Tutorial: Design of Real-time Embedded Control Systems using VDM++ and Bond graphs (part 1 of 2)

Professor Peter Gorm Larsen  
Engineering College of Aarhus (IHA)  
(pgl@iha.dk)

dr. ir. Marcel Verhoef  
Chess Embedded Technology B.V.  
(Marcel.Verhoef@chess.nl)
Programme for the day (part 1)

• Morning (before coffee break)
  • Background to the tutorial
  • Introduction to VDM-RT notation
  • Overview of the Overture tools
  • Example applications and installation of all tools

• Morning (after coffee break)
  • Practical case study
  • Round up (questions, discussions and conclusion(s))
Programme for the day (part 2)

• Afternoon (before tea break)
  • Background to co-simulation
  • Introduction to the Bond-graph notation
  • Overview of the 20-sim tool
  • Example applications

• Afternoon (after tea break)
  • Practical case study
  • Example application on industrial case study
  • Round up (questions, discussions and conclusion(s))
Introduction: Inspiration

• Inspiration
  • Use **collaborative multidisciplinary** design of Embedded Systems
  • **Rapid** construction and evaluation of system models
  • Evaluated on **industrial** applications

• Need because of Embedded Systems
  • More demanding functional & non-functional requirements
    • Reliability, Fault Tolerance
  • Increasingly distributed
    • More design possibilities, and faults
DESTECS approach

• Methods and Open tools
  - Model-based approach for collaborative design of Embedded Control Systems

• Co-simulation
  - Different tools, reflecting relevant aspects of design
  - Rapid, consistent analysis & comparison of models

• Advances needed in
  - Continuous time modeling
  - Discrete event modeling
  - Fault modeling and fault tolerance
  - Open tool frameworks
Model Based Design

Design Process:
• Gather requirements and make paper specification
• Mathematical models linked to specifications
• Simulation:
  • automatic code generation and testing of the models. Eliminate errors
• Detailed Design
• Verification and Validation Testing

Advantage:
• Testing and verification throughout the design process
• Same teams during all stages
• Less design iterations
Consortium

Partners
  University of Twente
  Newcastle University
  Engineering College of Aarhus
  C.H.e.S.S. Embedded Technology B.V.
  Controllab Products B.V.
  Neopost Technologies B.V.
  Verhaert New Products and Services NV

Funding
  • EU FP 7: € 2.7 million
  • Partners: € 0.9 million
Industry Follow Group

Members:

<table>
<thead>
<tr>
<th>Océ</th>
<th>Dutch Space</th>
<th>Vestas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airbus</td>
<td>ESA</td>
<td>Grundfors</td>
</tr>
<tr>
<td>Nokia</td>
<td>FKI Logistex</td>
<td>Volvo</td>
</tr>
<tr>
<td>Siemens</td>
<td>Darwin</td>
<td>Bang &amp; Olufsen</td>
</tr>
<tr>
<td>Martin Group</td>
<td>ASML</td>
<td>MBDA</td>
</tr>
<tr>
<td>Atlas Copco</td>
<td>Assembleon</td>
<td>Terma</td>
</tr>
</tbody>
</table>

- Annual meetings
- Monitor the DESTECS project
- Give feedback and advice
- Present industrial challenges
- New companies are welcome to join
Embedded Systems Development

• Boderc project (2002-2006): Model based design of high-tech systems
• Design gaps between disciplines lead to errors in designs
• Many of these errors are detected too late: during prototyping

• Example: Paper path setup
• Paper jams for high speed paper handling
Embedded Systems Development

Model Based Design:
• Controller in discrete event domain
• System in continuous time domain

Co-simulation:
• Coupling disciplines
• Analysis on virtual prototype
Embedded Systems Development

Cause of the problems
- Geometry changes were not adequately communicated
- Errors in acceleration and deceleration paths

Results
- These errors can be detected in an early stage of the design through co-simulation
- Dependability can be assessed by fault injection


