VDM++ Tutorial at FM'06

Professor Peter Gorm Larsen
Engineering College of Aarhus
Computer Technology & Embedded Systems
(pgl@iha.dk)

Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
  - Statements
  - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

Who gives this tutorial?

- Peter Gorm Larsen; MSc, PhD
- 18 years of professional experience
  - ½ year with Technical University of Denmark
  - 13 years with IFAD
  - 3,5 years with Systematic
  - 3/4 year with University College of Aarhus
- Consultant for most large defence contractors on large complex projects (e.g. JSF)
- Relations to industry and academia all over the world
- Has written books and articles about VDM
- See http://home0.inet.tele.dk/pgl/peter.htm for details
Vienna Development Method

- Invented at IBM's labs in Vienna in the 70's
- VDM-SL and VDM++
  - ISO Standardisation of VDM-SL
  - VDM++ is an object-oriented extension
- Model-oriented specification:
  - Simple, abstract data types
  - Invariants to restrict membership
- Functional specification:
  - Referentially transparent functions
  - Operations with side effects on state variables
  - Implicit specification (pre/post)
  - Explicit specification (functional or imperative)

Where has VDM++ been used?

- Modeling critical computer systems e.g. for industries such as
  - Avionics
  - Railways
  - Automotive
  - Nuclear
  - Defense
- I have used this industrially for example at:
  - Boeing, Lockheed-Martin (USA)
  - British Aerospace, Rolls Royce, Adelard (UK)
  - Matra, Dassault, Aerospatiale (France)
  -…

Industrially Inspired Examples

- Chemical Plant Alarm Management System
- A Robot Controller
- A Road Congestion Warning System
Validation Techniques

- **Inspection**: organized process of examining the model alongside domain experts.
- **Static Analysis**: automatic checks of syntax & type correctness, detect unusual features.
- **Testing**: run the model and check outcomes against expectations.
- **Model Checking**: search the state space to find states that violate the properties we are checking.
- **Proof**: use a logic to reason symbolically about whole classes of states at once.

Validation via Animation

Execution of the model through an interface. The interface can be coded in a programming language of choice so long as a dynamic link facility (e.g., CORBA) exists for linking the interface code to the model.

Formal model

Interpreter

Interface

C++ or Java

Interface Code

Testing can increase confidence, but is only as good as the test set. Exhaustive techniques could give greater confidence.

Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
  - Statements
  - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples
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 cannot 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.
Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
    - Types
    - Functions
    - Expressions, Patterns, Bindings
    - Operations
    - Statements
    - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

Instance Variables (1)

- Used to model attributes
- Consistency properties modelled as invariants

```vdm
class Person
  types
    string = seq of char
  instance variables
    name: string := "";
    age: int := 0;
  inv
    0 <= age and age <= 99;
end Person
```

Instance Variables (2)

- Used to model associations
- Object reference type simply written as the class name, e.g. Person
- Multiplicity using VDM data types

```vdm
class Person
  ... 
  instance variables
    name: string := "";
    age: int := 0;
    employer: set of Company
  ... 
end Person
```

```vdm
class Company
  ... 
end Company
```
Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
    - Functions
    - Expressions, Patterns, Bindings
    - Operations
    - Statements
    - Concurrence
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

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

- 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  Bimplication  bool * bool -> bool
- a = b  Equality  bool * bool -> bool
- a <> b  Inequality  bool * bool -> bool

Quantified expressions can also be considered to be basic operators but we will present them together with the other general expressions.
**Numeric (1)**

- $-x$  Unary minus  real $\rightarrow$ real
- $\text{abs} \ x$  Absolute value  real $\rightarrow$ real
- $\text{floor} \ x$  Floor  real $\rightarrow$ int
- $x + y$  Sum  real $\ast$ real $\rightarrow$ real
- $x - y$  Difference  real $\ast$ real $\rightarrow$ real
- $x \ast y$  Product  real $\ast$ real $\rightarrow$ real
- $x / y$  Division  real $\ast$ real $\rightarrow$ real
- $x \div y$  Integer division  int $\ast$ int $\rightarrow$ int
- $x \mod y$  Remainder  int $\ast$ int $\rightarrow$ int
- $x \mod y$  Modulus  int $\ast$ int $\rightarrow$ int
- $x \ast y$  Power  real $\ast$ real $\rightarrow$ real

**Numeric (2)**

- $x < y$  Less than  real $\ast$ real $\rightarrow$ bool
- $x > y$  Greater than  real $\ast$ real $\rightarrow$ bool
- $x \leq y$  Less or equal  real $\ast$ real $\rightarrow$ bool
- $x \geq y$  Greater or equal  real $\ast$ real $\rightarrow$ bool
- $x = y$  Equal  real $\ast$ real $\rightarrow$ bool
- $x \neq y$  Not equal  real $\ast$ real $\rightarrow$ bool

