The Overture Approach to VDM Language Evolution

Nick Battle Anne Haxthausen
Sako Hiroshi **Peter Jørgensen**
Nico Plat Shin Sahara Marcel Verhoef

The Overture Language Board

28 August 2013
The 11th Overture Workshop
Methods, Tools and Techniques for Modelling in VDM

Outline

VDM and the Overture Language Board (1/2)

1. VDM has evolved into several dialects
   - VDM-SL dates back to 1970 and is ISO standardized
   - VDM++ adds object-orientation (OO)
   - VDM-RT for distributed computation and deployment
2. All languages evolve
   - New needs arise from insights and applications
3. EU research has had impact on the language evolution
   - The Afrodite ESPRIT project proposed OO extensions
   - The VICE project proposed RT extensions (single CPU)
   - Improved RT extensions were proposed in Verhoef's PhD
   - VDM-RT is co-simulated in the DESTECS project
   - VDM++ is used in CML defined by the COMPASS project

VDM and the Overture Language Board (2/2)

1. 2004: The language was adopted by the Overture project
   - In the years following the language was defined by the tool
2. 2009: The Overture Language Board (LB) was introduced
   - Consisting of members from the Overture community
   - Coordinates changes and clarifies issues with the language
   - Handles Request for Modifications (RMs)
   - Holds regular closed Net Meetings (NMs) on Skype
   - NOT responsible for releasing tools
3. 2010: The language was termed VDM-10
   - For the family of languages based on VDM-SL

Request for Modification (RM)

Proposing changes to the language

1. Anyone can propose a RM (the originator)
2. The RM describes the change:
   - Rationale
   - Semantics
   - Impact on the language etc.
3. An owner (LB member) is assigned to the RM
   - The owner moves the RM forward in the process
4. SourceForge (SF) is used for managing RMs
   - SF provides audit trail for each RM
5. Accepted RMs undergo four phases (next slide)

RM Phases

1. Initial consideration
   - The LB may request expert opinions
   - The RM is rejected or approved (maybe subject to revision)
2. Discussion
   - The wider Overture community may participate
   - The discussion takes place via email and during NMs
3. Deliberation
   - The RM may be rejected or accepted without modifications
   - Accepted with modifications: The originator revises the RM
4. Execution
   - The owner must update the Language Reference Manual
   - The Overture release manager plans tool updates

RM Phases

1. Initial consideration
   - The LB may request expert opinions
   - The RM is rejected or approved (maybe subject to revision)
2. Discussion
   - The wider Overture community may participate
   - The discussion takes place via email and during NMs
3. Deliberation
   - The RM may be rejected or accepted without modifications
   - Accepted with modifications: The originator revises the RM
4. Execution
   - The owner must update the Language Reference Manual
   - The Overture release manager plans tool updates

RM Phases

1. Initial consideration
   - The LB may request expert opinions
   - The RM is rejected or approved (maybe subject to revision)
2. Discussion
   - The wider Overture community may participate
   - The discussion takes place via email and during NMs
3. Deliberation
   - The RM may be rejected or accepted without modifications
   - Accepted with modifications: The originator revises the RM
4. Execution
   - The owner must update the Language Reference Manual
   - The Overture release manager plans tool updates

RM Phases

1. Initial consideration
   - The LB may request expert opinions
   - The RM is rejected or approved (maybe subject to revision)
2. Discussion
   - The wider Overture community may participate
   - The discussion takes place via email and during NMs
3. Deliberation
   - The RM may be rejected or accepted without modifications
   - Accepted with modifications: The originator revises the RM
4. Execution
   - The owner must update the Language Reference Manual
   - The Overture release manager plans tool updates