Vienna Development Method

• Origins in IBM’s Vienna Lab in the 1970s
• Based on discrete mathematics:
  • Simple, abstract data types
  • Invariants to restrict membership
  • Functional specification: implicit (pre/post) & explicit (functional or imperative)
• Three major dialects:
  • ISO Standard base language VDM-SL
  • VDM++ is an object-oriented extension
  • VDM-RT extends VDM++ for real-time distributed systems
• Industry strength tools:
  • Overture (open source) → https://sourceforge.net/projects/overture/
  • VDMTools → http://www.vdmtools.jp/en
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>

- Internal object state
- Definitions
- Dynamic behaviour
- Synchronization control
- Test automation support
Access Modifiers

• VDM++ Class Members may have their access specified as `public`, `private` or `protected`.
• The default for all members is `private`.
• Access modifiers may not be narrowed e.g. a subclass can not override a public operation in the superclass with a private operation in the subclass.
• `static` modifiers can be used for definitions which are independent of the object state.
Constructors

• Each class can have a number of constructors
• Syntax identical to operations with a reference to the class name in return type
• The return does not need to be made explicitly
• Can be invoked when a new instance of a class gets created
Instance Variables

- Used to model persistent state in instances
- Always typed and can be initialised
- Consistency properties modelled as invariants

class Person
types
  string = seq of char
instance variables
  name: string := [];
  employer: set of Company
  age: int := 0;
  inv 0 <= age and age <= 99;
end Person

class Company
  ...
end Company
Type Definitions

• Basic types
  • Boolean
  • Numeric
  • Tokens
  • Characters
  • Quotations

• Compound types
  • Set types
  • Sequence types
  • Map types
  • Product types
  • Composite types
  • Union types
  • Optional types
  • Function types

Invariants can be added
## Boolean

<table>
<thead>
<tr>
<th>Expression</th>
<th>Operation</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not b</code></td>
<td>Negation</td>
<td><code>bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a and b</code></td>
<td>Conjunction</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a or b</code></td>
<td>Disjunction</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a =&gt; b</code></td>
<td>Implication</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a &lt;=&gt; b</code></td>
<td>Biimplication</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a = b</code></td>
<td>Equality</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
<tr>
<td><code>a &lt;&gt; b</code></td>
<td>Inequality</td>
<td><code>bool * bool -&gt; bool</code></td>
</tr>
</tbody>
</table>
### Numeric (1)

- **Unary minus**
  - `-x`
  - Type: `real -> real`

- **Absolute value**
  - `abs x`
  - Type: `real -> real`

- **Floor**
  - `floor x`
  - Type: `real -> int`

- **Sum**
  - `x + y`
  - Type: `real * real -> real`

- **Difference**
  - `x - y`
  - Type: `real * real -> real`

- **Product**
  - `x * y`
  - Type: `real * real -> real`

- **Division**
  - `x / y`
  - Type: `real * real -> real`

- **Integer division**
  - `x div y`
  - Type: `int * int -> int`

- **Remainder**
  - `x rem y`
  - Type: `int * int -> int`

- **Modulus**
  - `x mod y`
  - Type: `int * int -> int`

- **Power**
  - `x ** y`
  - Type: `real * real -> real`
### Numeric (2)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt; y$</td>
<td>Less than</td>
<td>real * real -&gt; bool</td>
</tr>
<tr>
<td>$x &gt; y$</td>
<td>Greater than</td>
<td>real * real -&gt; bool</td>
</tr>
<tr>
<td>$x \leq y$</td>
<td>Less or equal</td>
<td>real * real -&gt; bool</td>
</tr>
<tr>
<td>$x \geq y$</td>
<td>Greater or equal</td>
<td>real * real -&gt; bool</td>
</tr>
<tr>
<td>$x = y$</td>
<td>Equal</td>
<td>real * real -&gt; bool</td>
</tr>
<tr>
<td>$x &lt;&gt; y$</td>
<td>Not equal</td>
<td>real * real -&gt; bool</td>
</tr>
</tbody>
</table>
Product and Record Types

• Product type definition:
  \[A_1 \times A_2 \times \ldots \times A_n\]
  
  Construction of a tuple:
  \[\text{mk}_a(a_1, a_2, \ldots, a_n)\]

• Record type definition:
  \[A :: \text{selfirst} : A_1\]
  \[\text{selsec} : A_2\]
  \[\ldots\]
  \[\text{seln} : A_n\]
  