**Product and Record Types**

- **Product type definition:**
  \[
  A_1 \ast A_2 \ast \ldots \ast A_n
  \]
  Construction of a tuple:
  \[
  \text{mk}_{(a_1, a_2, \ldots, a_n)}
  \]
- **Record type definition:**
  \[
  A :: \text{selfirst} : A_1
  \quad \text{selsec} : A_2
  \quad \ldots
  \quad \text{seln} : A_n
  \]
  Construction of a record:
  \[
  \text{mk}_{a}(a_1, a_2, \ldots, a_n)
Example Record Definition

A record type could be defined as:

```
Address ::
    house : HouseNumber
    street : Street
    town : PostalTown
```

With field selectors:

```
mk_Address(15, "The Grove", <London>).street
```

Example Tuple Definition

A tuple type could type could be defined as:

```
nat1 * (seq of char) * PostalTown
```

Then fields can be used using the .# operator:

```
mk_(12, "Abstraction Avenue", <Manchester>).#2
```

Overview of Set Operators

- in set s1: Membership (∈) \( A \) * set of \( A \) -> bool
- not in set s1: Not membership (∉) \( A \) * set of \( A \) -> bool
- union s1 \( s2 \): Union (\( \cup \)) set of \( A \) * set of \( A \) -> set of \( A \)
- s1 \( \setminus \) s2: Intersection (\( \cap \)) set of \( A \) * set of \( A \) -> set of \( A \)
- s1 \subseteq s2: Difference (\( \setminus \)) set of \( A \) * set of \( A \) -> set of \( A \)
- s1 \( \subseteq \) s2: Subset (\( \subseteq \)) set of \( A \) * set of \( A \) -> bool
- s1 \( \subset \) s2: Proper subset (\( \subset \)) set of \( A \) * set of \( A \) -> bool
- s1 \( = \) s2: Equality (\( = \)) set of \( A \) * set of \( A \) -> bool
- s1 \( \neq \) s2: Inequality (\( \neq \)) set of \( A \) * set of \( A \) -> bool
- card s1: Cardinality set of \( A \) -> nat
- dunion s1: Distr. Union (\( \cup \)) set of set of \( A \) -> set of \( A \)
- dinter s1: Distr. Intersection (\( \cap \)) set of set of \( A \) -> set of \( A \)
- power s1: Finite power set (\( P \)) set of \( A \) -> set of set of \( A \)
Set Comprehensions

- Using predicates to define sets implicitly
- In VDM++ formulated like:
  - \( \{ \text{element} \mid \text{list of bindings} \& \text{predicate} \} \)
- The predicate part is optional
- Quick examples:
  - \( \{3 \cdot x \mid x : \text{nat} \& x < 3\} \)
  - \( \{x \mid x : \text{nat} \& x < 5\} \)

Sequence Operators

- \text{seq1 of A} \rightarrow A
- \text{seq1 of A} \rightarrow \text{seq of A}
- seq of A \rightarrow \text{set of A}
- \text{seq of A} \rightarrow \text{set of nat1}
- \text{seq of A} \rightarrow \text{seq of A}
- \text{seq of A} \rightarrow \text{seq of A}
- \text{seq of A} \rightarrow \text{set of A}
- \text{seq of A} \rightarrow \text{seq of A}
- \text{seq of A} \rightarrow \text{bool}
- \text{seq of A} \rightarrow \text{bool}

