Collaborative Design of Embedded Systems – co-modelling and co-simulation
FBK seminar 15-12-2016

Marcel Verhoef@esa.int

Contents

- Roots of the methodology
- Applications
- The family of languages
- A flavor of VDM
- Tool support
- Beyond VDM – co-modelling and co-simulation
- Some examples

Roots of the method

- Invented at IBM's labs in Vienna in the (19)70's
- Originally designed to define semantics of other programming languages: CHILL, ADA, MODULA-2, …
- Evolved into a family of general purpose formal specification languages
- Intuitive model-oriented notational style:
  - Simple and abstract data types (including collections)
  - Invariants to restrict membership
  - Focus on specification of functionality:
    - Referentially transparent functions
    - Operations with side effects on state variables
    - Implicit specification (pre/post conditions)
    - Explicit specification (functional or imperative, with optional pre/post)
  - Refinement as principle technique to elaborate specifications to design

Some (large scale) Applications of VDM

- British Aerospace (UK) : secure gateway
- GAO Gmbh (D) : bank note processing system
- JFITS (JP) : trading room application specification
- Flower auction Aalsmeer (NL) : auction clock system
- DoD (NL) : complex data handling subsystem
- Sony / Felica (JP) : firmware for NFC chip
- Specification of SPOT-4 payload software
- RTRI (JP) : railway interlocking specification
- Lockheed-Martin (USA) : JSF ground support equipment
- Rijkswaterstaat (NL): road congestion warning system

Key success factor(s)

- Notation is easy to learn
- Robust tool support and (a lot of) documentation
- Active research and user community (albeit small)
- Formal methods “light”: focus on prototyping and test rather than proof (get results fast)
- Simple to adopt and embed in current practice
- Support design dialogue very early in life-cycle
- Use *abstraction* as primary technique to focus on essential elements of the problem (requirements)

Definition of Abstraction

- the act of *withdrawing* or *removing* something
- the act or process of *leaving out* of consideration one or more properties of a complex object so as to attend to others

=> Remove detail (simplify) and focus (selection)

- a *general concept* formed by extracting *common* features from specific examples
- the process of formulating *general concepts* by abstracting *common* properties of instances

=> generalisation (core or essence)

Models and Modelling

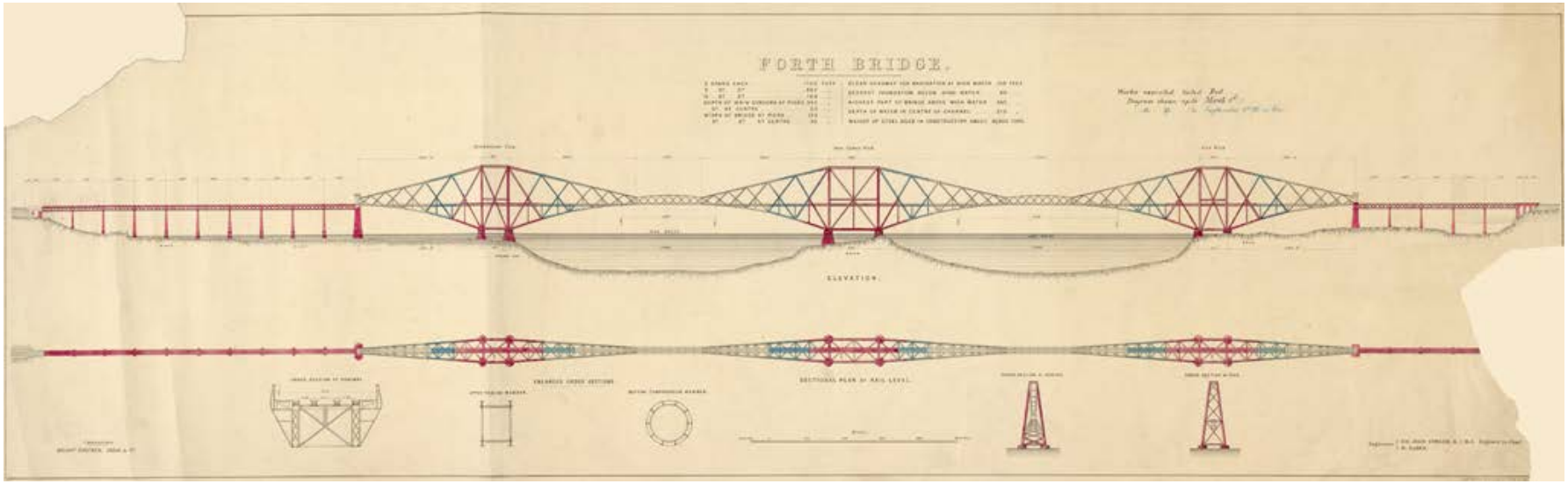
- A model is a description from which detail has been *removed* in a systematic manner and for a particular *purpose*.

- A simplification of reality intended to promote *understanding*, *reasoning* and *analysis*.

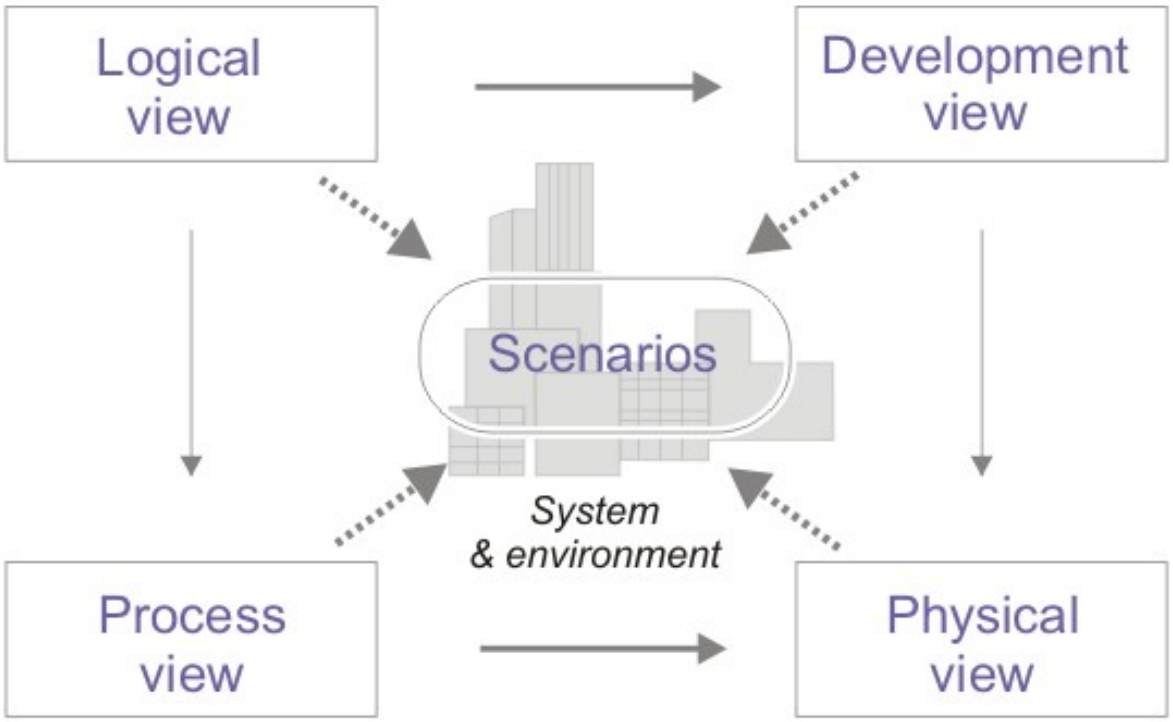
- Models are the most important engineering tool; they allow us to understand and analyze *large and complex problems*.

Abstract models (1)

Abstract models (2)



Abstract models of software (?)



Validation Techniques (gaining model trust)

- Inspection: organized process of examining the model alongside domain experts (build up domain knowledge explicitly – terms of reference)
- Static Analysis: automatic checks of syntax & type correctness, detect unusual features (assess intrinsic model quality)
- Prototyping: execute the model and interact / explore to validate modelling assumptions (i.e. connecting a UI mock-up)
- Testing: execute the model and check outcomes against explicitly formulated expectations (and easily perform regression testing when model is changed)

❖ Model Checking: search the state space of the model exhaustively to find states that violate the properties we are checking; presents a counter example in such case

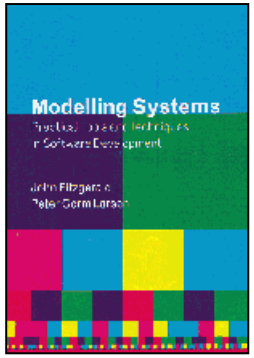
❖ Proof: use a logic to reason symbolically about whole classes of states at once.