RM History

RMs dealt with by the LB so far

ID	Summary	Milestone	Status	Tool support
1	International character support for Overture identifiers.	Completed	Closed	yes
2	The introduction of the“reverse” sequence operator.	Execution	Closed	yes
3	Generalising let-be expressions.	Completed	Closed	yes
4	Functions should be implicitly static.	Completed	Closed	yes
5	Scope rules for assignments.	Completed	Closed	yes
6	Inheritence of constructors.	Withdrawn	Closed	no
7	Adding explicit object reference expressions to VDM++.	Withdrawn	Closed	no
8	Need definition of VDM++ operation pre/post functions.	Withdrawn	Closed	no
9	VDM++ object oriented issues.	Rejected	Closed	no
10	Invariant functions for record types	Rejected	Closed	no
11	Exception handling in interpreter.	Rejected	Closed	no
12	Include the non-deterministic statement inside traces.	Execution	Open	yes (Overture only).
13	Static Initialization.	Withdrawn	Closed	no
14	Object Construction.	Withdrawn	Closed	no
15	Inheritance, Overloading, Overriding and Binding.	Withdrawn	Closed	no
16	Expressions in periodic thread definitions.	Execution	Open	yes (Overture only).
17	Values in duration / cycles statements.	Execution	Open	yes (Overture only).
18	Sporadic thread definitions.	Discussion	Open	n.a.
19	Extend duration and cycles (allow intervals + probabilities).	Execution	Open	no
20	Antagonist STOP operation for periodic threads is missing.	Discussion	Open	n.a.
21	More descriptive time expressions.	Discussion	Open	n.a.
22	Append narrow expression.	Execution	Open	yes
23	Append map pattern.	Completed	Closed	yes

RM #23, Map Patterns

Example 1 (November 2011)

pattern = ... *map enumeration pattern*
| *map munion pattern*
| ... ;

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,4 |-> 2,2 |-> 3}
in c -- Returns 3
```

```
let { a |-> 2,b |-> 2 } = {1 |-> 2,2 |-> 2}
in a -- (a = 1, b = 2) or (a = 2 or b = 1)
```

```
let {a |-> 2} = {1 |-> 2,2 |-> 2}
in a -- Runtime error
-- Error 4152: Wrong number of elements for map pattern
```

RM #23, Map Patterns

Example 1 (November 2011)

pattern = ... *map enumeration pattern*
| *map munion pattern*
| ... ;

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,4 |-> 2,2 |-> 3}
in c -- Returns 3
```

```
let { a |-> 2,b |-> 2 } = {1 |-> 2,2 |-> 2}
in a -- (a = 1, b = 2) or (a = 2 or b = 1)
```

```
let {a |-> 2} = {1 |-> 2,2 |-> 2}
in a -- Runtime error
-- Error 4152: Wrong number of elements for map pattern
```

RM #23, Map Patterns

Example 1 (November 2011)

pattern = ... *map enumeration pattern*
| *map munion pattern*
| ... ;

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,4 |-> 2,2 |-> 3}
in c -- Returns 3
```

```
let { a |-> 2,b |-> 2 } = {1 |-> 2,2 |-> 2}
in a -- (a = 1, b = 2) or (a = 2 or b = 1)
```

```
let {a |-> 2} = {1 |-> 2,2 |-> 2}
in a -- Runtime error
-- Error 4152: Wrong number of elements for map pattern
```

RM #23, Map Patterns

Example 1 (November 2011)

pattern = ... *map enumeration pattern*
| *map munion pattern*
| ... ;

```
let {1 |-> a,a |-> b,b |-> c} = {1 |-> 4,4 |-> 2,2 |-> 3}
in c -- Returns 3
```

```
let { a |-> 2,b |-> 2 } = {1 |-> 2,2 |-> 2}
in a -- (a = 1, b = 2) or (a = 2 or b = 1)
```

```
let {a |-> 2} = {1 |-> 2,2 |-> 2}
in a -- Runtime error
-- Error 4152: Wrong number of elements for map pattern
```

RM #12, Non-deterministic statements in traces

Example 2 (June 2010)

trace concurrent expression = ‘| |’, ‘(’, *trace definition*, ‘,’, *trace definition*, ‘,’, *trace definition*, { ‘,’, *trace definition* }, ‘)’ ;