Sequence Comprehensions

- Using predicates to define sequences implicitly
- In VDM++ formulated like:
  - \( \{ \text{element} \mid \text{numeric set binding} \& \text{predicate} \} \)
- The predicate part is optional
- The numeric order of the binding is used to determine the order in the sequence
- The smallest number is taken to be the first index
- Quick examples:
  - \( \{3 \cdot x \mid x \in \text{set} \{0 \ldots 2\}\} \)
  - \( \{x \mid x \in \text{set} \{0 \ldots 4\} \& x > 2\} \)
Map Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom =</td>
<td>Domain (map A to B) -&gt; set of A</td>
</tr>
<tr>
<td>rng n</td>
<td>Range (map A to B) -&gt; set of B</td>
</tr>
<tr>
<td>m union n2</td>
<td>Merge (map A to B) * (map A to B) -&gt; (map A to B)</td>
</tr>
<tr>
<td>m1 ++ m2</td>
<td>Override (map A to B) * (map A to B) -&gt; (map A to B)</td>
</tr>
<tr>
<td>merge x</td>
<td>Distr. merge set of (map A to B) -&gt; map A to B</td>
</tr>
<tr>
<td>x &lt;: m</td>
<td>Dom. restr. to set of A * (map A to B) -&gt; map A to B</td>
</tr>
<tr>
<td>x &gt;=: m</td>
<td>Dom. restr. by set of A * (map A to B) -&gt; map A to B</td>
</tr>
<tr>
<td>m =&gt; x</td>
<td>Rng. restr. to set of A * (map A to B) -&gt; map A to B</td>
</tr>
<tr>
<td>m := x</td>
<td>Rng. restr. by set of A * (map A to B) -&gt; map A to B</td>
</tr>
<tr>
<td>m(d)</td>
<td>Map apply (map A to B) * A -&gt; B</td>
</tr>
<tr>
<td>inverse m</td>
<td>Map inverse lmap A to B -&gt; lmap B to A</td>
</tr>
<tr>
<td>m1 = m2</td>
<td>Equality (map A to B) * (map A to B) -&gt; bool</td>
</tr>
<tr>
<td>m1 &lt;&gt; m2</td>
<td>Inequality (map A to B) * (map A to B) -&gt; bool</td>
</tr>
</tbody>
</table>

Mapping Comprehensions

- Using predicates to define mappings implicitly
- In VDM++ formulated like:
  - \{ maplet | list of bindings & predicate \}
- The predicate part is optional
- Quick examples
  - \{ i |-> i*i | i: nat1 & i <= 4 \}
  - \{ i**2 |-> i/2 | i in set {1,...,5} \}

Invariants

Even = nat
inv n n mod 2 = 0

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

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

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
  - Statements
  - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

Function Definitions (1)

- Explicit functions:
  
  \[
  f: A \times B \times \ldots \times Z \rightarrow R_1 \times R_2 \times \ldots \times R_n \\
  f(a, b, \ldots, z) == expr \\
  \hspace{1cm} \text{pre preexpr}(a, b, \ldots, z) \\
  \hspace{1cm} \text{post postexpr}(a, b, \ldots, z, \text{RESULT})
  \]

- Implicit functions:
  
  \[
  f(a:A, b:B, \ldots, z:Z) r_1:R_1, \ldots, r_n:R_n \\
  \hspace{1cm} \text{pre preexpr}(a, b, \ldots, z) \\
  \hspace{1cm} \text{post postexpr}(a, b, \ldots, z, r_1, \ldots, r_n)
  \]

Implicit functions cannot be executed by the VDM interpreter.

Function Definitions (2)

- Extended explicit functions:
  
  \[
  f(a:A, b:B, \ldots, z:Z) r_1:R_1, \ldots, r_n:R_n == expr \\
  \hspace{1cm} \text{pre preexpr}(a, b, \ldots, z) \\
  \hspace{1cm} \text{post postexpr}(a, b, \ldots, z, r_1, \ldots, r_n)
  \]

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

- Preliminary explicit functions:
  
  \[
  f: A \times B \times \ldots \times Z \rightarrow R_1 \times R_2 \times \ldots \times R_n \\
  f(a, b, \ldots, z) == \text{is not yet specified} \\
  \hspace{1cm} \text{pre preexpr}(a, b, \ldots, z) \\
  \hspace{1cm} \text{post postexpr}(a, b, \ldots, z, \text{RESULT})
  \]
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,n,b) == n < m;
post_ImplFn: nat * nat * bool * nat -> bool
post_ImplFn(n,m,b,r) ==
  if b then n = r
  else r = m
```

Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
    - Operations
    - Statements
    - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

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
- Define expressions
- Lambda 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 1, 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.