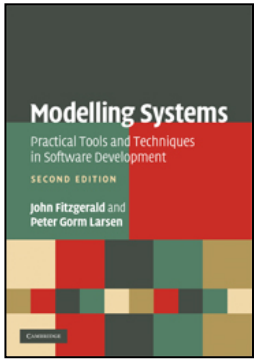
Caveat: use last two techniques only when sufficient confidence is built with first four as investment required to perform these steps is typically higher

Family of Languages

- VDM-SL
  - ISO standard ISO/IEC 13817-1 (1996)
  - Formal syntax, static and denotational semantics
- VDM++
  - Object-orientation
  - Concurrency / synchronization
- VDM-RT
  - Asynchronous operations
  - Timing
  - Deployment
- Caveat: VDM-SL $\subset$ VDM++ $\subset$ VDM-RT

VDM-SL in a nutshell






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

definitions
  state
  types
  values
  functions
  operations
    ...
end <module-name>
```

Interface

Definitions

VDM-SL : basic types

`bool`	Boolean datatype	`false, true`
`nat`	natural numbers (including zero)	0, 1, 2, 3, …
`nat1`	natural numbers (excluding zero)	1, 2, 3, 4, …
`int`	integers	…, -3, -2, -1, 0, 1, 2, 3, …
`rat`	rational numbers	a/b, where a and b are integers, b is not 0
`real`	real numbers	…
`char`	characters	`A, B, C, ...`
`token`	structureless tokens	…
`<A>`	the quote type containing the value `<A>`	…

VDM-SL : sets

```
UGroup = set of UserId
```

- Set enumeration: the set of elements `a`, `b` and `c`: `{a, b, c}`
- Set comprehension: the set of `x` from type `T` such that `P(x)`: `{x | x:T & P(x)}`
- The set of integers in the range `i` to `j`: `{i, ..., j}`
- Set membership, `e` is an element of set `s`: `e in set s`
- Not a member of, `e` is not an element of set `s`: `e not in set s`
- Union of sets `s1` and `s2`: `s1 union s2`
- Intersection of sets `s1` and `s2`: `s1 inter s2`
- Set difference of sets `s1` and `s2`: `s1 \ s2`
- Distributed union of set of sets `s`: `dunion s`
- Proper subset, s1 is a (proper) subset of `s2`: `s1 psubset s2`
- (weak) subset, s1 is a (weak) subset of `s2`: `s1 subset s2`
- The cardinality of set `s`: `card s`

VDM-SL : sequences

```
String = seq of char
```

- Sequence enumeration: the sequence of elements `a`, `b` and `c`: `[a, b, c]`
- Sequence comprehension: sequence of expressions `f(x)` for each `x` of (numeric) type `T` such that `P(x)` holds (`x` values taken in numeric order): `[f(x) | x:T & P(x)]`
- The head (first element) of `s`: `hd s`
- The tail (remaining sequence after head is removed) of `s`: `tl s`
- The length of `s`: `len s`
- The set of elements of `s`: `elems s`
- The `i`^th^ element of `s`: `s(i)`
- The set of indices for the sequence `s`: `inds s`
- The sequence formed by concatenating sequences `s1` and `s2`: `s1^s2`

VDM-SL : mappings

```
Birthdays = map String to Date
```

- Mapping enumeration: `a` maps to `r`, `b` maps to `s`: `{a |-> r, b |-> s}`
- Mapping comprehension: `x` maps to `f(x)` for all `x` for type `T` such that `P(x)`: `{x |-> f(x) | x:T & P(x)}`
- The domain of `m`: `dom m`
- The range of `m`: `rng m`
- Application, `m` applied to `x`: `m(x)`
- Union of mappings `m1` and `m2` (`m1`, `m2` must be consistent where they overlap): `m1 munion m2`
- Override, `m1` overwritten by `m2`: `m1 ++ m2`

VDM-SL : unions, tuple, records

- Union of types `T1,...,Tn`: `T2 | ... | Tn`
- Cartesian product of types `T1,...,Tn`: `T1*T2*...*Tn`
- Composite (Record) type: `T :: f1:T1 ... fn:Tn`

```
SignalColour = <Red> | <Amber> | <FlashingAmber> | <Green>
```

```
Date :: day:nat1 month:nat1 year:nat inv mk_Date(d,m,y) == d <=31 and m<=12
```

VDM-SL : specifying functions

Implicit style (1) – *just focus on interface and properties*

```
SQRTP(x:real) r:real
pre x >=0
post r*r = x and r >= 0
```

Implicit style (2) – *pre- and post conditions can be arbitrarily complex*

```
SqListImp(s:seq of nat)r:seq of nat
post len r = len s and forall i in set inds s & r(i) = s(i)**2
```

Explicit style (2) – *executable specification (can still have pre- and postconditions!)*

```
SqList: seq of nat -> seq of nat
SqList(s) ==
  if s = []
  then []
  else [(hd s)**2] ^ SqList(tl s)
```

VDM-SL : specifying state and operations

Definition of the global state

```
state Register of
  someStateRegister : nat
end
```

Implicit style (showing side effect – state is modified)
caveat: postcondition is a logical expression, not an assignment

```
LOAD(i:nat)
        ext wr someStateRegister:nat
        post someStateRegister = i
```

Implicit style (showing side effect – state is modified)
caveat: postcondition is referring to the previous state: `someStateRegister~`

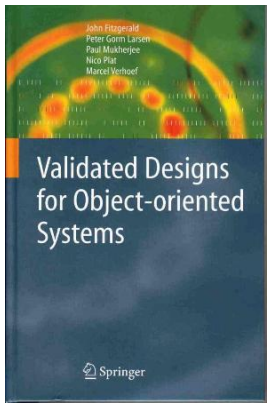
```
ADD(i:nat)
        ext wr someStateRegister : nat
        post someStateRegister = someStateRegister~ + i
```

Explicit style not shown here for brevity

VDM-SL : advanced features

- Optional and function types
- Polymorphic and higher-order functions
- Conditional, Lambda, let-be, let-be-such-that expressions
- Undefined expression
- Type judgement expression
- Pattern matching
- Type bindings
- Block, assignment, loop and conditional statements
- Exception handling, error and non-deterministic choice
- And much more … see language reference manual

VDM++ in a nutshell




```
class <class-name>

  instance variables
    ...

  types
  values
  functions
  operations

  thread
    ...

  sync
    ...

  traces
    ...

end <class-name>
```

Internal object state

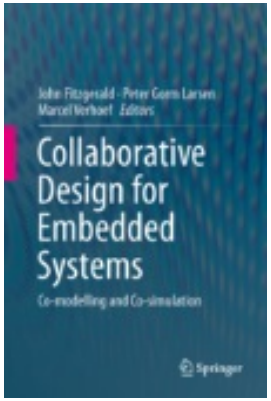
Definitions

Dynamic behavior

Synchronization control

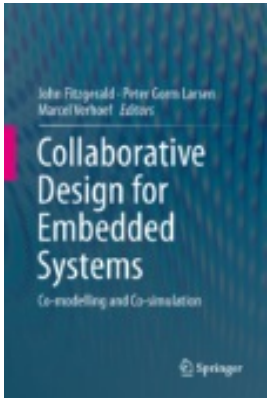
Test automation support

VDM-RT in a nutshell (1)



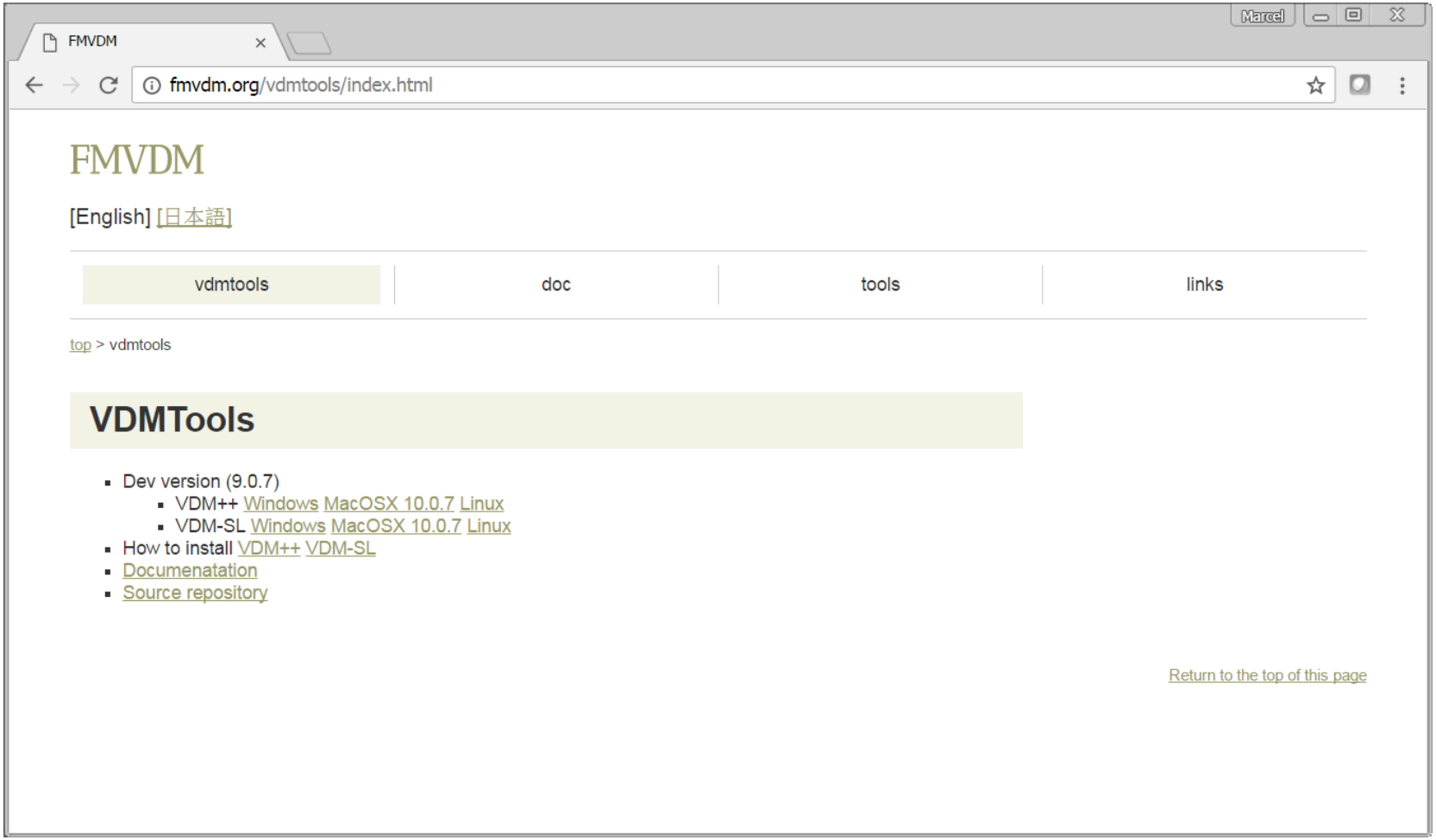

