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

iha.dk

- 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?

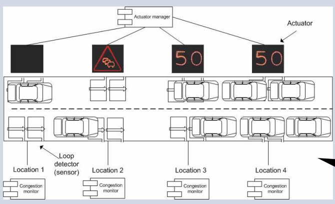
iha.dk

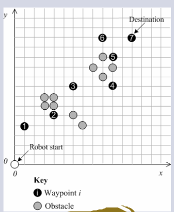
- Modeling critical computer systems e.g. for industries such as
  - Avionics
  - Railways
  - Automotive
  - Nuclear
  - Defense
- I have used this industrially for example at:
  - Boeing, Lockheed-Martin (USA)
  - British Aerospace, Rolls Royce, Adelard (UK)
  - Matra, Dassault, Aerospatiale (France)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

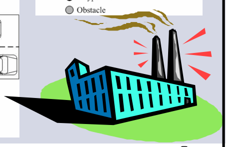
Industrially Inspired Examples

iha.dk

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







Validation Techniques

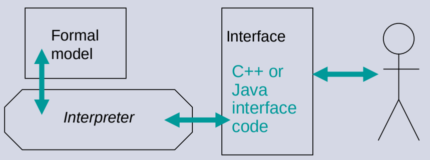
iha.dk

- 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

iha.dk

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.



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

Agenda

iha.dk

- ➢ 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

VDM++ Class Outline

```
class <class-name>
   instance variables
     ...                      } Internal object state
   types
   values
   functions                  } Definitions
   operations
     ...
   thread
     ...                      } Dynamic behaviour
   sync
     ...                      } Synchronization control
end <class-name>
```


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

Agenda

iha.dk

- 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)

iha.dk

- Used to model attributes
- Consistency properties modelled as invariants

```
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)

iha.dk

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

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

```
class Company
  ...
end Company
```

Agenda

iha.dk

- 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

Type Definitions

iha.dk

- 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

iha.dk

not b	Negation	**bool** -> **bool**
a **and** b	Conjunction	**bool** * **bool** -> **bool**
a **or** b	Disjunction	**bool** * **bool** -> **bool**
a => b	Implication	**bool** * **bool** -> **bool**
a <=> b	Biimplication	**bool** * **bool** -> **bool**
a = b	Equality	**bool** * **bool** -> **bool**
a <> b	Inequality	**bool** * **bool** -> **bool**

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** -> **real**
abs x	Absolute value	**real** -> **real**
floor x	Floor	**real** -> **int**
x + y	Sum	**real** * **real** -> **real**
x - y	Difference	**real** * **real** -> **real**
x * y	Product	**real** * **real** -> **real**
x / y	Division	**real** * **real** -> **real**
x **div** y	Integer division	**int** * **int** -> **int**
x **rem** y	Remainder	**int** * **int** -> **int**
x **mod** y	Modulus	**int** * **int** -> **int**
x ** y	Power	**real** * **real** -> **real**

Numeric (2)

x < y	Less than	**real** * **real** -> **bool**
x > y	Greater than	**real** * **real** -> **bool**
x <= y	Less or equal	**real** * **real** -> **bool**
x >= y	Greater or equal	**real** * **real** -> **bool**
x = y	Equal	**real** * **real** -> **bool**
x <> y	Not equal	**real** * **real** -> **bool**

Product and Record Types

- Product type definition:
  `A1 * A2 * … * An`

  Construction of a tuple:
  mk_(a1,a2,…,an)
- Record type definition:

```
A :: selfirst : A1
     selsec   : A2
      …
     seln     : An
```

  Construction of a record:
  mk_A(a1,a2,...,an)

Example Record Definition

iha.dk

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

iha.dk

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

iha.dk

`e in set s1`	Membership (∈)	`A * set of A -> bool`
`e not in set s1`	Not membership (∉)	`A * set of A -> bool`
`s1 union s2`	Union (∪)	`set of A * set of A -> set of A`
`s1 inter s2`	Intersection (∩)	`set of A * set of A -> set of A`
`s1 \ s2`	Difference (\)	`set of A * set of A -> set of A`
`s1 subset s2`	Subset (⊆)	`set of A * set of A -> bool`
`s1 psubset s2`	Proper subset (⊂)	`set of A * set of A -> bool`
`s1 = s2`	Equality (=)	`set of A * set of A -> bool`
`s1 <> s2`	Inequality (≠)	`set of A * set of A -> bool`
`card s1`	Cardinality	`set of A -> nat`
`dunion s1`	Distr. Union (⋃)	`set of set of A -> set of A`
`dinter s1`	Distr. Intersection (⋂)	`set of set of A -> set of A`
`power s1`	Finite power set (ℙ)	`set of A -> set of set of A`

Set Comprehensions

iha.dk

- Using predicates to define sets implicitly
- In VDM++ formulated like:
  - {*element* | *list of bindings* & *predicate*}
- The predicate part is optional
- Quick examples:
  - {3 * x | x : **nat** & x < 3} or {3 * x | x **in set** {0,…,2}}
  - {x | x : **nat** & x < 5} or {x | x **in set** {0,…,4}}

Sequence Operators

iha.dk

hd l	Head	**seq1 of** A -> A
tl l	Tail	**seq1 of** A -> **seq of** A
len l	Length	**seq of** A -> **nat**
elems l	Elements	**seq of** A -> **set of** A
inds l	Indexes	**seq of** A -> **set of nat1**
l1 ^ l2	Concatenation	**seq of** A * **seq of** A -> **seq of** A
conc ll	Distr. conc.	**seq of seq of** A -> **seq of** A
l(i)	Seq. application	**seq1 of** A * **nat1** -> A
l ++ m	Seq. modification	**seq1 of** A * **map nat1 to** A -> **seq1 of** A
l1 = l2	Equality	**seq of** A * **seq of** A -> **bool**
l1 <> l2	Inequality	**seq of** A * **seq of** A -> **bool**

Sequence Comprehensions

iha.dk

- Using predicates to define sequences implicitly
- In VDM++ formulated like:
  - [*element* | *numeric set binding* & *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 * x | x **in set** {0,…,2}]
  - [x | x **in set** {0,…,4} & x > 2]

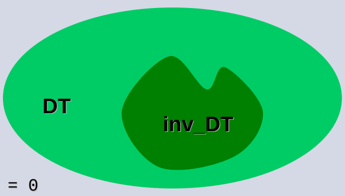
Map Operators

dom m	Domain	(**map** A **to** B) -> **set of** A
rng m	Range	(**map** A **to** B) -> **set of** B
m1 **munion** m2	Merge	(**map** A **to** B) * (**map** A **to** B) -> (**map** A **to** B)
m1 ++ m2	Override	(**map** A **to** B) * (**map** A **to** B) -> (**map** A **to** B)
merge ms	Distr. merge	**set of** (**map** A **to** B) -> **map** A **to** B
s <: m	Dom. restr. to	**set of** A * (**map** A **to** B) -> **map** A **to** B
s <-: m	Dom. restr. by	**set of** A * (**map** A **to** B) -> **map** A **to** B
m :> s	Rng. restr. to	(**map** A **to** B) * **set of** A -> **map** A **to** B
m :-> s	Rng. restr. by	(**map** A **to** B) * **set of** A -> **map** A **to** B
m(d)	Map apply	(**map** A **to** B) * A -> B
inverse m	Map inverse	**inmap** A **to** B -> **inmap** B **to** A
m1 = m2	Equality	(**map** A **to** B) * (**map** A **to** B) -> **bool**
m1 <> m2	Inequality	(**map** A **to** B) * (**map** A **to** B) -> **bool**

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 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 * B * … * Z -> R1 * R2 * … * Rn
f(a,b,…,z) ==
  expr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,RESULT)
```

- Implicit functions:

```
f(a:A, b:B, …, z:Z) r1:R1, …, rn:Rn
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,r1,…,rn)
```

  Implicit functions cannot be executed by the VDM interpreter.