  Construction of a record:
  \[\text{mk}_A(a_1, a_2, \ldots, a_n)\]
Invariants

Even = \texttt{nat}

\texttt{inv \ n == n \ mod \ 2 = 0}

SpecialPair = \texttt{nat * real} – the first is smallest

\texttt{inv \ mk_{(n,r)} == n < r}

DisjointSets = \texttt{set of set of A}

\texttt{inv \ ss == forall \ s1, s2 \ in \ set \ ss \ &}

\hspace{1cm} \texttt{s1 <> s2 => s1 \ inter \ s2 = {}}
Function Definitions (1)

• Explicit functions:
  \[ f: A \times B \times \ldots \times Z \rightarrow R_1 \times R_2 \times \ldots \times R_n \]
  \[ f(a, b, \ldots, z) == \]
  \[ \text{expr} \]
  \[ \text{pre \ preexpr(a, b, \ldots, z)} \]
  \[ \text{post \ postexpr(a, b, \ldots, z, RESULT)} \]

• Implicit functions:
  \[ f(a:A, b:B, \ldots, z:Z) \ r_1:R_1, \ldots, \ r_n:R_n \]
  \[ \text{pre \ preexpr(a, b, \ldots, z)} \]
  \[ \text{post \ postexpr(a, b, \ldots, z, r_1, \ldots, r_n)} \]

Implicit functions cannot be executed by the VDM interpreter.
Function Definitions (2)

• Extended explicit functions:
  \[ f(a:A, b:B, \ldots, z:Z) \ r1:R1, \ldots, \ rn:Rn \]  
  \[ \text{expr} \]  
  \[ \text{pre} \ \text{preexpr}(a,b,\ldots,z) \]  
  \[ \text{post} \ \text{postexpr}(a,b,\ldots,z,r1,\ldots,\ rn) \]  

Extended explicit functions are a non-standard combination of the implicit colon style with an explicit body.

• Preliminary explicit functions:
  \[ f: A \ast B \ast \ldots \ast Z \rightarrow R1 \ast R2 \ast \ldots \ast Rn \]  
  \[ f(a,b,\ldots,z) \]  
  \[ \text{is not yet specified} \]  
  \[ \text{pre} \ \text{preexpr}(a,b,\ldots,z) \]  
  \[ \text{post} \ \text{postexpr}(a,b,\ldots,z,\text{RESULT}) \]
Quoting pre- and post-conditions

Given an implicit function definition like:

```plaintext
ImplFn(n,m: nat, b: bool) r: nat
pre  n < m
post if b then n = r else r = m
```

Two extra functions which can be used elsewhere are automatically created:

```plaintext
pre_ImplFn: nat * nat * bool -> bool
pre_ImplFn(n,m,b) ==
    n < m;

post_ImplFn: nat * nat * bool * nat -> bool
post_ImplFn(n,m,b,r) ==
    if b
    then n = r
    else r = m
```
Recursive Functions and Pre-conditions

A recursive function definition could look like:

```plaintext
fac: nat -> nat
fac (n) ==
  if n > 1
  then n * fac(n-1)
  else 1
measure Id;
Id: nat -> nat
Id(a) == a
```

Pre-conditions can also be used:

```plaintext
Division: real * real -> real
Division(p,q) ==
  p/q
pre q <> 0
```
Expressions

- Let and let-be expressions
- If-then-else expressions
- Cases expressions
- Quantified expressions
- Set expressions
- Sequence expressions
- Map expressions
- Tuple expressions
- Record expressions
- Is expressions

Special VDM++ Expressions

- New and Self expressions
- Class membership expressions
- Object comparison expressions
- Object reference expressions
Example Let Expressions

• Let expressions are used for naming complex subexpressions:
  ```
  let d = b ** 2 - 4 * a * c
  in
  mk_(((-b - sqrt(d))/2a,(-b + sqrt(d))/2a)
  ```

• Let expressions can also be used for breaking down complex data structures into components:
  ```
  let mk_Report(tel,-,ov) = rep
  in
  sub-expr
  ```
Example Let-be expressions