```
class Controller

instance variables
 private i : Interface := new Interface()

operations
 async public Open:() ==> ()
 Open() == duration (50) i.SetValve(true);

 async public Close:() ==> ()
 Close() == cycles (1000) i.SetValve(false);

sync
 mutex(Open, Close);
 mutex(Open);
 mutex(Close)

end Controller
```

VDM-RT in a nutshell (2)

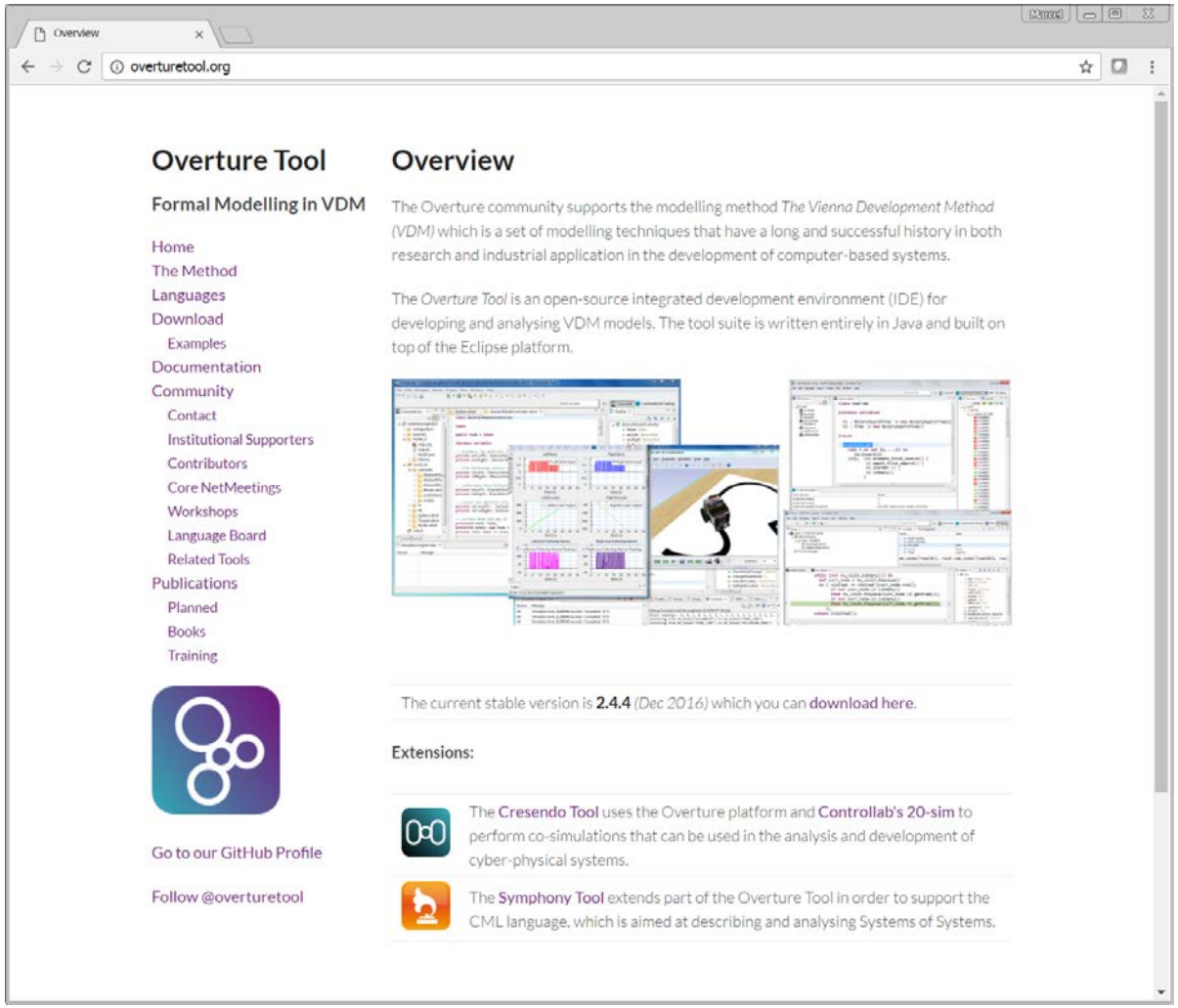



```
system WaterTank

instance variables
  public static controller : [Controller] := nil;

  -- architecture definition
  cpu1 : CPU := new CPU(<FP>, 20);

operations

  public WaterTank : () ==> WaterTank
  WaterTank () == (
    controller := new Controller();
    cpu1.deploy(controller, "Controller");
  );

