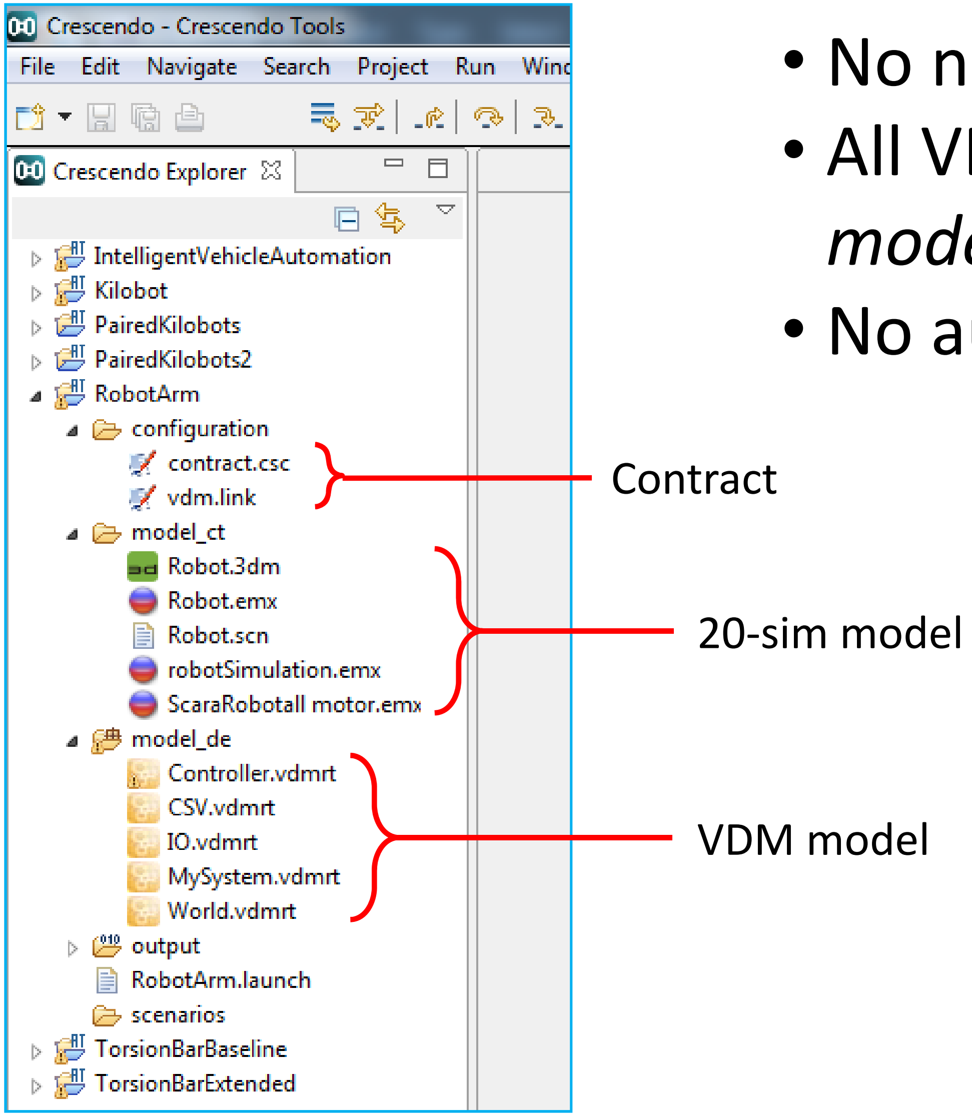
- Our goal: well-founded but accessible modelling & analysis technology
- VDMTools → Overture → Crescendo → Symphony
  - Pragmatic development methodologies
  - Industry applications
- VDM: Model-oriented specification language
  - Extended with objects and real time.
  - Basic tools for static analysis
  - 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
- DE-first modelling in Crescendo
  - Modelling approximations

Crescendo Workspace

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



Debugging

- Printing:

```
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 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

- Sections (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)
  - Default is `private` (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
```

- Functions are pure
  - No side effects
  - Cannot access instance variables
- No return keyword:
  - Value of function application is defined by an expression representing the returned value of the correct type
- Useful for auxiliary / helper calculations
- Signature above definition
  `real * int * bool -> real`
- No loops
  - 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 (only at the start)

```
Step() == (
    dcl x: real := 0;
```

