VDMTools®

Validated Design through Modelling

Overview of VDM -SL/++

IFAD A/S
Forskerparken 10
DK-5230 Odense M
Denmark

VDM-SL

- ISO Standard 1996 for flat language
- Different module proposals
- A de-facto standard module approach
  - Imports
  - Exports
  - Parameterisation
  - Instantiation

VDM-SL Module Outline

`module` <*module-name*>

- ~~`parameters`~~
- `imports`
- ~~`instantiations`~~
- `exports`
- ...

Interface

`definitions`

- `state`
- `types`
- `values`
- `functions`
- `operations`
- ...

Definitions

`end` <*module-name*>

VDM++ Class Outline

```
class <class-name>

  instance variables
    ...

  types
  values
  functions
  operations
    ...

  thread
    ...

  sync
    ...

end <class-name>
```

- instance variables: Internal object state
- types, values, functions, operations: Definitions
- thread: Dynamic behaviour
- sync: Synchronization control

VDM++ Overview

- **Access Modifiers and Constructors**
- Instance Variables
- Types
- Functions
- Expressions,Patterns,Bindings
- Operations
- Statements
- Concurrency

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

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ➢ **Instance Variables**
- Types
- Functions
- Expressions,Patterns,Bindings
- Operations
- Statements
- Concurrency

Instance Variables (1)

- 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)

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

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

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

Instance Variable Access

- Instance variables can only be accessed directly from within the object they belong to.
- To read/write instance variables “from outside” access operations must be defined

```
class Person
    ...
instance variables
  name: string := [];
    ...
operations
  public GetName: () ==> string
  GetName () ==
    return name
end Person
```

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ➢ **Types**
- Functions
- Expressions,Patterns,Bindings
- Operations
- Statements
- Concurrency

Type Definitions

- Basic types
  - Boolean
  - Numeric
  - Tokens
  - Characters
  - Quotations

Invariants can be added

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

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`	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)
```

Set Operators

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	set of A -> set of set of A

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`

Sequence Operators

`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`

Comprehension Notation

Convenient comprehensions exist for sets, maps and sequences:

- Set comprehension:

  `{ elem | bind-list & pred }` e.g.

  `{ x * 2 | x in set {1,…,10} & x mod 2 = 0}`

- Map comprehension:

  `{ maplet | bind-list & pred }` e.g.

  `{ x |-> f(x) | x in set s & p(x)}`

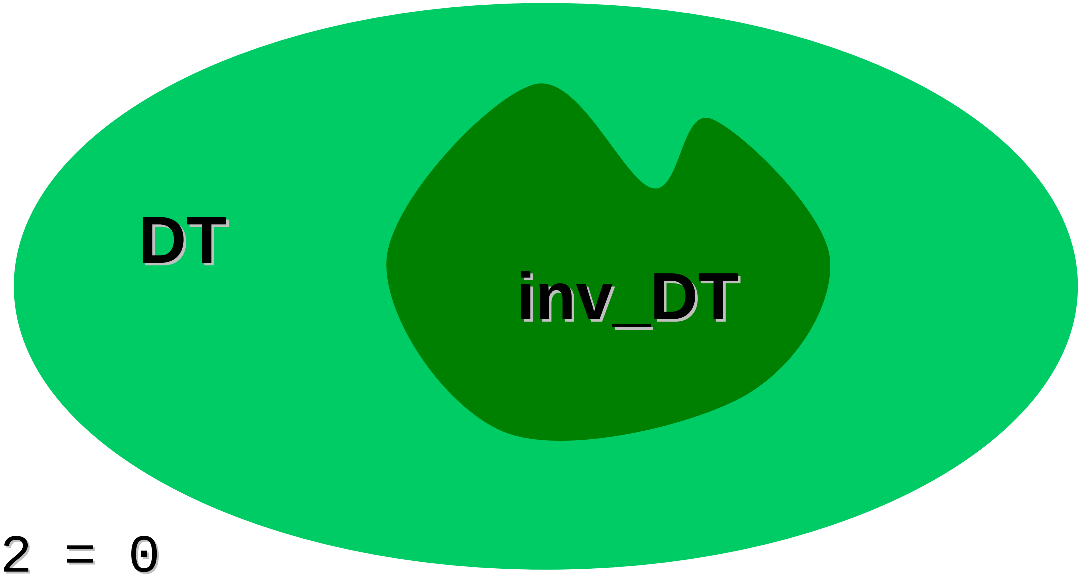
- Sequence comprehension:

  `[ elem | setbind & pred ]` e.g.

  `[ l(i) ** 2 | I in set inds l & l(i) < 10]`

- The set binding is restricted to sets of numeric values, which are used to find the order of the resulting sequence

Invariants




```
Even = nat
inv n == n mod 2 = 0

SpecialPair = nat * real
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 = {}
```

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ✓ Types
- ➢ **Functions**
- Expressions,Patterns,Bindings
- Operations
- Statements
- Concurrency

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)
```

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ✓ Types
- ✓ Functions
- **Expressions,Patterns,Bindings**
- Operations
- Statements
- Concurrency

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

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:

  ```
  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 Toolbox, because such types can be infinitely large.

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ✓ Types
- ✓ Functions
- ✓ Expressions,Patterns,Bindings
- ➢ **Operations**
- Statements
- Concurrency

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)

- 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)

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

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ✓ Types
- ✓ Functions
- ✓ Expressions,Patterns,Bindings
- ✓ Operations
- ➢ **Statements**
- Concurrency

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

Special VDM++ Statement

- start and startlist statements

VDM++ Overview

- ✓ Access Modifiers and Constructors
- ✓ Instance Variables
- ✓ Types
- ✓ Functions
- ✓ Expressions,Patterns,Bindings
- ✓ Operations
- ✓ Statements
- ➢ **Concurrency**

Concurrency in VDM++

Objects can be

- **Passive:** Change state on request only, i.e. as a consequence of an operation invocation.
- **Active:** Can change their internal state spontaneously without any influence from other objects. Active objects have their own thread of control.

Why use concurrency in specifications?

- The real world is highly concurrent.
  Consequently models of the world are likely to be concurrent too.
- For efficiency reasons in a multi processor environment.

Passive Objects

- Respond to requests (operation invocations) from active objects (clients).
- Supply an interface (a set of operations) for their clients.
- No thread.
- Can serve several clients.

Permission Guards

Synchronization for objects is specified using VDM++'s `sync` clause:

```
sync
  per <operation-name> => <condition>
```

The `per` clause is known as a *permission guard. condition* is a boolean expression, which involves the attributes of the class, that must hold in order for *operation-name* to be invoked.

Permission guards reflecting the bounding of the buffer :

```
sync
  per GetItem => len buf > 0
  per PutItem => len buf < size
```

Further Information

John Fitzgerald, Peter Gorm Larsen

Modelling Systems, Practical Tools and Techniques in Software Development

John Dawes

The VDM-SL Reference Guide

Derek Andrews, Darrel Ince

Practical Formal Methods with VDM

Cliff Jones

Systematic Software Development with VDM (2nd edition)

John Lathan, Vicky Bush, Ian Cottam

The Programming Process

John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat

Round-trip engineering with VDM++ and UML (forthcomming)