end WaterTank
```

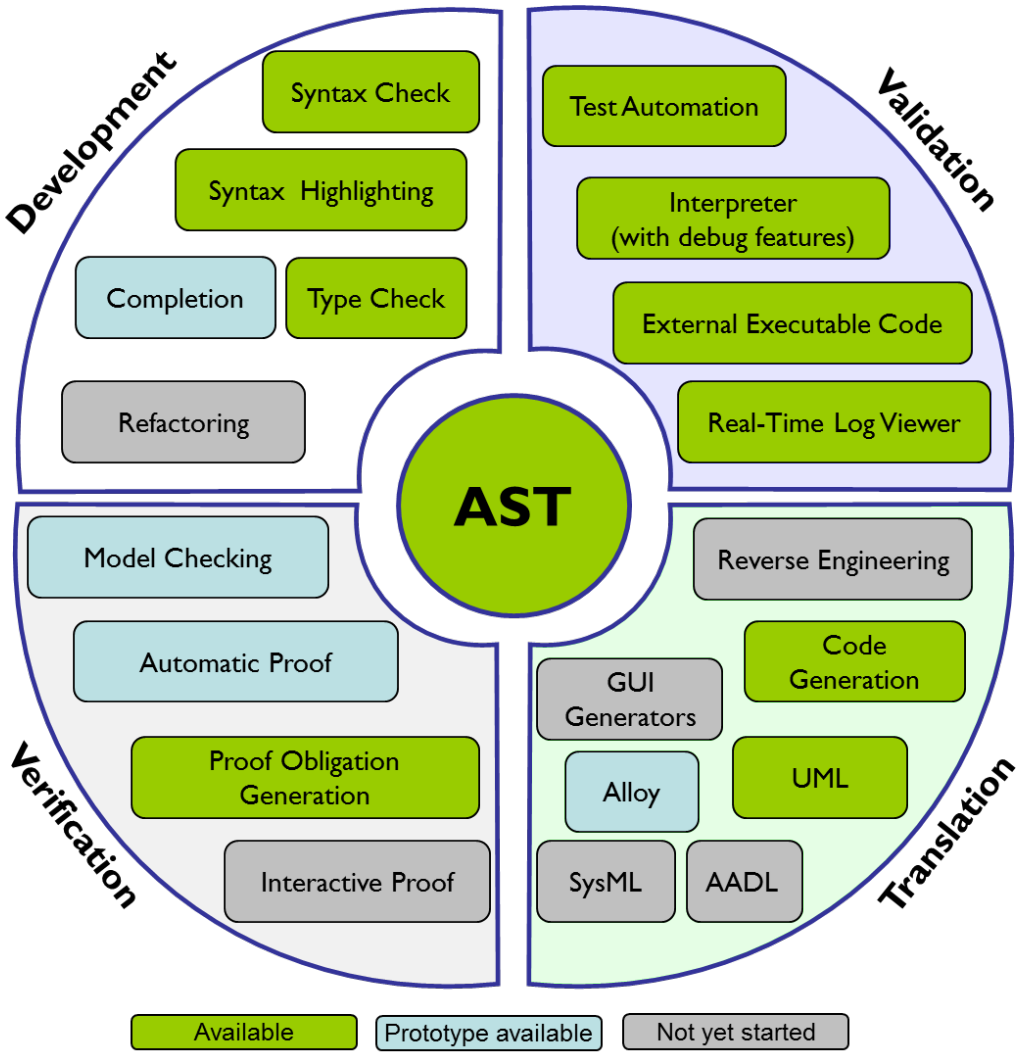
Tool support (1)



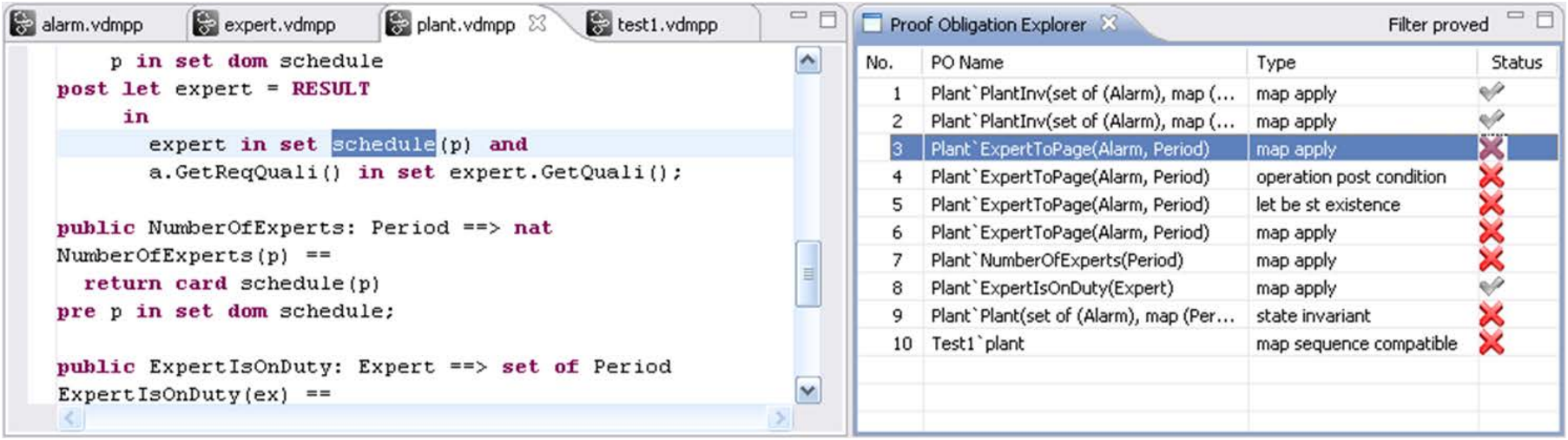
Tool support (2)



Tool support (3) - Overture

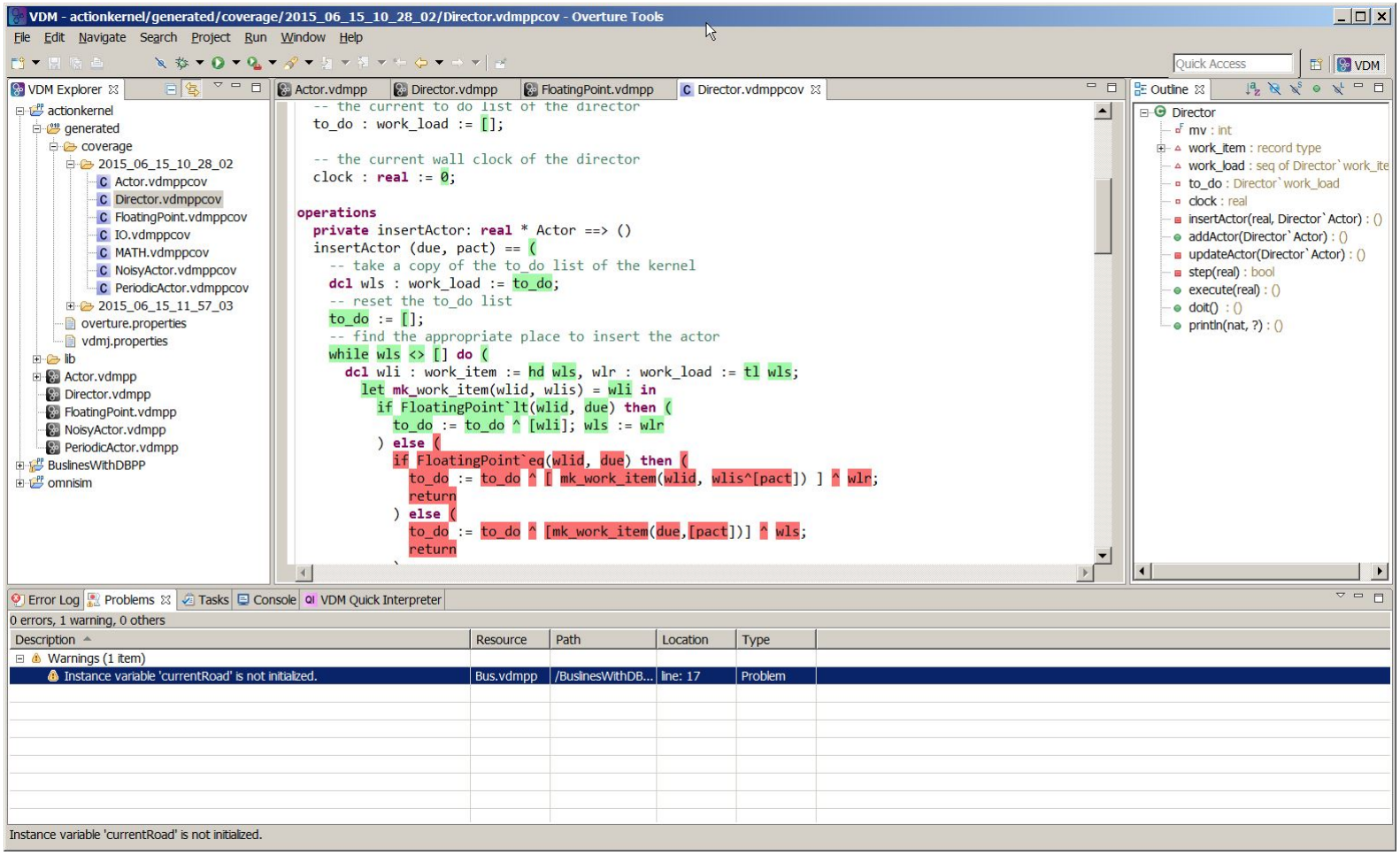


Useful features – enhanced static analysis

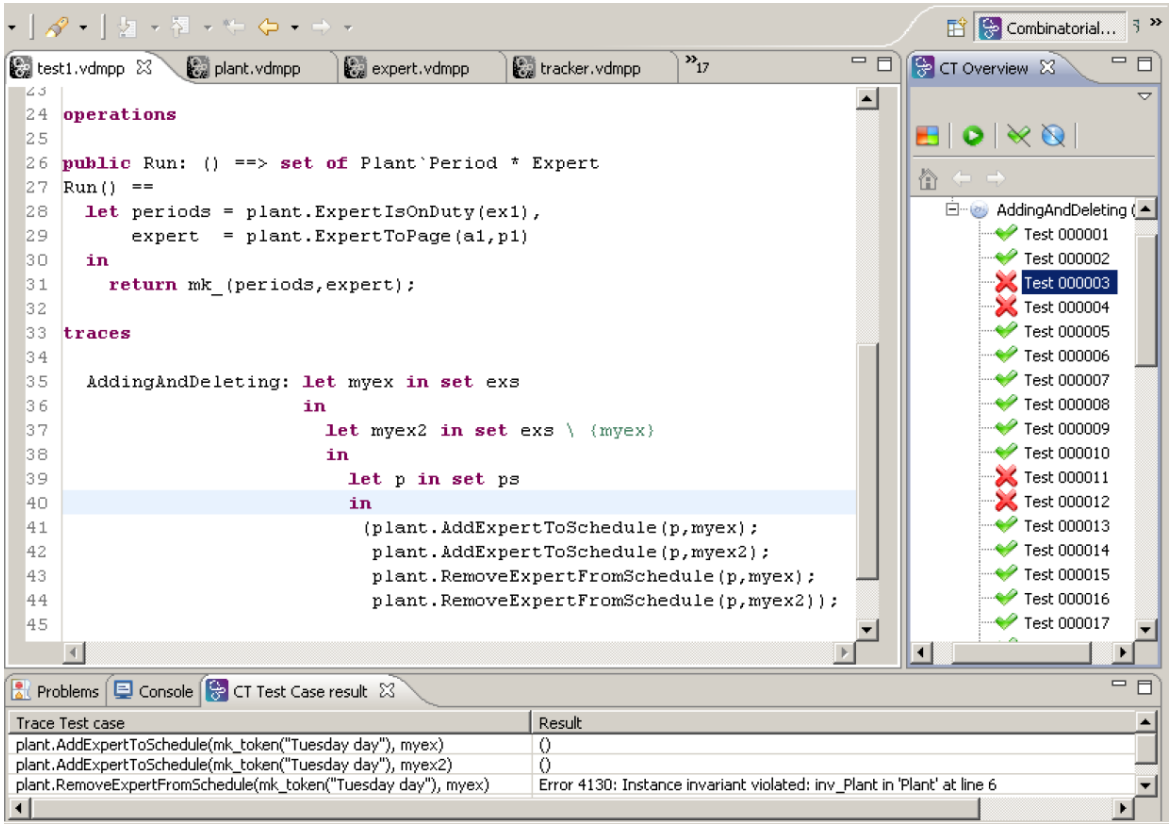


Caveat: VDM is a strongly typed language, but not always compile time analyzable due to its fundamental set-theoretic basis: three valued logic and logic of partial functions. A specification is considered correct if there exists at least one valid interpretation of the model. The enhanced static analysis identifies possible weaknesses that can be addressed by either reinforcing the model or by proving that the some assumption will always hold at run-time.

Useful features – test coverage



Useful features – combinatorial testing



Traces specifications can be used to describe a large number of test cases that are used to feed a small piece of specification (here a statement block) The tool executes each individual test case and reports the result. Failing test cases can be isolated and loaded in the interpreter for later analysis. Moreover, a JUNIT like framework exists to allow another style of testing.

Usefull features – specifications in-line in deliverables

Exercise 3.3 Use the interpreter to evaluate the following expression: `new Test1().Run()`.
□

3.8 Test coverage

It is often useful to know how much of a model has been exercised by a set of tests[12]. This gives some insight into the thoroughness of a test suite and may also help to identify parts of the model that have not been assessed, allowing new tests to be devised to cover these. When any evaluation is performed on a VDM++ model, the interpreter records the lines of the VDM++ model that are executed. This permits the line coverage to be examined after a test to identify the parts of the VDM++ model that have not yet been exercised – coverage is cumulative, so a set of tests can be executed and their total coverage examined at the end.

In our simple example, the different tests in the exercise above does cause the majority of the VDM++ model to be executed, but for demonstration purposes let us start by cleaning the model (select "`Project`" at the top and select `Clean...` in the menu). If we simply take the `AlarmPP` debug launch configuration the `ExpertIsOnDuty` and `ExpertToPage` operations in `plant.vdmpp` are called by the `Run` function. Remember that whenever test coverage information is desired the `Generate Coverage` option must be selected as shown in Figure 3.11[13]. Once the debugger has completed and the result is written out in the `console` it is possible to right click on the `AlarmPP` project and select the *Latex → Latex coverage*. The coverage information that have been gathered in any expressions that have been debugged since the last change to a file have been saved or the project have been cleaned will be turned into a pdf file. The `AlarmPP.pdf` file is placed in the `generated/latex` directory as shown in Figure 3.15 and it includes the VDM definitions from all the files included in the project including coverage information. Note that whenever the model is adjusted or it is cleaned so it gets type checked again all the files in the `generated` directory are deleted.

The coverage information is provided in a way where uncovered expressions are shown in red in the generated pdf file. In addition to the content of each VDM++ source file a table with coverage overview is provided in tabular form. For the `plant.vdmpp` file this looks like:

Function or operation	Coverage	Calls
ExpertIsOnDuty	100.0%	1
ExpertToPage	100.0%	1
NumberOfExperts	0.0%	0
Plant	100.0%	1
PlantInv	100.0%	2
plant.vdmpp	89.0%	5

[12]Note that this feature is not yet supported for models using unicode characters such a Japanese identifiers.
[13]Note that using this feature required pdflatex to be installed

3.9 Combinatorial Testing

The previous sections have shown how to test and debug models manually. However, Overture also contains a feature enabling more automation in the testing process, making more comprehensive high-volume testing feasible. It is possible to write regular expressions, as *traces*, that one would like to expand into a large set of individual tests. When new traces are incorporated in a VDM project you may need to press the Refresh button (↻) in the *CT Overview* view.

In order to illustrate how this can be used, we extend the `Plant` class with two additional operations for adding and removing experts from a given schedule. Both operations take a given `Period` and an `Expert` and then update the `schedule` instance variable from the `Plant` class. The `AddExpertToSchedule` operation can be defined as:

```
public AddExpertToSchedule: Period * Expert ==> ()
AddExpertToSchedule(p,ex) ==
  schedule(p) := if p in set dom schedule
                 then schedule(p) union {ex}
                 else {ex};
```

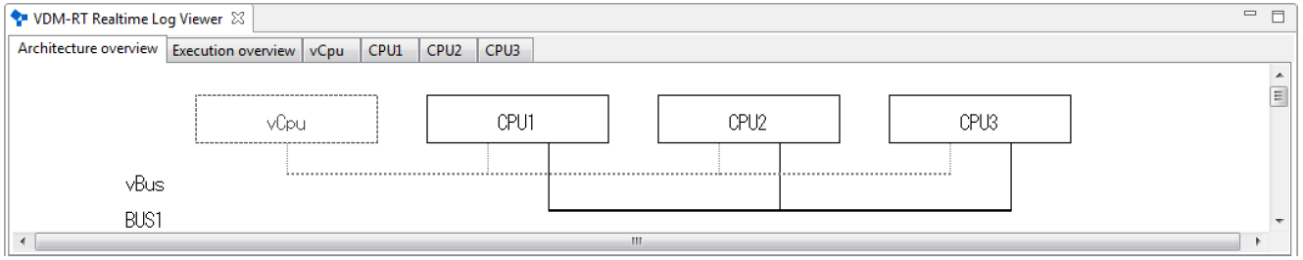
and the `RemoveExpertFromSchedule` operation can be expressed as:

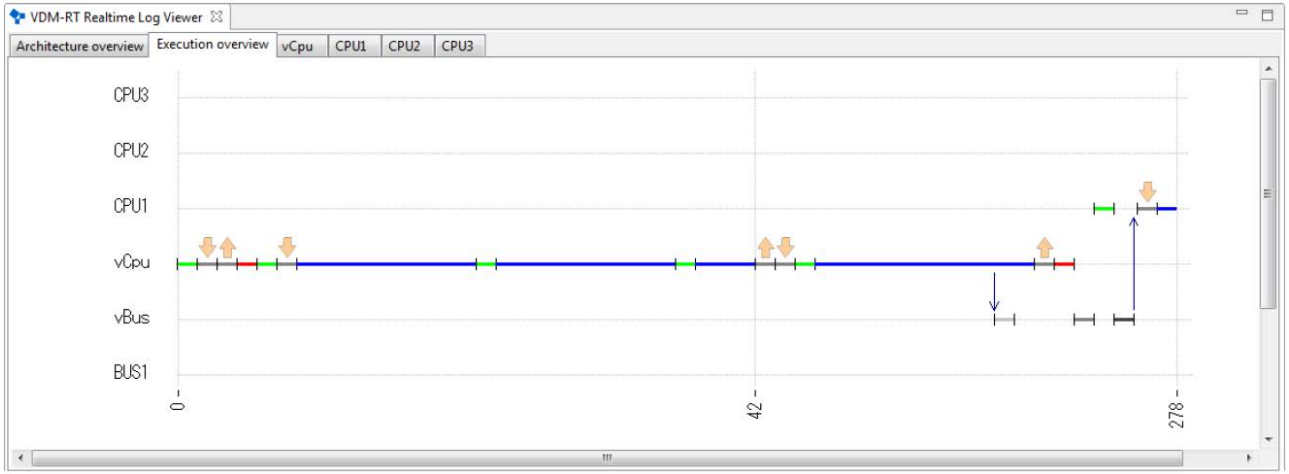
```
public RemoveExpertFromSchedule: Period * Expert ==> ()
RemoveExpertFromSchedule(p,ex) ==
  let exs = schedule(p) in
    schedule := if card exs = 1
                then {p} <-: schedule
                else schedule ++ {p |-> exs \ {ex}}
pre p in set dom schedule;
```

Note that `RemoveExpertFromSchedule` contains a deliberate error. It fails to take account of the invariant so the operation can leave the `Plant` in a state where it cannot be guaranteed that experts with the right qualifications are available in the periods that have been scheduled. `AddExpertToSchedule` has a similar error. If nobody is scheduled at the period provided as an argument, and the expert added for the schedule at this period does not have all the necessary qualifications, the invariant will again be violated. In fact this means that one would probably have to change the signature of this operation such that it instead of taking a simple expert would take a collection of experts. Instead of adding the two operations manually, use the `Alarm++tracesPP` project.

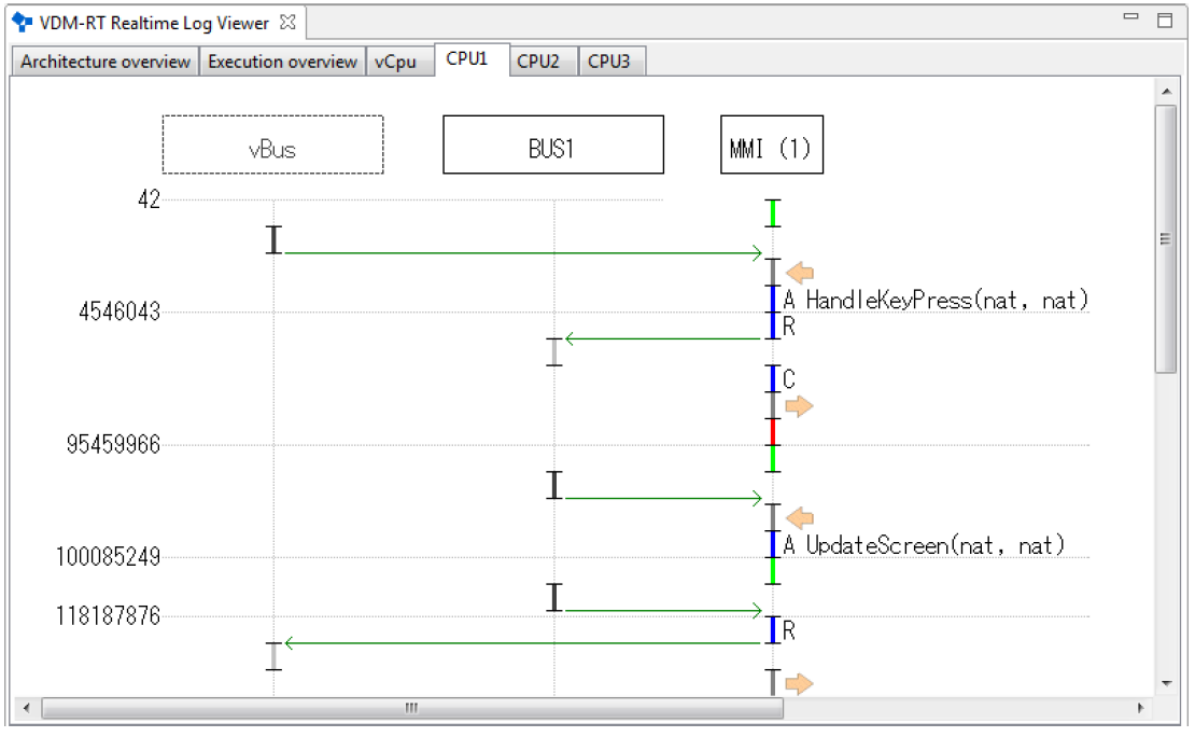