```
if c in set dom rq
then rq(c)
else {}

and

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

Cases expressions are very powerful because of pattern matching:

```plaintext
case 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

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

Set Expressions

- Set enumeration: 
  \{(a,3,3,\texttt{true})\}
- Set comprehension can either use set binding:
  \{(a+2 | (a,n) in set \{mk_{\texttt{(true,1)},mk_{\texttt{(1,1)}}}\})
  \text{or type binding:}
  \{(a | a:nat & a<10)\}
- Set range expression:
  \{3,...,10\}

Sequence Expressions

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

- Map enumeration:
  \{1\mapsto \text{true}, 7\mapsto 6\}

- Map comprehension can either use type binding:
  \{(i\mapsto \text{mk}(i, \text{true}) | i: \text{bool})\}
  or set binding:
  \{(a+b)\mapsto b-a | a \text{ in set } \{1,2\}, b \text{ in set } \{3,6\}\}
  and
  \{(i\mapsto i | i \text{ in set } \{1,\ldots,10\} \& i \text{ 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}, \{\mapsto\})

- 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:

\[
\begin{align*}
\text{A} &:: n: \text{nat} \\
b &:: \text{bool} \\
s &:: \text{set of nat} \\
\text{B} &:: n: \text{nat} \\
r &:: \text{real}
\end{align*}
\]

one can write expressions like:

\[
\begin{align*}
\text{mk}_A(1, \text{true}, \{8\}) \\
\text{mk}_B(3, 3) \\
\mu \text{mk}_A(7, \text{false}, \{1,4\}, n\mapsto 1, s\mapsto \{\}) \\
\mu \text{mk}_B(3, 4, r\mapsto 5.5)
\end{align*}
\]

The \mu operator is called "the record modifier".
Apply Expressions

- **Map applications:**
  \[
  \text{let } \ m = \{ \text{true} \rightarrow 5, \ 6 \rightarrow \{\}\} \\
  \text{in } m(\text{true})
  \]

- **Sequence applications:**
  \[
  [2,7,\text{true}](2)
  \]

- **Field select expressions:**
  \[
  \text{let } \ r = \text{mk}_A(2,\text{false},\{6,9\}) \\
  \text{in } r.b
  \]

Is Expressions

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

\[
\begin{align*}
\text{is}_\text{nat}(5) & \quad \text{will yield true.} \\
\text{is}_C(\text{mk}_C(5)) & \quad \text{will also yield true, given that } C \text{ is defined as a record type having one component which 5 belongs to.} \\
\text{is}_A(\text{mk}_B(3,7)) & \quad \text{will always yield false.} \\
\text{is}_A(6) & \quad \text{will also always yield false.}
\end{align*}
\]

Define Expressions

The right-hand side of a define expression has access to the instance variables.

The state could be changed by an operation call:

\[
\text{def } a = \text{OpCall(arg1,arg2)} \text{ in } f(a)
\]

or parts of the state could simply be read:

\[
\text{def } a = \text{instance_variable in } g(a)
\]
Lambda Expressions

- Lambda expressions are an alternative way of defining explicit functions.
  \[ \text{lambda } n: \text{nat } & n \times n \]
- They can take a type bind list:
  \[ \text{lambda } a: \text{nat}, b: \text{bool } & \]
  \[ \text{if } b \text{ then } a \text{ else } 0 \]
- Or use more complex types:
  \[ \text{lambda } mk_{(a,b)}: \text{nat } \times \text{nat } & a + b \]

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:
  \[ \text{new } C() \]
- The self expression yields the reference of an object.
- Given a class with instance variable \( a \) of type \( \text{nat} \) this will initialize an object and yield its reference:
  Create: \( \text{nat} \Rightarrow C \)
  \[
  \text{Create } (n) == \\
  \{ a := n; \\
  \text{return self } \}
  \]

Class Membership Expressions

Check if an object is of a particular class.

\[ \text{isofclass} \{ \text{Class_name,object_ref} \} \]

Returns true if \( \text{object_ref} \) is of class \( \text{Class_name} \) or a subclass of \( \text{Class_name} \).

Check for the baseclass of a given object.

\[ \text{isofbaseclass} \{ \text{Class_name,object_ref} \} \]

For the result to be true, \( \text{object_ref} \) must be of class \( \text{Class_name} \), and \( \text{Class_name} \) cannot have any superclasses.
Object Comparison Expressions

Compare two objects.

\[
\text{sameclass}(\text{obj1}, \text{obj2})
\]

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

- \text{sameclass}(\text{m}, \text{new Manager}) = \text{true}

Comparison of baseclasses of two objects.

\[
\text{samebaseclass}(\text{obj1}, \text{obj2})
\]

- \text{samebaseclass}(\text{m}, \text{new Temporary}) = \text{false}

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. \( \text{pat} \) is bound to the elements of the set \( expr \).
- A type binding:
  \[ \text{pat : type} \]
  Here \( \text{pat} \) is bound to the elements of type. Type bindings cannot be executed by the interpreter, because such types can be infinitely large.

Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
    - Statements
    - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

Operation Definitions (1)

- Explicit operation definitions:
  \[ o : A \times B \times \ldots \implies R \]
  \[ o(a,b,\ldots) = \]
  \[ \text{stmt} \]
  \[ \text{pre} \ expr \]
  \[ \text{post} \ expr \]
- Implicit operations definitions:
  \[ o(a:A, b:B,\ldots) : R \]
  \[ \text{ext rd \ldots} \]
  \[ \text{wr \ldots} \]
  \[ \text{pre} \ expr \]
  \[ \text{post} \ expr \]
Operation Definitions (2)

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

• Delegated operation definitions:
  \[ o: A \times B \times \ldots \Rightarrow R \]
  \[ o(a,b,\ldots) == \text{is subclass responsibility} \]
  \[ \text{pre expr} \]
  \[ \text{post expr} \]

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)

Example Operation Definitions

An implicit operation definition could look like:

\[
\text{Withdraw(amount: nat) newBalance: int}
\]
\\ext \text{rd limit : int}
\\wr \text{balance : int}
\]
\[
\text{pre balance - amount > limit}
\]
\[
\text{post balance + amount = balance\ and newBalance = balance}
\]

An explicit operation definition could look like:

\[
\text{Withdraw: nat => int}
\]
\[
\text{Withdraw(amount) == ( balance := balance - amount; return balance )}
\]
\[
\text{pre balance - amount > limit}
\]
Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
  - Statements
  - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples

Statements

- Let and Let-be statements
- Define Statements
- Block statements
- Assign statements
- Conditional statements
- For loop statements
- While loop statements
- Call Statements
- Non deterministic statements
- Return statements
- Exception handling statements
- Error statements
- Identity statements

Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
  - Introduction
  - Access Modifiers and Constructors
  - Instance Variables
  - Types
  - Functions
  - Expressions, Patterns, Bindings
  - Operations
  - Statements
  - Concurrency
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples
Concurrent 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
  
  ```
  class SimpleThread
  thread
  let t = new IO().echo("Hello World!")
  end SimpleThread
  ```

  - Thread execution begins using start statement with an instance of a class with a thread definition
    
    ```
    start (new SimpleThread)
    ```

Thread Communication

- Threads operating in isolation have limited use.
- In VDM++, threads communicate using shared objects.
A Producer-Consumer Example

- Concurrent threads must be synchronized
- Illustrate with a producer-consumer example
- Produce before consumption...
- Assume a single producer and a single consumer
- Producer has a thread which repeatedly places data in a buffer
- Consumer has a thread which repeatedly fetches data from a buffer

The Producer Class

class Producer

instance variables

b : Buffer

operations

Produce() : seq of char
Produce() == ...

thread

while true do
b.Put(Produce())
end Producer

The Consumer Class

class Consumer

instance variables

b : Buffer

operations

Consume : seq of char => ()
Consume(d) == ...

thread

while true do
Consume(b.Get())
end Consumer
The Buffer Class