Function Definitions (2)

- Extended explicit functions:

```
f(a:A, b:B, …, z:Z) r1:R1, …, rn:Rn ==
  expr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,r1,…,rn)
```

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

```
f: A * B * … * Z -> R1 * R2 * … * Rn
f(a,b,…,z) ==
  is not yet specified
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,RESULT)
```

Quoting pre- and post-conditions

iha.dk

Given an implicit function definition like:

```
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:

```
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
```


Agenda

iha.dk

- 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

iha.dk

•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

iha.dk

- 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
```

VDM++ tutorial Introduction 37

Example Let-be expressions

iha.dk

- 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
```

VDM++ tutorial Introduction 38

If-then-else Expressions

iha.dk

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>
```

VDM++ tutorial Introduction 39

Cases Expressions

iha.dk

Cases expressions are very powerful because of pattern matching:

```
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

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


Set Expressions

iha.dk

- Set enumeration:
  `{a,3,3,true}`
- Set comprehension can either use set binding:
  `{a+2 | mk_(a,a) in set {mk_(true,1),mk_(1,1)}}`

  or type binding:
  `{a | a: nat & a<10}`
- Set range expression:
  `{3,...,10}`

Sequence Expressions

iha.dk

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

Map Expressions

iha.dk

- 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

iha.dk

- A tuple expression looks like:
  `mk_(2,7,true,{|->})`
- 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

iha.dk

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)
mu (mk_A(7,false,{1,4}), n|->1, s|->{})
mu (mk_B(3,4), r|->5.5)
```
The **mu** operator is called "the record modifier".

Apply Expressions

iha.dk

- 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

iha.dk

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

`is_nat(5)` will yield true.

`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.

`is_A(mk_B(3,7))` will always yield false.

`is_A(6)` will also always yield false.

Define Expressions

iha.dk

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

The state could be changed by an operation call:

```
def a = OpCall(arg1,arg2) in f(a)
```

or parts of the state could simply be read:

```
def a = instance_variable in g(a)
```

Lambda Expressions

iha.dk

- Lambda expressions are an alternative way of defining explicit functions.
```
lambda n: nat & n * n
```
- They can take a type bind list:
```
lambda a: nat, b: bool &
   if b then a else 0
```
- or use more complex types:
```
lambda mk_(a,b): nat * nat & a + b
```


New and Self Expressions

iha.dk

- 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 yield its reference:
```
Create: nat ==> C
Create (n) ==
( a := n;
  return self )
```


Class Membership Expressions

iha.dk

Check if an object is of a particular class.

```
isofclass(Class_name,object_ref)
```

Returns true if *object_ref* is of class *Class_name* or a subclass of *Class_name*.

Check for the baseclass of a given object.

```
isofbaseclass(Class_name,object_ref)
```

For the result to be true, *object_ref* must be of class *Class_name*, and *Class_name* cannot have any superclasses.

Object Comparison Expressions

iha.dk

Compare two objects.

```
sameclass(obj1,obj2)
```

True if and only if *obj1* and *obj2* are instances of the same class

- ❒ **sameclass**(m,s) ≡ false
- ❒ **sameclass**(m, **new** Manager()) ≡ true

Comparison of baseclasses of two objects.

```
samebaseclass(obj1,obj2)
```

- ❒ **samebaseclass**(m,s) ≡ true
- ❒ **samebaseclass**(m, **new** Temporary()) ≡ false

Object Reference Expressions

iha.dk

- 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

iha.dk

- 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:
```
pat in set expr
```
  where *expr* must denote a set expression.
  pat is bound to the elements of the set *expr*
- A type binding:
```
pat : type
```
  Here *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 * B * ... ==> R
o(a,b,...) ==
  stmt
pre expr
post expr
```
- Implicit operations definitions:
```
o(a:A, b:B,...) r:R
ext rd ...
    wr ...
pre expr
post expr
```

Operation Definitions (2)

iha.dk

- Preliminary operation definitions:

```
o: A * B * ... ==> R
o(a,b,...) ==
  is not yet specified
pre expr
post expr
```

- Delegated operation definitions:

```
o: A * B * ... ==> R
o(a,b,...) ==
  is subclass responsibility
pre expr
post expr
```


Operation Definitions (3)

iha.dk

- 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

iha.dk

An implicit operation definition could look like:

```
Withdraw(amount: nat) newBalance: int
ext rd limit   : int
    wr balance : int
pre balance - amount > limit
post balance + amount = balance~ and newBalance = balance
```

An explicit operation definition could look like:

```
Withdraw: nat ==> int
Withdraw(amount) ==
( balance := balance - amount;
  return balance
)
pre balance - amount > limit
```

Agenda

iha.dk

- 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

iha.dk

- 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

iha.dk

- 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

Concurrency Primitives in VDM++

iha.dk

- Concurrency in VDM++ is based on *threads*
- Threads communicate using shared objects
- Synchronization on shared objects is specified using *permission predicates*

Threads

iha.dk

- Modelled by a class with a thread section

```
class SimpleThread
thread
  let - = 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

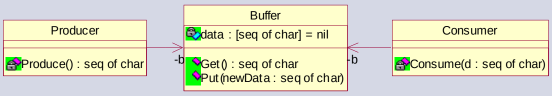
iha.dk

- Threads operating in isolation have limited use.
- In VDM++ threads communicate using shared objects.

A Producer-Consumer Example

iha.dk

- 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

iha.dk

```
class Producer

instance variables

b : Buffer

operations

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

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


The Consumer Class

iha.dk

```
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

iha.dk

```
class Buffer

instance variables

data : [seq of char] := nil

operations

public Put: seq of char ==> ()
Put(newData) ==
  data := newData;

public Get: () ==> seq of char
Get() ==
  let oldData = data
  in
  ( data := nil;
    return oldData
  )

end Buffer
```


Permission Predicates

iha.dk

- 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.