We could use the debugger presented above to test these two new operations manually, but we can also automate a part of this process. In order to do the automation, Overture needs to know about the combinations of operation calls that you would like to have carried out, so it is necessary to write a kind of regular expression called a *trace*. VDM++ has been extended such that traces can be written directly as a part of a VDM++ model. In our case, inside the `Test1` class one can

Useful features – VDM-RT runtime analysis (1)



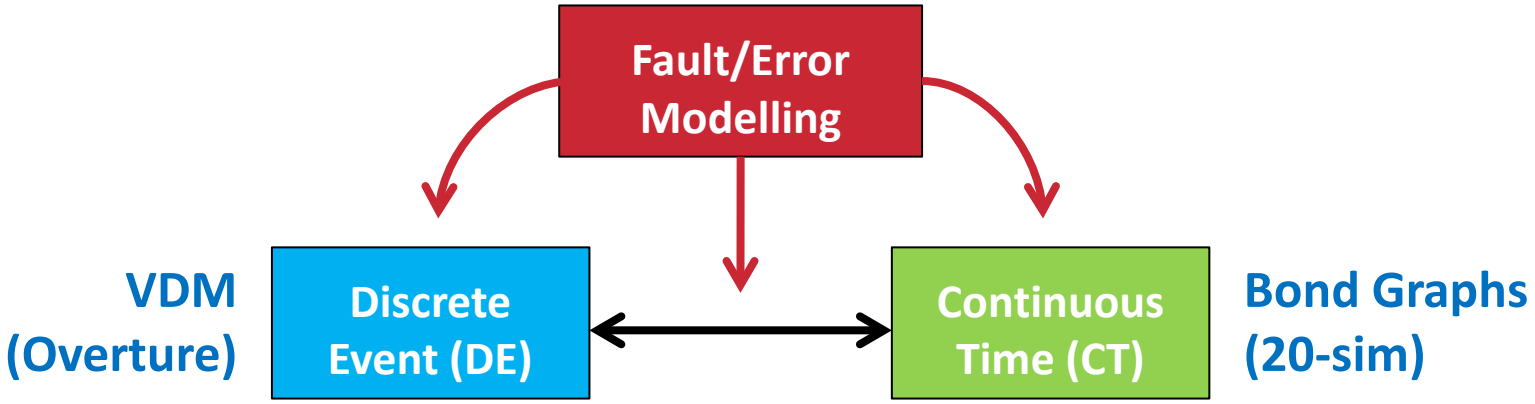


Useful features – VDM-RT runtime analysis (2)

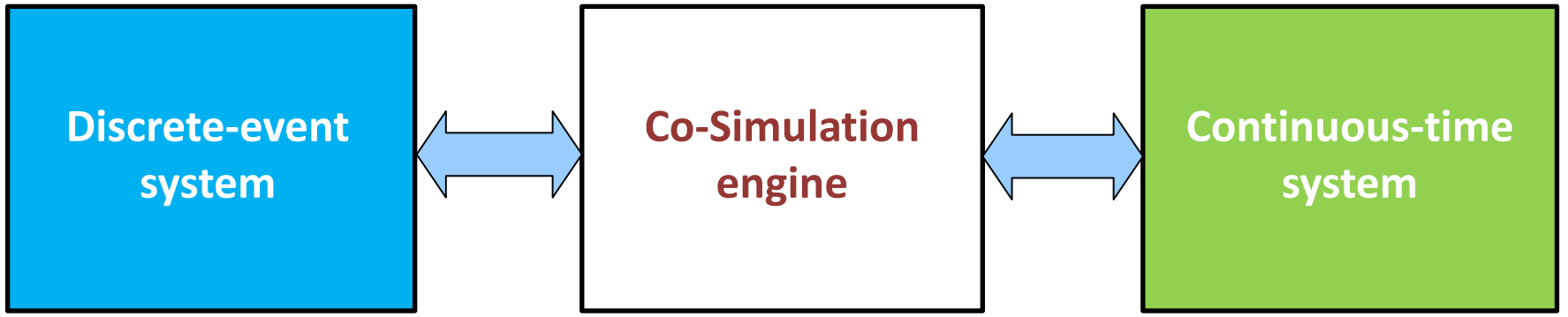


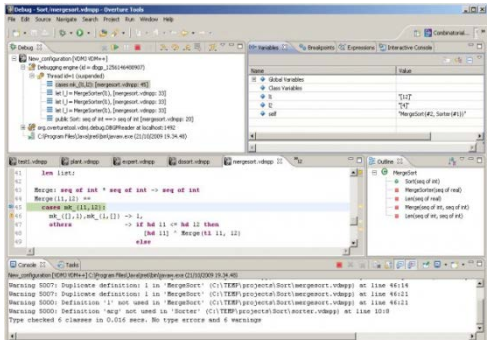
status	name	expression	src time	src thread	dest time	dest thread
FAIL	C1	separate(#fin(MMI`UpdateScreen),#fin(MMI`UpdateScreen),500000000)	118254267	14	209165360	15
FAIL	C1	separate(#fin(MMI`UpdateScreen),#fin(MMI`UpdateScreen),500000000)	209165360	15	300076453	16

Beyond VDM – co-simulation

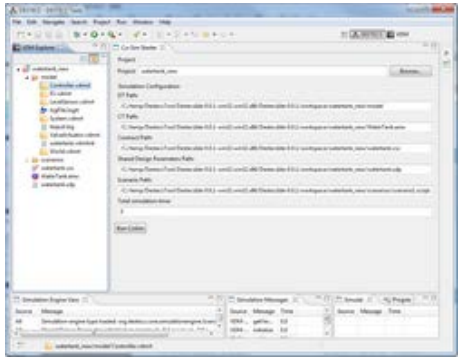


Crescendo: co-simulation (1)

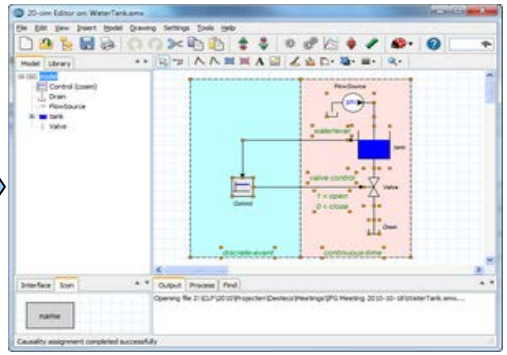




Overture

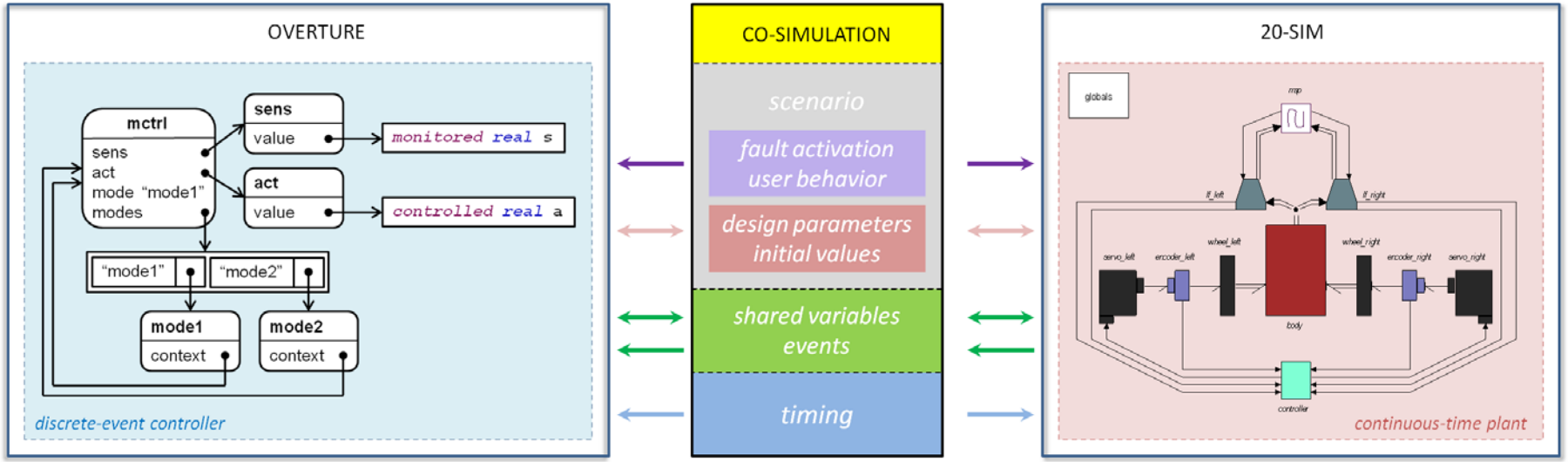


Crescendo



20-sim
(or Matlab/Simulink)

Crescendo: co-simulation (2)



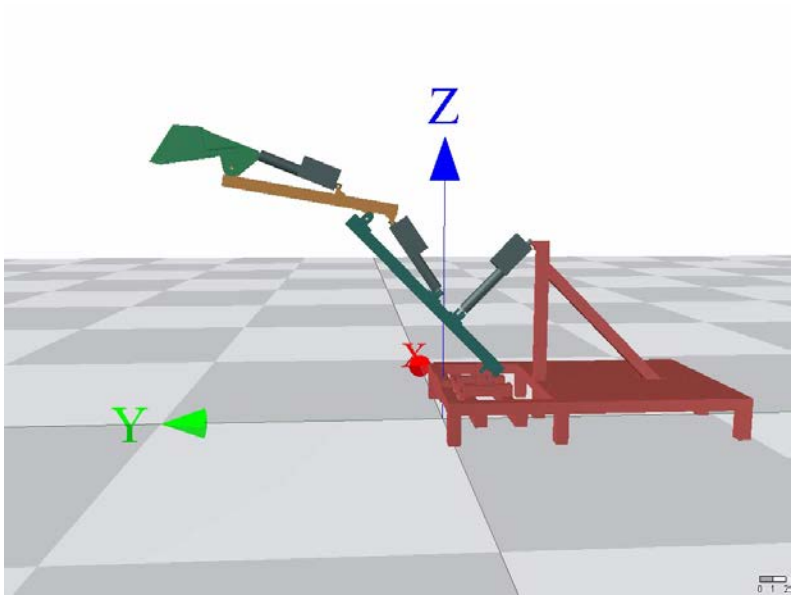
Language: VDM-RT

- Time
- Concurrency
- Architecture
- Deployment

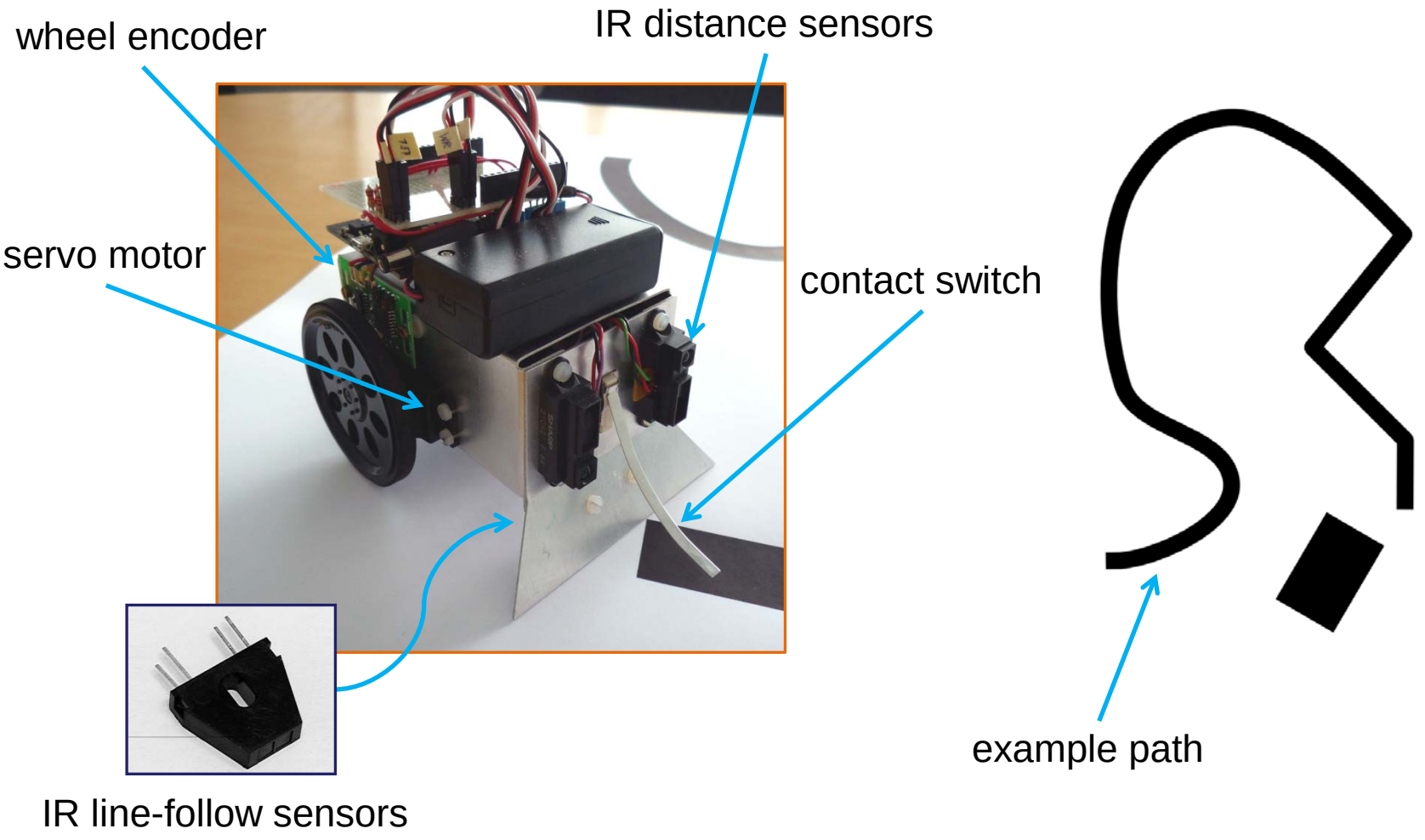
Language: Bond Graphs

- Mechanics