- Let-be-such-that expressions are even more powerful. A free choice can be expressed:

  ```
  let i in set inds l be st Largest(elems l, l(i))
  in
      sub_expr
  ```

  and

  ```
  let l in set Permutations(list) be st
      forall i,j in set inds l & i < j => l(i) <= l(j)
  in l
  ```
If-then-else Expressions

If-then-else expressions are similar to those known from programming languages.

\[
\begin{align*}
\text{if } c \text{ in set dom } rq \\
\text{then } rq(c) \\
\text{else } {} \\
\end{align*}
\]

and

\[
\begin{align*}
\text{if } i = 0 \\
\text{then } <\text{Zero}> \\
\text{elseif } 1 \leq i \text{ and } i \leq 9 \\
\text{then } <\text{Digit}> \\
\text{else } <\text{Number}> \\
\end{align*}
\]
Cases Expressions

Cases expressions are very powerful because of pattern matching:

```plaintext
cases com:
    mk_Loan(a,b) -> a" has borrowed "^b,
    mk_Receive(a,b) -> a" has returned "^b,
    mk_Status(l) -> l" are borrowing "^Borrows(l),
    others -> "some other command is used"
end
```

and

```plaintext
cases a:
    mk_A(a',-,a') -> Expr(a'),
    mk_A(b,b,c) -> Expr2(b,c)
end
```
Set Expressions

• Set enumeration:
  \{a,3,3,\text{true}\}

• Set comprehension can either use set binding:
  \{a+2 \mid \text{mk}_{}(a,a) \ \text{in set} \ \{\text{mk}_{}(\text{true},1),\text{mk}_{}(1,1)\}\}

  or type binding:
  \{a \mid a: \text{nat} \ & a<10\}

• Set range expression:
  \{3,\ldots,10\}
Sequence Expressions

- Sequence enumeration:
  \[ [7.7, \text{true}, "I", \text{true}] \]
- Sequence comprehension can only use a set bind with numeric values (numeric order is used):
  \[ [i \times i \mid i \text{ in set } \{1, 2, 4, 6\}] \]
  and
  \[ [i \mid i \text{ in set } \{6, 3, 2, 7\} \& i \mod 2 = 0] \]
- Subsequence expression:
  \[ [4, \text{true}, "string", 9, 4](2, \ldots, 4) \]
Map Expressions

• Map enumeration:
  \{1|->true, 7|->6\}

• Map comprehension can either use type binding:
  \{i|->mk_(i, true) | i: bool\}

or set binding:
  \{a+b|->b-a | a in set \{1,2\},
   b in set \{3,6\}\}

and
  \{i|->i | i in set \{1,...,10\} &
   i mod 3 = 0\}

One must be careful to ensure that every domain element maps uniquely to one range element.
Tuple Expressions

• A tuple expression looks like:
  \[ \text{mk\_}(2, 7, \text{true}, \{ \mid \rightarrow \}) \]

• Remember that tuple values from a tuple type will always
  • have the same length and
  • use the same types (possible union types) at corresponding positions.

• On the other hand the length of a sequence value may vary but the elements of the sequence will always be of the same type.
Record Expression

Given two type definitions like:

A :: n: nat
    b: bool
    s: set of nat;
B :: n: nat
    r: real

one can write expressions like:

mk_A(1, true, {8})
mk_B(3, 3)

\textbf{mu} (mk_A(7, false, {1, 4}), n|->1, s|->{})

\textbf{mu} (mk_B(3, 4), r|->5.5)

The \textbf{mu} operator is called “the record modifier”.
Apply Expressions

• Map applications:
  ```
  let m = {true|->5, 6|->{}}
  in m(true)
  ```

• Sequence applications:
  ```
  [2,7,true](2)
  ```

• Field select expressions:
  ```
  let r = mk_A(2,false,{6,9})
  in r.b
  ```
Is Expressions

Basic values and record values can be tested by is- expressions.

\[ \text{is\_nat}(5) \] will yield true.

\[ \text{is\_C(mk\_C(5))} \] will also yield true, given that C is defined as a record type having one component which 5 belongs to.

\[ \text{is\_A(mk\_B(3,7))} \] will always yield false.

\[ \text{is\_A}(6) \] will also always yield false.
New and Self Expressions

- The new expression creates an instance of a class and yields a reference to it.
- Given a class called $C$ this will create an instance of $C$ and return its reference:
  ```
  new $C()$
  ```
- The `self` expression yields the reference of an object.
- Given a class with instance variable $a$ of type $nat$ this will initialize an object and implicitly yield its reference (always for constructors):
  ```
  C: nat ==> C
  C(n) ==
  a := n;
  ```
Class Membership Expressions

Check if an object is of a particular class.