```vdm++
class Buffer
  instance variables
    data : [seq of char] := nil
  operations
    public Put : seq of char => ()
      data := newData;
    public Get : () => seq of char
      let oldData = data
      in
        ( data := nil;
          return oldData
        )
end Buffer
```

Permission Predicates

- What if the producer thread generates values faster than the consumer thread can consume them?
- Shared objects require *synchronization*.
- Synchronization 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.

```vdm++
per operation name => predicate;
```

- For Put and Get we could write:
  ```vdm++
  per Put => data = nil;
  per Get => data <> nil;
  ```
History Counters and mutex

<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>
</tbody>
</table>

- Mutual exclusion (mutex)
- Blocking Puts and Gets while executing:
  *mutex(Put,Get)*

Permission Predicates: Details

- Permission predicates are described in the sync section of a class:
  sync
  per <operation name> => predicate
- The predicate may refer to the class’s instance variables.
- The predicate may also refer to special variables known as history counters.

History Counters

- History counters provide information about the number of times an operation has been
  - requested
  - activated
  - completed

<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 currently active invocations of op</td>
</tr>
</tbody>
</table>
  (\#req - \#fin)
The Buffer Synchronized

- 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

The Buffer Synchronized (2)

- The previous predicates could also have been written using history counters:
  - For example
    - 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
Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples
  - VDMTools overview
    - The VDM++/UML Process with the alarm example
    - Industrial usage of VDMTools

VDMTools® Overview

- Java to VDM++
- Syntax & Type Checker
- Integrity Checker
- The Rose-VDM++ Link
- Interpreter (Debugger)
- Document Generator
- Code Generators
  - C++, Java
- API (Corba), DL Facility

Japanese Support via Unicode
Steps to Develop a Formal Model

1. Determine the purpose of the model.
2. Read the requirements.
3. Analyze the functional behavior from the requirements.
4. Extract a list of possible classes or data types (often from nouns) and operations (often from actions). Create a dictionary by giving explanations to items in the list.
5. Sketch out representations for the classes using UML class diagrams. This includes the attributes and the associations between classes. Transfer the model to VDM++ and check its internal consistency.
6. Sketch out signatures for the operations. Again, check the model's consistency in VDM++.
7. Complete the class (and data type) definitions by determining potential invariant properties from the requirements and formalizing them.
8. Complete the operation definitions by determining pre- and post conditions and operation bodies, modifying the type definitions if necessary.
9. Validate the specification using systematic testing and rapid prototyping.
10. Implement the model using automatic code generation or manual coding.
A Chemical Plant

A Chemical Plant Requirements

1. A computer-based system is to be developed to manage the alarms of this plant.
2. Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.
3. There must be experts on duty during all periods allocated in the system.
4. Each expert can have a list of qualifications.
5. Each alarm reported to the system has a qualification associated with it along with a description of the alarm that can be understood by the expert.
6. Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged.
7. The experts should be able to use the system database to check when they will be on duty.
8. It must be possible to assess the number of experts on duty.

The Purpose of the VDM++ Model

The purpose of the model is to clarify the rules governing the duty roster and calling out of experts to deal with alarms.
Creating a Dictionary

- Potential Classes and Types (Nouns)
  - Alarm: required qualification and description
  - Plant: the entire system
  - Qualification (electrical, mechanical, biological, chemical)
  - Expert: list of qualifications
  - Period (whatever shift system is used here)
  - System and system database? This is probably a kind of schedule.

- Potential Operations (Actions)
  - Expert to page: when an alarm appears (what’s involved? Alarm operator and system)
  - Expert is on duty: check when on duty (what’s involved? Expert and system)
  - Number of experts on duty: presumably given period (what’s involved? Operator and system)

Guideline 1

Nouns from a dictionary should be modeled as types if, for the purposes of the model, they need have only trivial functionality in addition to read/write.

Sketching an Alarm

Defined as a VDM++ class:

```vdm++
class Alarm
    instance variables
    reqQuali: Expert Qualification
    descr : String;
end Alarm
```
Alternative Alarm

Alarm could also have been defined as a composite type:

\[
\text{Alarm} :: \text{reqQuali} : \text{Expert}'\text{Qualification} \\
\text{descr} : \text{String}
\]

Then if \(a\) is of type \(\text{Alarm}\):

\[
\begin{align*}
\text{a.descr} & \text{ is the description of } a \\
\text{a.descr} & : \text{String} \\
\text{a.reqQuali} & : \text{Expert}'\text{Qualification}
\end{align*}
\]

Guideline 2

Create an overall class to represent the entire system so that the precise relationships between the different classes and their associations can be expressed there.

Guideline 3 and 4

Whenever an association is introduced consider its multiplicity and give it a rôle name in the direction in which the association is to be used.

If an association depends on some value, a qualifier should be introduced for the association. The name of the qualifier must be a VDM++ type.
Guideline 5

Declare instance variables to be private or protected to keep encapsulation. If nothing is specified by the user, private is assumed automatically.

class Expert
instance variables
private quali: set of Qualification;
end Expert

class Alarm
instance variables
private descr: String;
private reqQuali: Qualification;
end Alarm

Guideline 6 and 7

Use VDMTools to check internal consistency as soon as class skeletons have been completed and before any functionality has been introduced.

- Definition of types missing
- To be updated in the respective classes
- Resynchronized with the UML model

class Plant
types
Period = token;
end Plant

Tokens are useful for abstract models where unspecified values are to be used.
Adding Quantification and String

class Expert
types
  Qualification = <Mech> | <Chem> | <Bio> | <Elec>
end Expert

class Alarm
types
  public String = seq of char;
  instance variables
descr : String;
  reqQuali : Expert'Qualification;
end Alarm

Guideline 8

Think carefully about the parameter types and the result type as this often helps to identify missing connections in the class diagram.

Updated UML Class Diagram
Guideline 9

Document important properties or constraints as invariants.

```vdm++
class Plant
...
instance variables
alarms: set of Alarm;
schedule: map Period to set of Expert;
inv forall p in set dom schedule & schedule[p] <> {};
end Plant
```

Guideline 10

When there are several alternative ways of performing some functionality, use an implicit definition so that subsequent development work is not biased.