Permission Predicates

iha.dk

- General structure

```
sync

per operation name => predicate;
    ...
```

- For Put and Get we could write:

```
per Put => data = nil;

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

History Counters and mutex

iha.dk

Counter	Description
`#req op`	The number of times that op has been requested
`#act op`	The number of times that op has been activated
`#fin op`	The number of times that op has been completed
`#active op`	The number of active executions of op

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

Permission Predicates: Details

iha.dk

- 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

iha.dk

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

Counter	Description
#req(op)	The number of times that op has been requested
#act(op)	The number of times that op has been activated
#fin(op)	The number of times that op has been completed
#active(op)	The number of currently active invocations of op (#req - #fin)

The Buffer Synchronized

iha.dk

- 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`

VDM++ tutorial Introduction 76

The Buffer Synchronized (2)

iha.dk

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

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

VDM++ tutorial Introduction 77

Mutual Exclusion

iha.dk

- 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`

VDM++ tutorial Introduction 78

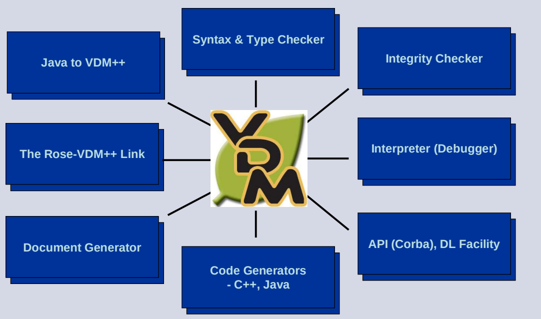
Agenda

iha.dk

- ✓ 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

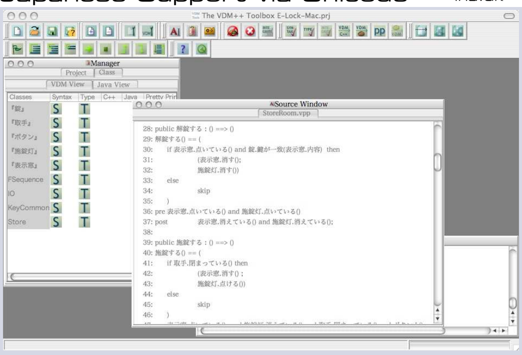
VDMTools® Overview

iha.dk



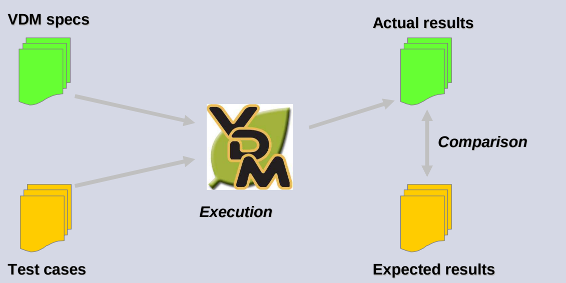
Japanese Support via Unicode

iha.dk



Validation with VDMTools®

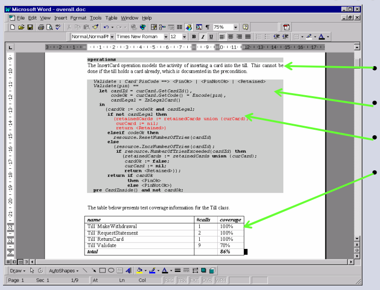
iha.dk



Documentation in MS Word/RTF

iha.dk

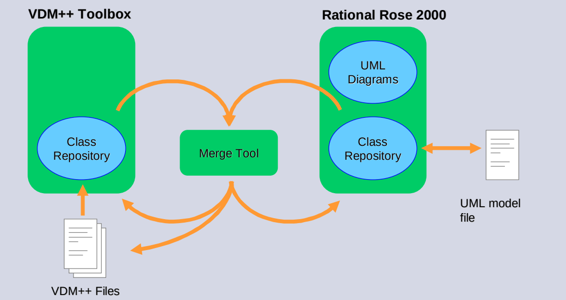
One compound document:



- Documentation
- Specification
- Test coverage
- Test coverage statistics

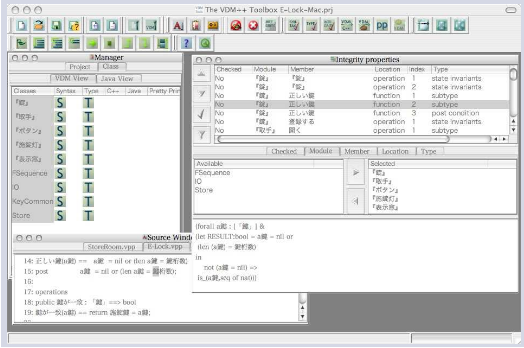
Architecture of the Rose VDM++ Link

iha.dk



Integrity checker

iha.dk



Agenda

iha.dk

- ✓ 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

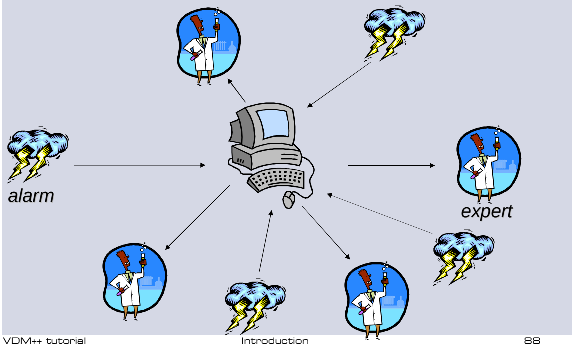
Steps to Develop a Formal Model

iha.dk

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 this 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

iha.dk



A Chemical Plant Requirements

iha.dk

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

iha.dk

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

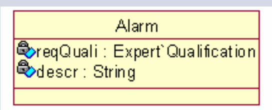
iha.dk

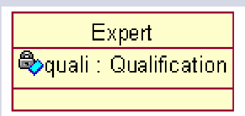
- Potential Classes and Types (Nouns)
  - Alarm: required qualification and description
  - Plant: the entire system
  - Qualification (electrical, mechanical, biological, chemical)
  - Expert: list of qualifications
  - Period (whatever shift system is used here)
  - System and system database? This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

iha.dk

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.



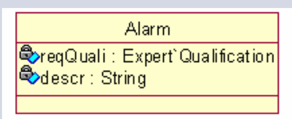


Sketching an Alarm

iha.dk

Defined as a VDM++ class:

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



Alternative Alarm

iha.dk

Alarm could also have been defined as a composite type:

```
Alarm :: reqQuali : Expert`Qualification
         descr    : String
```

Then if *a* is of type *Alarm*:

`a.descr` is the description of *a*

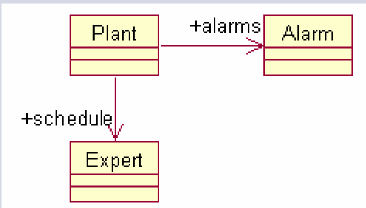
`a.descr : String`

`a.reqQuali : Expert`Qualification`

Guideline 2

iha.dk

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

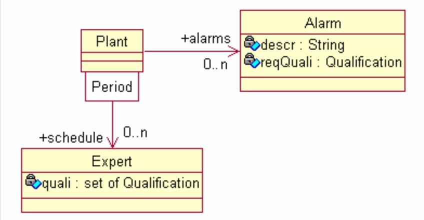
iha.dk

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.

Initial Class Diagram

iha.dk




```
class Plant
instance variables
public alarms   : set of Alarm;
public schedule : map Period to set of Expert;

end Plant
```


Guideline 5

iha.dk

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

iha.dk

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

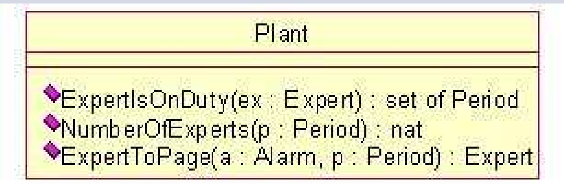
iha.dk

```
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

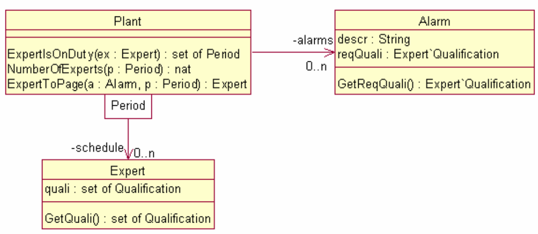
iha.dk

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

iha.dk



Guideline 9

iha.dk

Document important properties or constraints as invariants.

```
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

iha.dk

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

```
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?

iha.dk

- 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

iha.dk

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();
```


Further Operations inside Plant

iha.dk

```
class Plant
operations
…

public NumberOfExperts: Period ==> nat
NumberOfExperts(p) ==
  return card schedule(p)
pre 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
```


Guideline 12

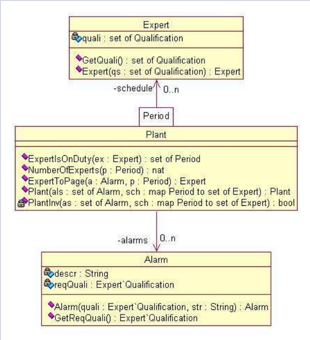
iha.dk

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

```
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;
 }
}
```

Final UML Class Diagram

iha.dk



Guideline 13

iha.dk

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(as,sch) ==
  (forall p in set dom sch & sch(p) <> {}) and
  (forall a in set as &
     forall p in set dom sch &
       exists expert in set sch(p) &
         a.GetReqQuali() in set expert.GetQuali());
```


To be used inside Plant Constructor

iha.dk

```
class Plant
…
public Plant: set of Alarm *
              map Period to set of Expert ==>
              Plant
Plant(als,sch) ==
( alarms := als;
  schedule := sch
)
pre PlantInv(als,sch);
end Plant
```

Review Requirements (1)

iha.dk

R1: A computer-based system managing this plant is to be developed.