\textbf{isofclass}(\textit{Class	extunderscore name, object	extunderscore ref})

Returns true if \textit{object	extunderscore ref} is of class \textit{Class	extunderscore name} or a subclass of \textit{Class	extunderscore name}.

Check for the baseclass of a given object.

\textbf{isofbaseclass}(\textit{Class	extunderscore name, object	extunderscore ref})

For the result to be true, \textit{object	extunderscore ref} must be of class \textit{Class	extunderscore name}, and \textit{Class	extunderscore name} cannot have any superclasses.
Object Comparison Expressions

Compare two objects.

\texttt{sameclass} (\texttt{obj1}, \texttt{obj2})

True if and only if \texttt{obj1} and \texttt{obj2} are instances of the same class

Comparison of baseclasses of two objects.

\texttt{samebaseclass} (\texttt{obj1}, \texttt{obj2})
Object Reference Expressions

• The = and <> operators perform comparison of object references.
• = will only yield true, if the two objects are in fact the same instance.
• <> will yield true, if the two objects are not the same instance, even if they have the same values in all instance variables.
Patterns and Pattern Matching

- Patterns are empty shells
- Patterns are matched thereby binding the pattern identifiers
- There are special patterns for
  - Basic values
  - Pattern identifiers
  - Don’t care patterns
  - Sets
  - Sequences
  - Tuples
  - Records
  but not for maps
Bindings

• A binding matches a pattern to a value.

• A set binding:
  \[ \text{pat in set expr} \]
  where \( expr \) must denote a set expression.
  \( pat \) is bound to the elements of the set \( expr \).

• A type binding:
  \[ \text{pat : type} \]
  Here \( pat \) is bound to the elements of \( type \).
  Type bindings can only be executed by the interpreter, if the type statically finite.
Operation Definitions (1)

- Explicit operation definitions:
  \[ o : A \times B \times \ldots \Rightarrow R \]
  \[ o(a,b,\ldots) = \]
  \[
  \text{stmt}\
  \text{pre expr}\
  \text{post expr}
  \]

- Implicit operations definitions:
  \[ o(a:A, b:B,\ldots) \Rightarrow r:R \]
  \[
  \text{ext rd \ldots}\n  \text{wr \ldots}\n  \text{pre expr}\n  \text{post expr}
  \]
Operation Definitions (2)

• Preliminary operation definitions:
  \( o: A \ast B \ast \ldots \implies R \)
  \( o(a,b,\ldots) = \)
  \text{is not yet specified}
  \( \text{pre expr} \)
  \( \text{post expr} \)

• Delegated operation definitions:
  \( o: A \ast B \ast \ldots \implies R \)
  \( o(a,b,\ldots) = \)
  \text{is subclass responsibility}
  \( \text{pre expr} \)
  \( \text{post expr} \)
Operation Definitions (3)

• Operations in VDM++ can be overloaded
• Different definitions of operations with same name
• Argument types must not be overlapping statically (structural equivalence omitting invariants)
Statements

- Let and Let-be statements
- Block statements
- Assign statements
- Conditional statements
- For loop statements
- While loop statements
- Call Statements

- Return statements
- Exception handling statements
- Error statements
- Identity statements
Let and Let-be Statements

Let statements are used for naming complicated constructs and for decomposing complex structures

let cs' = {c |-> cs(c) union {s}},
ct' = {s |-> ct(s) union {c}}
in sub_stmt1

Let-be-such-that statements are even more powerful. A free choice can be expressed:

let i in set inds l be st Largest (elems l, l(i))
in sub_stmt2

and

let l in set Permutations(list) be st
  forall i,j in set inds l & i < j => l(i) <= l(j)
in return l
Block Statements

• The *block statement* corresponds to block statements from traditional high-level programming languages.
• It can contain local variables, which must be typed and can be initialised.
  