- Parentheses: `( )`, not `{}`
- Different arrow from 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
```

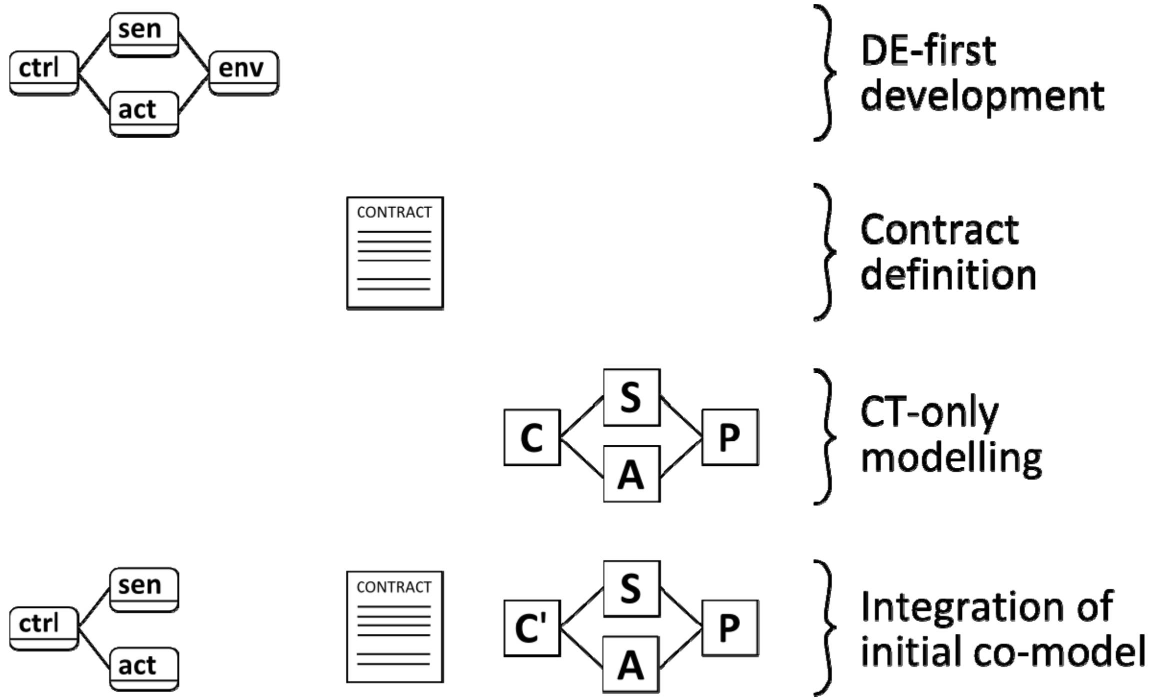
- Special class for CPU and deployment
- Can only define instance variables and a constructor
- CPU speed in (simulated) MIPS
  - getting a model within ~20% of the real thing is typically “good enough”

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

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

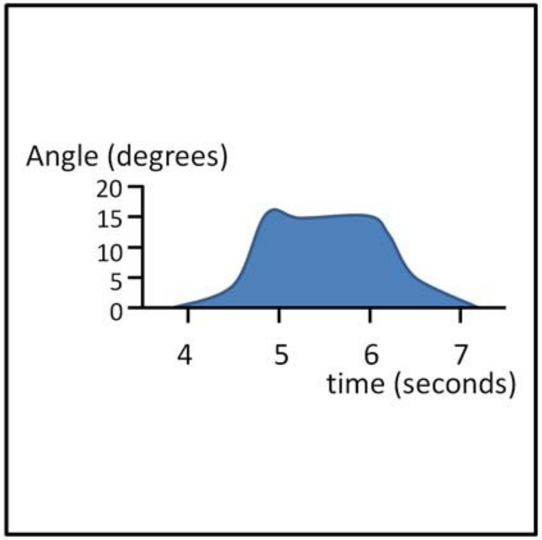
Simple Integration

- 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;
```

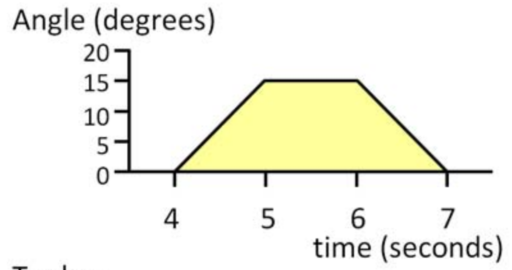
- Simplifying assumptions used, e.g.
  - acceleration is constant, or
  - motors have no acceleration and instantly reach speed

Approximating CT Behaviour



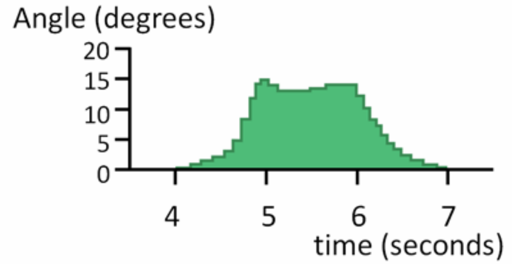
- Linear approximations are okay for the plant model, what about non-linear (e.g. user input)?
- e.g. the plot here might represent user input on the self-balancing scooter
  - it is high fidelity
  - but for testing safety and modes (e.g. start-up), only an approximation will do

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
- DE-first
  - simplified plant model
  - runs as a thread, like a simple simulator
  - approximations of CT behaviour

Practical:
Line-following Robot Co-model

John Fitzgerald
Peter Gorm Larsen

Instructions

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