Considered in the `Plant` class definition and the operation and function definitions.

R2: Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.

Considered in the `Qualification` type definition of the `Expert` class.

R3: There must be experts on duty at all times during all periods which have been allocated in the system.

Invariant on the instance variables of class `Plant`.

Review Requirements (2)

iha.dk

R4: Each expert can have a list of qualifications.

Assumption: non-empty set instead of list in class `Expert`.

R5: Each alarm reported to the system must have a qualification associated with it and a description which can be understood by the expert.

Considered in the instance variables of the `Alarm` class definition assuming that it is precisely one qualification.

R6: Whenever an alarm is received by the system an expert with the right qualification should be paged.

The `ExpertToPage` operation with additional invariant on the instance variables of the `Plant` class definition.

Review the Requirements (3)

iha.dk

R7: The experts should be able to use the system database to check when they will be on duty.

The `ExpertOnDuty` operation.

R8: It must be possible to assess the number of experts on duty.

The `NumberOfExperts` with assumption for a given period.

Agenda

iha.dk

- ✓ 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

VDM++ tutorial Introduction 115

ConForm (1994)

iha.dk

- 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"

VDM++ tutorial Introduction 116

DustExpert (1995-7)

iha.dk

- 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"

VDM++ tutorial Introduction 117

Adelard Metrics

iha.dk

Initial requirements	450 pages
VDM specification	16kloc (31 modules) 12kloc (excl comments)
Prolog implementation	37kloc 16kloc (excl comments)
C++ GUI implementation	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)

iha.dk

- 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)

iha.dk

- 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)

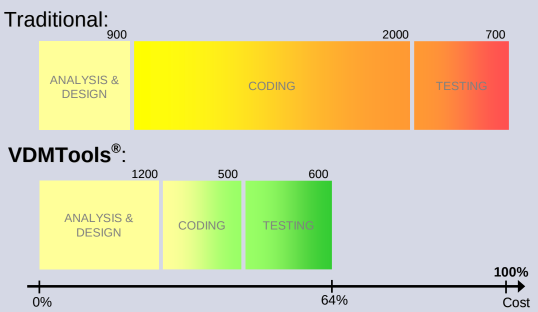
iha.dk

	kloc	hours	loc/hour
spec	15	1196	13
manual impl	4	471	8.5
automatic impl	90	0	NA
test	NA	612	NA
total code	94	2279	41.2

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

DoD - Comparative Metrics

iha.dk



BPS 1000 (1997-)

iha.dk

- 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)

iha.dk

- 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)

iha.dk

- 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)

iha.dk

- 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-)

iha.dk

- Organisation: JFITS (CSK group company), Japan
- Domain: Financial
- Tools: The IFAD VDM++ Toolbox
- Reason for CSK to purchase VDMTools

Tax exemption	COCOMO	Realized
Effort	38,5 person months	14 person months
Schedule	9 months	3,5 months

Options	COCOMO	Realized
Effort	147,2 person months	60,1 person months
Schedule	14,3 months	7 months

Reverse Engineering (2001)

iha.dk

- Organisation: Boeing
- Domain: Avionics
- Tools: The IFAD VDM++ Toolbox
- Included development of Java to VDM++ reverse engineering feature

Optimisation (2001)

iha.dk

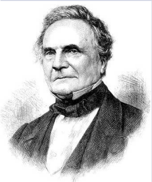
- Organisation: Transitive Technologies, UK
- Domain:Embedded
- Tools: The IFAD VDM-SL Toolbox
- Making software independent of hardware platform

Quote of the day

iha.dk

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?