  (dcl a: nat := 5;
   dcl b: bool;
   stmt1;
   stmt2;
   ...
   stmtn)

• If *stmt1* returns a value the execution of the block is terminated and this value is returned as the result of the entire block.
Assign Statements

• The *assign statement* is a generalization of assignment statements from traditional high-level programming languages. It is used to change the value of a global or local state.

• Assume that $x$ is a global or local state component of type *nat*, then it can be assigned a value:
  
  \[ x := 5 \]

• More complex assignments exist. Assume $x$ is of record type with a field $s$ which is a map:
  
  \[ x.s(3) := \text{true} \]
If-then-else Statements

If-then-else statements are similar to those known from programming languages.

```
if c in set dom rq
then  reqs := rq(c)
else return {}
```

and

```
if i = 0
then return <Zero>
elseif 1 <= i and i <= 9
then return <Digit>
else return <Number>
```

Notice that the consequence and alternative parts are statements.
Cases Statements

Cases statements are very powerful because of pattern matching:

```plaintext
cases com:
    mk_Loan(a, b) -> return a^^ has borrowed ^b,
    mk_Receive(a, b) -> return a^^ has returned ^b,
    mk_Status(l) -> return l^^ are borrowing ^Borrows(l),
    others -> return "some other command is used"
end

and

cases a:
    mk_A(a',-,a') -> Expr(a'),
    mk_A(b.b.c) -> Expr2(b,c)
end
```

Notice that the bodies of the alternatives are statements.
For Loop Statements

• Index for loop statement:

```plaintext
for id = lower to upper by step 
do stmt
```

• Set for loop statement:

```plaintext
for all pat in set setexpr 
do stmt
```

• Sequence for loop statement:

```plaintext
for pat in seqexpr 
do stmt
```
While Loop Statements

• While loop statement:

\[
\text{while expr do stmt}
\]

• \textit{expr} is recalculated at the termination of each evaluation of the body statement \textit{stmt}.
Return Statements

The *return statement* returns the value of an expression inside an operation:

```
return expr
```

or simply

```
return
```

Note that if a value is returned by either a call statement or a return statement the execution of the operation is terminated. Thus, succeeding statements will not be executed.
Exception Handling Statements

• The exit statement raises an exception:
  
  \[ \text{exit expr} \]

  \textit{expr} is optional.

• Always statement:

  \[(\text{dcl mem: Memory;}
      \textbf{always Free(mem) in}
      (mem := \text{Allocate}();
       \text{Command}(mem, \ldots)))\]
Exception Handling Statements

• Trap statement:
  
  ```
  trap pat with ErrorAction(pat) in 
  (dcl mem: Memory; 
    always Free(mem) in 
    (mem := Allocate(); 
     Command(mem, ...)))
  ```

• Recursive trap statement:
  
  ```
  tixe { (<NOMEM>) |-> return -1, 
    (<BUSY>) |-> DoCommand(), 
    err |-> return -2 } 
  in DoCommand() 
  ```
Error and Identity Statements

• The *error statement* states that the result of a statement is undefined and therefore an error has occurred.

```plaintext
if a = <OK>
  then DoSomething()
else error
```

• The *identity statement* is used to signal that no evaluation takes place.

```plaintext
if a <> []
  then str := str ^ a
else skip
```
Concurrency Primitives in VDM++

• Concurrency in VDM++ is based on threads

• Threads communicate using shared objects

• Synchronization on shared objects is specified using permission predicates
Threads

- Modelled by a class with a thread section

```ruby
class SimpleThread
  thread
    IO.print("Hello World")
  end
end SimpleThread
```

- Thread execution begins using start statement with an instance of a class with a thread definition

```ruby
start (new SimpleThread())
```

- Note that execution ends when debug thread is finished
Permission Predicates

• Permission predicates are described in the sync section of a class

  \texttt{sync}
  \texttt{per \ <operation\ name> => predicate}

• The predicate may refer to the class’s instance variables.