```vdm++
ExpertToPage: Alarm * Period ==> Expert
ExpertToPage(a, p) ==
is not yet specified
pre a in set alarms and
  p in set dom schedule
post let expert = RESULT
  in
    expert in set schedule(p) and
    a.GetReqQuali() in set expert.GetQuali();
```

Will the Qualification exist?

- How can we be sure that an expert with the required qualification exists in the required period?
- We need to add an invariant to the instance variables of the `Plant` class
- That is using guideline 11
Guideline 11

When defining operations, try to identify additional invariants.

instance variables
alarms : set of Alarm;
schedule : map Period to set of Expert;
inv forall p in set dom schedule & schedule(p) <> {};
inv forall a in set alarms & forall p in set dom schedule &
   exists expert in set schedule(p) &
   a.GetReqQuali() in set expert.GetQuali();

Guideline 12

Try to make explicit operation definitions precise and clear
and yet abstract compared to code written in a
programming language.

Further Operations inside Plant

class Plant
operations
--
public NumberOfExperts: Period => nat
NumberOfExperts(p) ==
return card schedule(p)
pres p in set dom schedule;
public ExpertIsOnDuty: Expert => set of Period
ExpertIsOnDuty(ex) ==
return {p | p in set dom schedule &
          ex in set schedule(p)};
end Plant

import java.util.*
class Plant {
    Map schedule;
    Set ExpertIsOnDuty(Integer ex) {
        TreeSet resset = new TreeSet();
        Set keys = schedule.keySet();
        Iterator iterator = keys.iterator();
        while(iterator.hasNext()) {
            Object p = iterator.next();
            if( ((Set) schedule.get(p)).contains(ex))
                resset.add(p);
        }
        return resset;
    }
}
Guideline 13

Whenever a class has an invariant on its instance variables and it has a constructor, it is worth placing the invariant in a separate function if the constructor needs to assign values to the instance variables involved in the invariant.

functions

PlantInv: set of Alarm * map Period to set of Expert -> bool
PlantInv(als,sch) ==
(foreall p in set dom sched & sched(p) <> {}) and
(foreall a in set als & forall p in set dom sched &
exists expert in set sched(p) &
a.GetReqQuali() in set expert.GetQuali());

To be used inside Plant Constructor