- Electronics
- Hydraulics
- Pneumatics
- Thermodynamics

Crescendo: co-simulation (3)

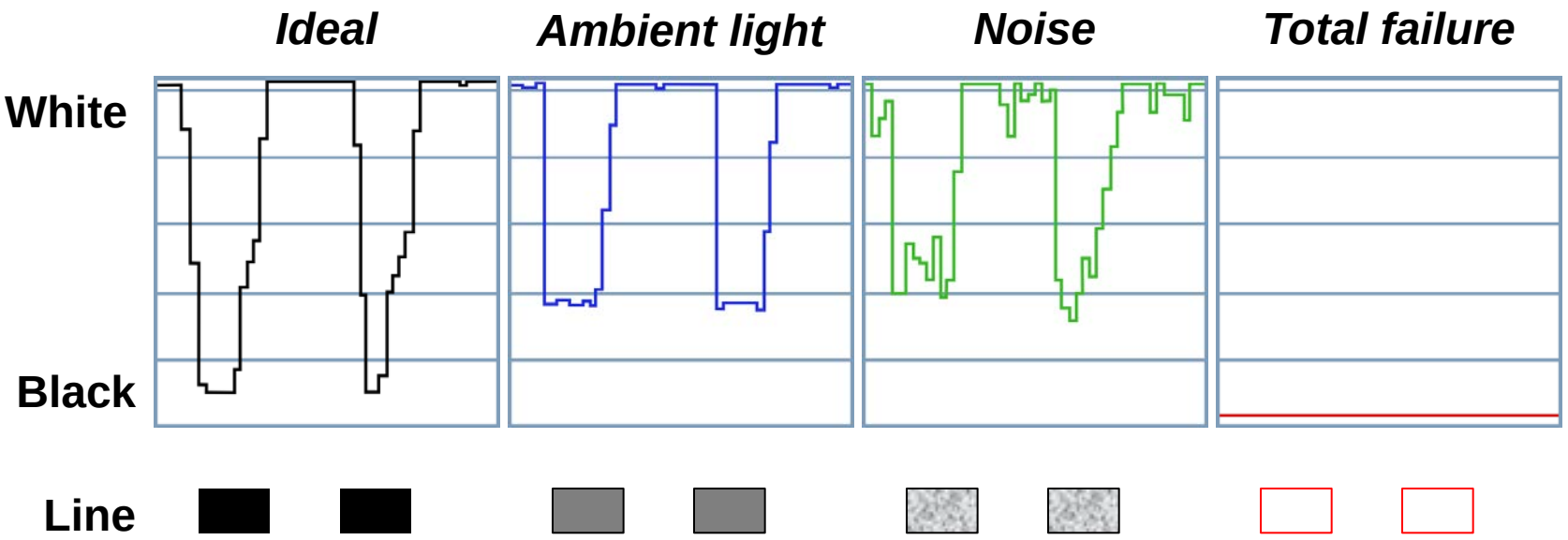


Illustrative Case Study: Line-Follow Robot



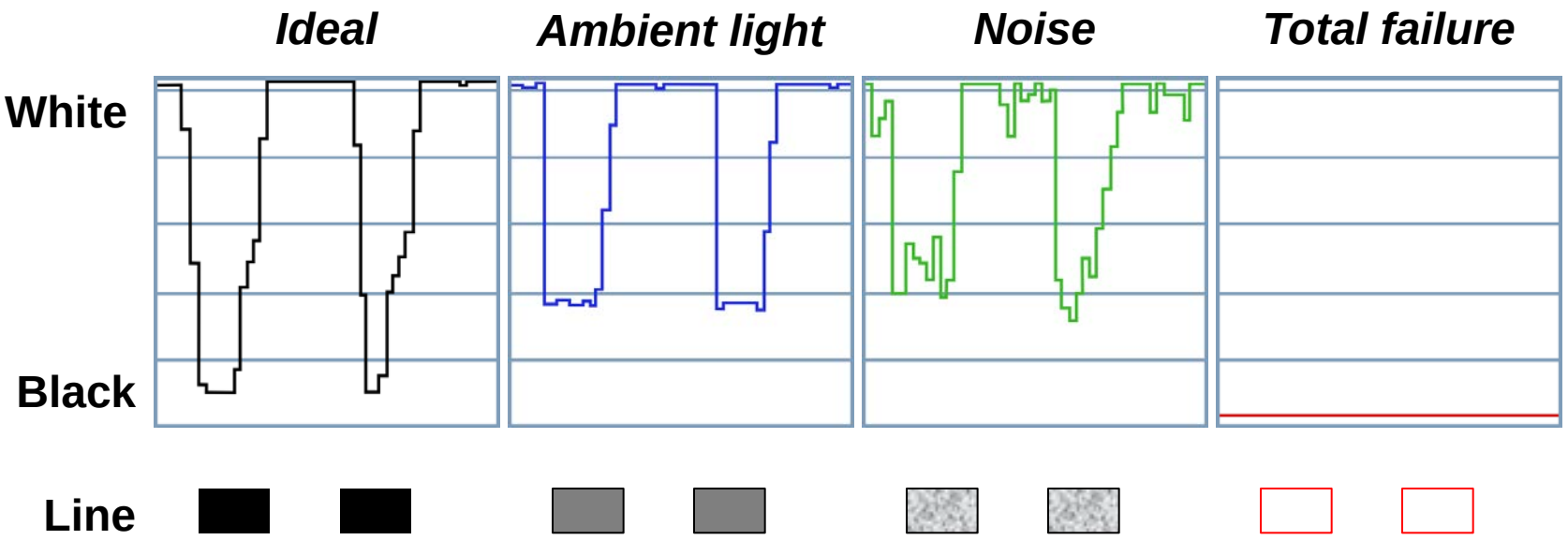
Fault Modelling

- Initial model behaviour is ideal
- Add realistic and faulty behaviour
  - Ambient light levels affect readings (black level)
  - Realistic sensor noise
  - Total failure



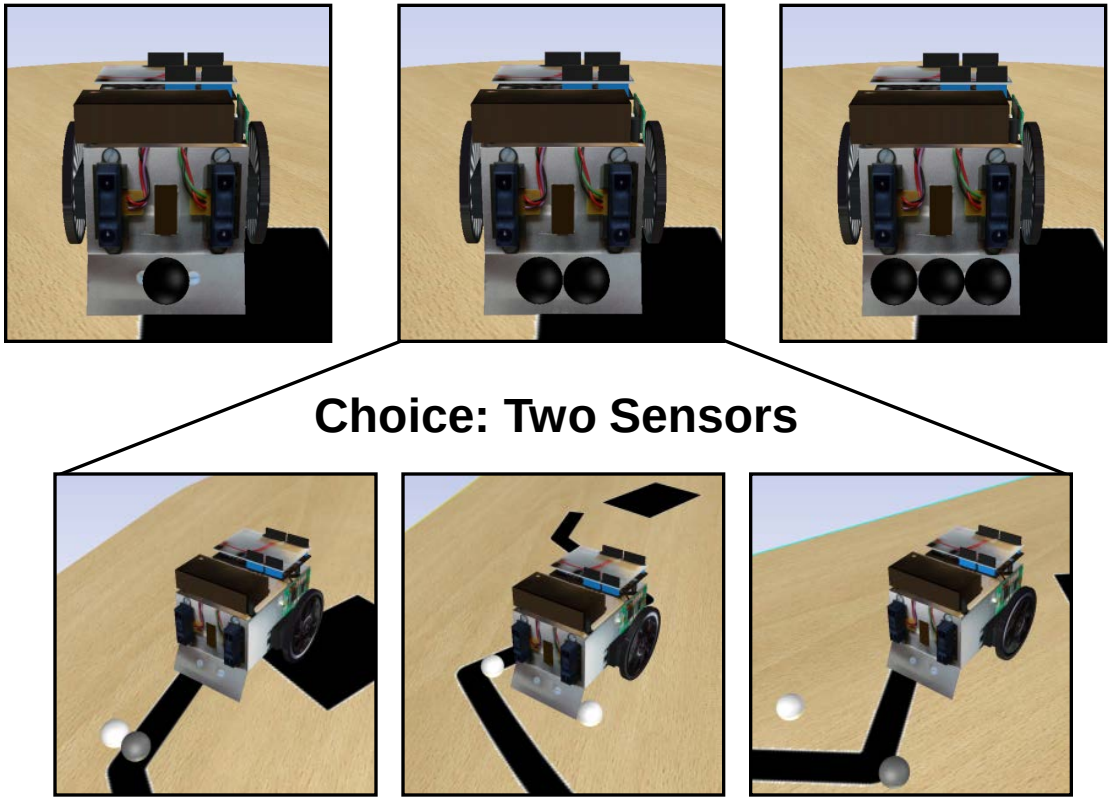
Fault Tolerance

- Light levels: calibration mode
- Sensor failure: one-sensor mode
- Noise: filtering



Design Space Exploration (DSE)

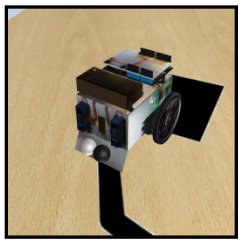
- Building and evaluating models to reach a design
- Design choices restrict the design space

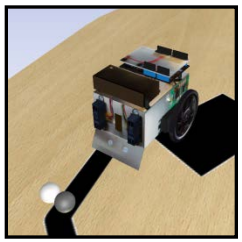


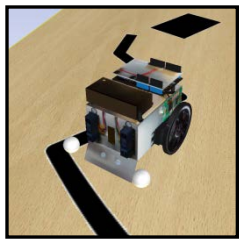
Automated Co-model Analysis (ACA)

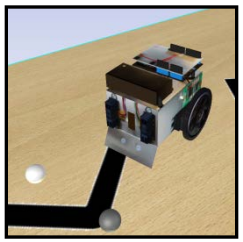
- Tool-support for DSE
- Run multiple co-simulations to gather results

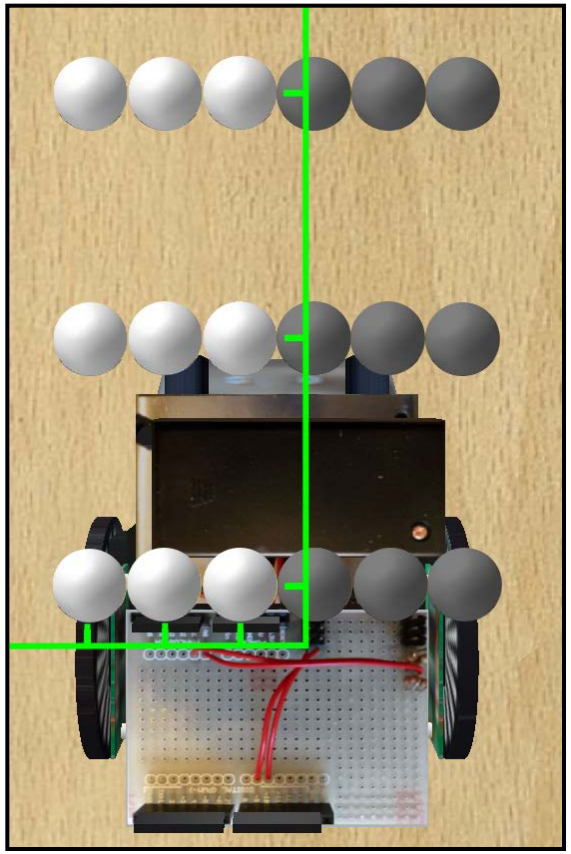
		Longitudinal sensor offset		
		0.01m	**0.07m**	**0.13m**
Lateral sensor offset	**0.01m**	(a)	(b)	(c)
	0.03m	(d)	(e)	(f)
	0.05m	(g)	(h)	(i)


(b)


(c)


(h)

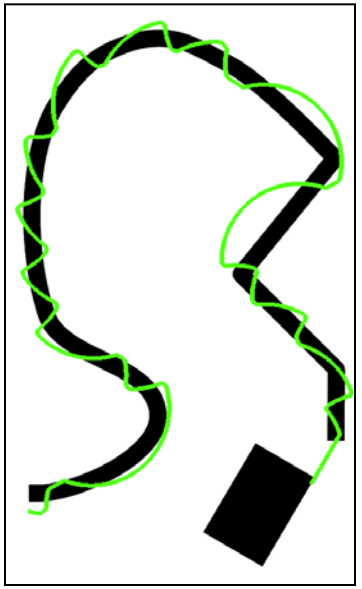

(i)



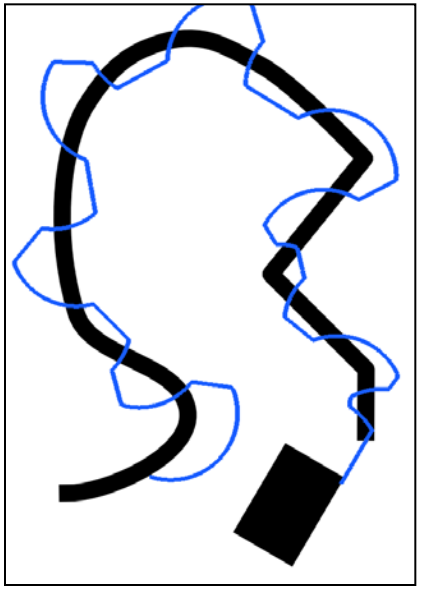
Results and Rankings

- Results can be graphical or numerical
- Designs can be evaluated by ranking functions

		Metric*				
Rank	Design	A	B	C	D	Mean Rank
1	**(b)**	**1**	**5**	**1**	**2**	**2.2**