```
-- This trace:
|| (Op1(); Op2(); Op3())
```

```
-- Expands to this:
(Op1(); Op2(); Op3()) |
(Op1(); Op3(); Op2()) |
(Op2(); Op1(); Op3()) |
(Op2(); Op3(); Op1()) |
(Op3(); Op2(); Op1()) |
(Op3(); Op1(); Op2())
```

RM #12, Non-deterministic statements in traces

Example 2 (June 2010)

trace concurrent expression = ‘| |’, ‘(’, *trace definition*, ‘,’, *trace definition*, ‘,’, *trace definition*, { ‘,’, *trace definition* }, ‘)’ ;

```
-- This trace:
|| (Op1(); Op2(); Op3())
```

```
-- Expands to this:
(Op1(); Op2(); Op3()) |
(Op1(); Op3(); Op2()) |
(Op2(); Op1(); Op3()) |
(Op2(); Op3(); Op1()) |
(Op3(); Op2(); Op1()) |
(Op3(); Op1(); Op2())
```

RM #12, Non-deterministic statements in traces

Example 2 (June 2010)

trace concurrent expression = ‘| |’, ‘(’, *trace definition*, ‘,’, *trace definition*, ‘,’, *trace definition*, { ‘,’, *trace definition* }, ‘)’ ;

```
-- This trace:
|| (Op1(); Op2(); Op3())
```

```
-- Expands to this:
(Op1(); Op2(); Op3()) |
(Op1(); Op3(); Op2()) |
(Op2(); Op1(); Op3()) |
(Op2(); Op3(); Op1()) |
(Op3(); Op2(); Op1()) |
(Op3(); Op1(); Op2())
```

RM #2, The “reverse” sequence operator

Example 3 (May 2009)

1. Prior to this RM **`reverse`** was limited to for loops
2. The change proposed a generalization of **`reverse`**
   - While removing the redundant **`reverse`** from the for loop
3. **`reverse`** was proposed a new unary operator
4. The RM introduced a change in for loop semantics
5. `for elem in reverse S1^S2`
   - Used to mean the reverse of the entire concatenated list
   - Under the new grammar this means `(reverse S1)^S2`
   - Warning 5013: New reverse syntax affects this statement

RM #3, Generalising let-be expressions

Example 4 (May 2009)

1. Prior to this RM **`let be st`** used a simple pattern bind
   - So did the equivalent trace **`let be st`** binding
   - The RM proposed extending this to include a multiple bind
2. The change proposed by the RM
   - Helps avoiding use of nested let expressions
   - Is backward compatible with existing specifications
3. The RM was considered a natural extension
   - It was accepted with little discussion

let be expression = *'**`let`**',*
multiple bind,
*['**`be`**', '**`st`**', expression],*
*'**`in`**', expression ;*

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

RM #4, Functions should be implicitly static in VDM++

Example 5 (May 2009)

At the LB NM on the 20th Sept. 2009 it was agreed that:

1. Every function is implicitly static
2. Reference to `self` is not allowed in a function
3. `obj.fn()` binding is determined by the actual type of obj
4. `fn()` binding is determined statically
5. `C`fn()` is still possible, as is `obj.B`fn()`
6. Calling operations in a function definition is disallowed
7. 1), 2) and 6) where new at the time
   - For existing models they generate deprecated warning
   - Going forward, violations of these will be type errors

Future plans

1. Widening the LB scope to cover standard libraries
   - The Overture tool standard libraries: IO, VDMUtil etc.
   - Legacy models depend on these libraries
2. OO issues of VDM
   - Example: Object Oriented Issues in VDM++, [Battle&10]
     - When an operation call occur in a super class constructor
     - Does that call an overridden version in the subclass?
     - .. or does it always call the version for the super class?
   - The formal semantics of VDM++ needs to be defined
   - The LB would like to coordinate this
     - ..But it is difficult to find the resources for it