```plaintext
class Plant

public Plant: set of Alarm *
            map Period to set of Expert ->
Plant
Plant(als,sch) ==
(alarm := als;
schedule := sch)
pre PlantInv(als,sch);
end Plant
```
<table>
<thead>
<tr>
<th>Review Requirements (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1: A computer-based system managing this plant is to be developed.</td>
</tr>
<tr>
<td><em>Considered in the Plant class definition and the operation and function definitions.</em></td>
</tr>
<tr>
<td>R2: Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.</td>
</tr>
<tr>
<td><em>Considered in the Qualification type definition of the Expert class.</em></td>
</tr>
<tr>
<td>R3: There must be experts on duty at all times during all periods which have been allocated in the system.</td>
</tr>
<tr>
<td><em>Invariant on the instance variables of class Plant.</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Review Requirements (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4: Each expert can have a list of qualifications.</td>
</tr>
<tr>
<td><em>Assumption: non-empty set instead of list in class Expert.</em></td>
</tr>
<tr>
<td>R5: Each alarm reported to the system must have a qualification associated with it and a description which can be understood by the expert.</td>
</tr>
<tr>
<td><em>Considered in the instance variables of the Alarm class definition assuming that it is precisely one qualification.</em></td>
</tr>
<tr>
<td>R6: Whenever an alarm is received by the system an expert with the right qualification should be paged.</td>
</tr>
<tr>
<td><em>The ExpertToPage operation with additional invariant on the instance variables of the Plant class definition.</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Review the Requirements (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7: The experts should be able to use the system database to check when they will be on duty.</td>
</tr>
<tr>
<td><em>The ExpertOnDuty operation.</em></td>
</tr>
<tr>
<td>R8: It must be possible to assess the number of experts on duty.</td>
</tr>
<tr>
<td><em>The NumberOfExperts with assumption for a given period.</em></td>
</tr>
</tbody>
</table>
Agenda

- Part 1 (9:00 – 10:30) The VDM++ Language
- Part 2 (11:00 – 12:30) VDMTools and VDM++ examples
  - VDMTools overview
  - The VDM++/UML Process with the alarm example
  - Industrial usage of VDMTools

ConForm (1994)

- Organisation: British Aerospace (UK)
- Domain: Security (gateway)
- Tools: The IFAD VDM-SL Toolbox
- Experience:
  - Prevented propagation of error
  - Successful technology transfer
  - At least 4 more applications without support
- Statements:
  - “Engineers can learn the technique in one week”
  - “VDMTools can be integrated gradually into a traditional existing development process”

DustExpert (1995-7)

- Organisation: Adelard (UK)
- Domain: Safety (dust explosives)
- Tools: The IFAD VDM-SL Toolbox
- Experience:
  - Delivered on time at expected cost
  - Large VDM-SL specification
  - Testing support valuable
- Statement:
  - “Using VDMTools we have achieved a productivity and fault density far better than industry norms for safety related systems”
### Adelard Metrics

- **Initial requirements**: 450 pages
- **VDM specification**: 16kloc (31 modules), 12kloc (excl comments)
- **Prolog implementation**: 37kloc
- **C++ GUI implementation**: 16kloc (excl comments), 23kloc, 18kloc (excl comments)

- 31 faults in Prolog and C++ (<1/kloc)
- Most minor, only 1 safety-related
- 1 (small) design error, rest in coding

### CAVA (1998-2000)
- **Organisation**: Baan (Denmark)
- **Domain**: Constraint solver (Sales Configuration)
- **Tools**: The IFAD VDM-SL Toolbox
- **Experience**:
  - Common understanding
  - Faster route to prototype
  - Earlier testing
- **Statement**:
  - "VDMTools® has been used in order to increase quality and reduce development risks on high complexity products"

### Dutch DoD (1997-8)
- **Organisation**: Origin, The Netherlands
- **Domain**: Military
- **Tools**: The IFAD VDM-SL Toolbox
- **Experience**:
  - Higher level of assurance
  - Mastering of complexity
  - Delivered at expected cost and on schedule
  - No errors detected in code after delivery
- **Statement**:
  - "We chose VDMTools® because of high demands on maintainability, adaptability and reliability"
**DoD, NL Metrics (1)**

<table>
<thead>
<tr>
<th></th>
<th>kloc</th>
<th>hours</th>
<th>loc/hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>spec</td>
<td>15</td>
<td>1196</td>
<td>13</td>
</tr>
<tr>
<td>manual impl</td>
<td>4</td>
<td>471</td>
<td>8.5</td>
</tr>
<tr>
<td>automatic impl</td>
<td>90</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>test</td>
<td>NA</td>
<td>612</td>
<td>NA</td>
</tr>
<tr>
<td>total code</td>
<td>94</td>
<td>2279</td>
<td>41.2</td>
</tr>
</tbody>
</table>

- Estimated 12 C++ loc/h with manual coding!

**DoD - Comparative Metrics**

Traditional:
- 500
- 2000
- 700

VDMTools®:
- 1200
- 500
- 400

**BPS 1000 (1997-)**

- Organisation: GAO, Germany
- Domain: Bank note processing
- Tools: The IFAD VDM-SL Toolbox
- Experience:
  - Better understanding of sensor data
  - Errors identified in other code
  - Savings on maintenance
- Statement:
  - VDMTools provides unparalleled support for design abstraction ensuring quality and control throughout the development life cycle.
Flower Auction (1998)
- Organisation: Chess, The Netherlands
- Domain: Financial transactions
- Tools: The IFAD VDM++ Toolbox
- Experience:
  - Successful combination of UML and VDM++
  - Use iterative process to gain client commitment
  - Implementers did not even have a VDM course
- Statement:
  - "The link between VDMTools and Rational Rose is essential for understanding the UML diagrams"

SPOT 4 (1999)
- Organisation: CS-CI, France
- Domain: Space (payload for SPOT4 satellite)
- Tools: The IFAD VDM-SL Toolbox
- Experience:
  - 38% less lines of source code
  - 36% less overall effort
  - Use of automatic C++ code generation
- Statement:
  - The cost of applying Formal methods is significantly lower than without them.

Japanese Railways (2000-2001)
- Domain: Railways (database and interlocking)
- Experience:
  - Prototyping important
  - Now also using it for ATC system
  - Engineer working at IFAD for two years with PROSPER proof support
Stock-options (2000- )
• Organisation: JFITS (CSK group company), Japan
• Domain: Financial
• Tools: The IFAD VDM++ Toolbox
• Reason for CSK to purchase VDMTools

<table>
<thead>
<tr>
<th>Tax exemption</th>
<th>COCOMO</th>
<th>Realized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort</td>
<td>38.5 person months</td>
<td>14 person months</td>
</tr>
<tr>
<td>Schedule</td>
<td>9 months</td>
<td>3.5 months</td>
</tr>
</tbody>
</table>

Options COCOMO Realized
Effort 147.2 person months 60.1 person months
Schedule 14.3 months 7 months

Reverse Engineering (2001)
• Organisation: Boeing
• Domain: Avionics
• Tools: The IFAD VDM++ Toolbox
• Included development of Java to VDM++ reverse engineering feature

Optimisation (2001)
• Organisation: Transitive Technologies, UK
• Domain: Embedded
• Tools: The IFAD VDM-SL Toolbox
• Making software independent of hardware platform
Quote of the day

The successful construction of all machinery depends on the perfection of the tools employed, and whoever is the master in the art of tool-making possesses the key to the construction of all machines.

*Charles Babbage, 1851*

Any questions?