2	(f)	7	2	4	1	3.5
3	(a)	2	8	2	4	4.0
4	(e)	3	6	3	5	4.2
5	(i)	9	1	5	3	4.5
6	(c)	5	3	6	8	5.5
7	(d)	6	4	7	7	6.0
8	(h)	4	7	8	9	7.0
9	(j)	8	9	9	6	8.0



(b)



(h)

*** A = distance, B = energy, C = deviation area, D = maximum deviation**

Summary (1)

- Pragmatic approach to formal specification (no *all-or-nothing* paradigm)
- Focus on capturing requirements succinctly
- Small abstract models – easy to understand, maintain and explain
- Support design dialogue – get value for money fast
  - *gain better problem domain understanding*
  - *fewer residual errors in downstream design artifacts (hence: less rework)*
  - *early validation and test creates opportunity to reuse test cases*
- Not a panacea or “silver bullet”; abstraction still requires skill!

- Similar techniques: Z, B-method, Event-B, ASM
  (but these focus more on proof and refinement)
- Many of the VDM language concepts now “available” in high-level languages: (OCA)ML, Haskell, Python, Java, Ada, Eiffel
- Nevertheless modelling $\neq$ coding!

Summary (2) – useful links

- TOOLS:
  - (open source) Overture (all VDM dialects) : www.overturetool.org (tools, manuals, examples, books)
  - (open source) Crescendo (co-sim) : www.crescendotool.org
  - (commercial) 20-sim : http://www.20sim.com/
  - (open source) Modelica : https://www.modelica.org/

- PROJECTS:
  - DESTECS (completed in 2012) : www.destecs.org
  - INTO-CPS (on-going H2020) : http://into-cps.au.dk/