• The predicate may also refer to special variables known as \textit{history counters}.
## History Counters

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#req op</td>
<td>The number of times that op has been requested</td>
</tr>
<tr>
<td>#act op</td>
<td>The number of times that op has been activated</td>
</tr>
<tr>
<td>#fin op</td>
<td>The number of times that op has been completed</td>
</tr>
<tr>
<td>#active op</td>
<td>The number of active executions of op</td>
</tr>
<tr>
<td>#waiting op</td>
<td>The number of waiting executions of op</td>
</tr>
</tbody>
</table>

\[
\#\text{active} \ op = \#\text{act} \ op - \#\text{fin} \ op
\]

\[
\#\text{waiting} \ op = \#\text{req} \ op - \#\text{act} \ op
\]
Mutual Exclusion

- VDM++ provides the keyword mutex
  - `mutex(Put, Get)`
- Shorthand for
  - `per Put => #active(Get) = 0`
  - `per Get => #active(Put) = 0`
Concurrent Real-Time and Distributed Design Model

• Timing built in:
  • New keyword: `time`
  • Use of default durations
  • Use of duration and cycles statements

• Setting task switching overhead

• Typical Design Structure
  • `SystemName` is now turned into a system
  • `CPU’s and BUS’es are introduced inside SystemName`
  • `Environment` may be turned into a system
  • Some operations are made asynchronous (`async`)
  • Some `Step` like threads are made `periodic`
Periodic Threads

- **periodic** \((\text{period}, \text{jitter}, \text{delay}, \text{offset}) (\text{Op})\)
- **Period**: This is a non-negative, non-zero value that describes the length of the time interval between two adjacent events in a strictly periodic event stream.
- **Jitter**: This is a non-negative value that describes the amount of time variance that is allowed around a single event.
- **Delay**: This is a non-negative value smaller than the period which is used to denote the minimum inter arrival distance between two adjacent events.
- **Offset**: This is a non-negative value which is used to denote the absolute time value at which the first period of the event stream starts.
Overture Perspective

Project explorer with VDM model files

VDM Editors

Changing perspective

Outline of VDM model

Errors and warnings
Debug Perspective

- Call traces in debug
- Inspecting variables
- Editor
- Interactive console
- Outline
The VDMJ Interpreter

• Able to interpret VDM-SL, VDM++ and VDM-RT dialects
• Options for checking dynamic properties (inv, pre, post)
• Can collect test coverage information
• Deterministic interpretation even with
  • Multi-threading
  • Multi-CPU simulation
• Facility for defining libraries (IO, Math, etc.)
• Able to combine with GUI in Java using jar file
Real-Time Log View
Mapping between VDM++ and UML
Combinatorial Testing Perspective

Operations

```
op1 = (0 == 1) = set f Plant \ Period * Expert
Run() =>
let periods = plant.ExpertIsOnDuty(ex1),
    expert = plant.ExpertToPage(a1,p1)
in
return mk_(periods,expert);
```

Traces

```
AddingAndDeleting: let myex in set exs
    let myex2 in set exs \ {myex}
in
    let p in set ps
    let x = p in set ps

plant.AddExpertToSchedule(p,myex);
plant.AddExpertToSchedule(p,myex2);
plant.RemoveExpertFromSchedule(p,myex);
plant.RemoveExpertFromSchedule(p,myex2);
```

Overview of results

Detailed test case and results
Combinatorial Testing Overview

- VDM source
- Parser AST
- Pattern tree
- Filtered test results
- Shape reduced tests
- Fully expanded tests
Test case execution

- Test cases
- Test engine
  - Search test cases and mark test cases with same prefix
  - Check if filtered by other test cases
- Test results
  - Passed
  - Inconclusive
  - Failed
  - Filtered
Proof Obligation Perspective

Proof obligation view

(let expert:Expert = RESULT in
  p in set dom schedule)
Proof Obligation Generation

• Used to identify all places where run-time errors could occur
• Different types (application, satisfiability, invariant, …)
• Simple POs can automatically be verified
• Can also be used for termination of recursive functions
• A measure needs to be defined for these
• Interpreter can also check these dynamically
Prototype Verification Support

- Not yet incorporated into the Eclipse GUI
- Making use of the HOL theorem prover
- Will probably have to be redone so it can be incorporated into Eclipse

VDM++ Model → HOL4 Model → Consistency Proof

VDM++ POs
Modelling & Simulation

Model

Abstract
- Competent if detailed enough for analysis

Variables

Design Parameters fixed per run

Model Interface

Script

Runs a simulation
- Initialises variables and design parameters
- Forces selections and external updates, e.g. set point

Faults - errors - failures
- Fault Modelling: including error states & faulty functionality in the model
- Fault Injection during a simulation managed by script
Co-modelling & Co-simulation

Co-model Interface

DE Model → Contract → CT Model

Shared
- design parameters
- variables
- events

Script
Example: Watertank

- Continuous input flow
- Valve operated by controller
- Water level sensor
- Water level should be between minimum and maximum level
Model

Continuous-time model
• Model with hydraulic circuit elements.
• Waterlevel sensor triggers events: upper level reached, lower level reached.

Discrete-time model
• Event-driven action: open valve when upper level reached, close valve when lower level reached.
Co-Simulation Engine

- Run smallest possible steps until output changes
- Run until an event occurs
- Hold if input changes
Co-Simulation Benefits

- Variables and timing information: run co-simulations
- Design parameters: shared knowledge
- Fault injection: testing dependability
- Scenarios: testing fault tolerance
Ideal World

- We can find a model that is a perfect representation of the real system
- Using this virtual machine, we can design the perfect controller that will never fail
Dependability

- The real world has lots of uncertainties, caused by human behavior, failure of system and things we do not know.

  *We want to design systems that can handle these uncertainties: dependable systems*

- Practice: In discrete code 90% of the code is used for making machines save and cope with uncertainties, 10% for actually running it.
Dependability

• The ability to deliver a service that can **justifiably** be trusted

**Attributes**
- Reliability
- Safety
- Availability
- Maintainability
- ...

**Means**
- Prevention
- Tolerance
- Removal
- Forecasting

**Threats**
- Faults
- Errors
- Failures

Avizienis et al: Fundamental concepts of Dependability
Dependability: faults, errors and failures

**Fault**: Adjudged or hypothesised cause of an error

**Error**: Deviation from intended or desired state

**Failure**: Delivered service deviates from correct service

*e.g.*

<table>
<thead>
<tr>
<th>Fault</th>
<th>Error</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>design flaws</td>
<td>dose mismatch</td>
<td>wrong dose given</td>
</tr>
</tbody>
</table>

Failure: Delivered service deviates from correct service

Error: Deviation from intended or desired state

Fault: Adjudged or hypothesised cause of an error
Many kinds of potential fault can be considered for an embedded system. They need to be incorporated into design models. One aim of DESTECs is to consider types of fault that we encounter and provide modelling patterns for them. … so we do not have to re-invent them every time.
Dependability: faults, errors and failures

- Start from models of “normal” behaviour
- Augment with models of faults
- Test against scenarios
- Redesign for fault avoidance/tolerance
- Do this in Environment, Controller and interface.

Tools to manage this exploration of the design space, maintaining models and managing evolution are essential!
Model-Driven Design Flow

- **Way of Working**
  - Abstraction
    - Hierarchy
      - Split into Subsystems
      - Cope with complexity
  - Model-driven design
    - Design Space Exploration
      - Aspect models
      - Make choices
      - Limit solution space
    - Step-wise refinement
      - Add detail
      - Lower abstraction
    - Implementation
    - Realization
- **Concurrent design trajectory**
- **Early** Integration where possible
Abstraction

- Abstraction
- Top-level split-up
  - Components
  - Concurrent design
  - Disciplines, people, departments

Interactions ⇔ cross cutting concerns
Summary

DESTECS will:
• go beyond the state of the art in model based design
• combine disciplines through co-simulation
• use co-simulation to share knowledge between disciplines
• support design space exploration

This allows:
• The use of standard tools for each discipline
• error detection in early stages of a project
• an early validation of system dependability

Conditions
• The project will restrict to the discrete-event domain and the continuous-time domain disciplines
• An Industrial Follow Group will monitor the results