Métodos Formais em Engenharia de Software

Ana Paiva

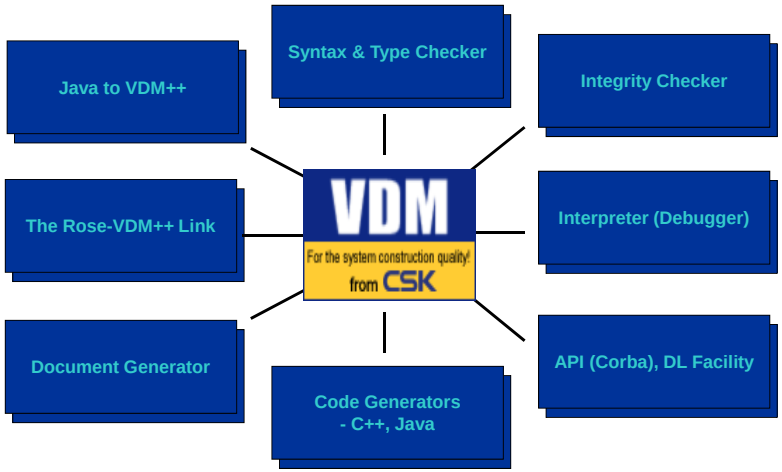
apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

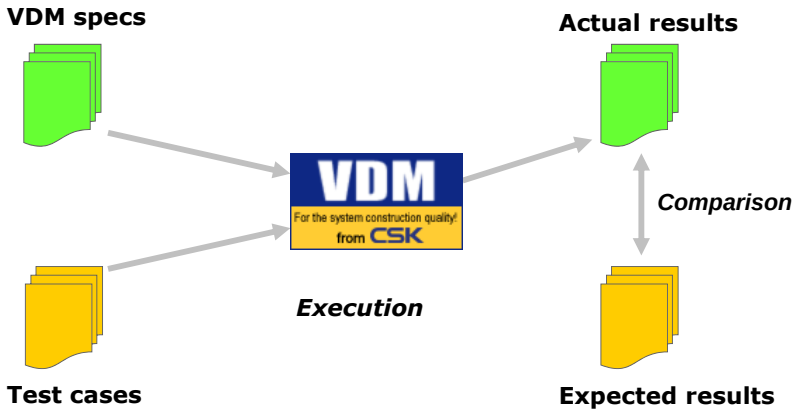
Agenda

- VDMTools
- Characteristics of the VDM++ language
  - Classes; Instance variables; Operations; Functions (polymorphic, Higher-order functions, lambda, …); Types; Operators; Expressions
  - Design-by-contact:
    - Definitions of invariants; pre and postconditions
    - Link between VDM++ and UML
- Internal consistency: proof obligations
- Concurrency in VDM++

VDMTools - Overview

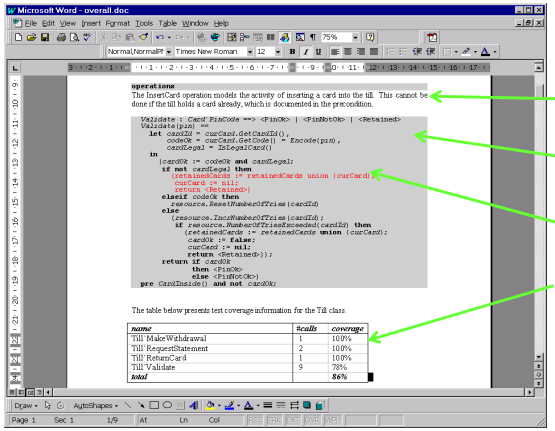


Validation with VDMTools

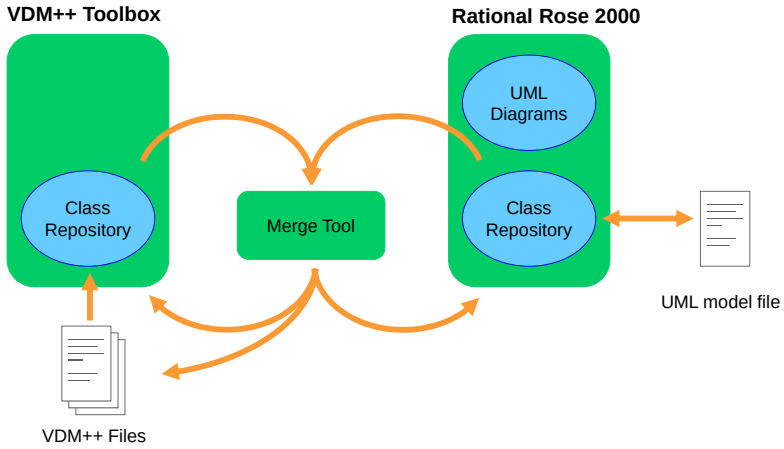


Documentation in MS Word/RTF

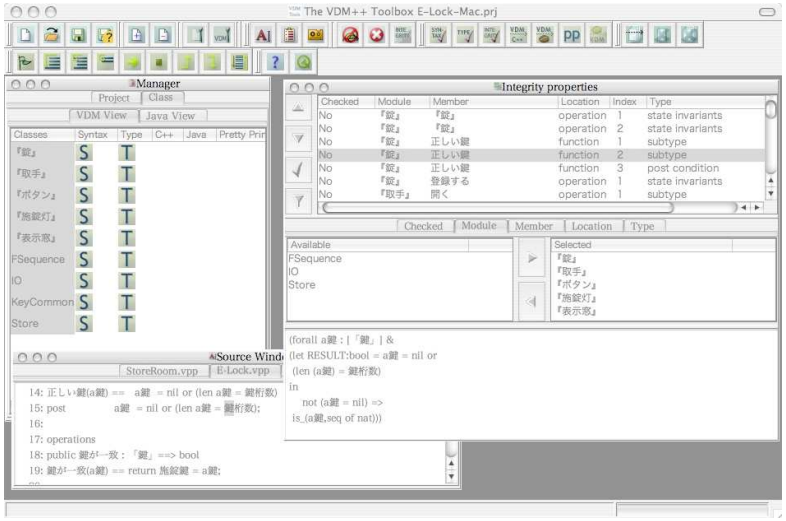
One compound document:



Architecture of the Rose VDM++ Link

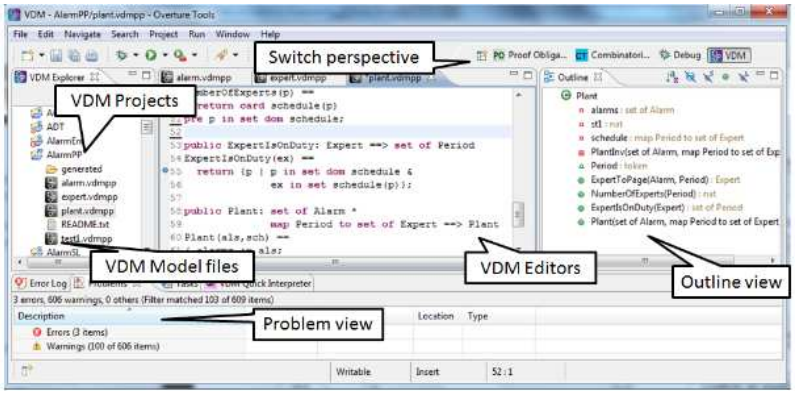


Integrity checker



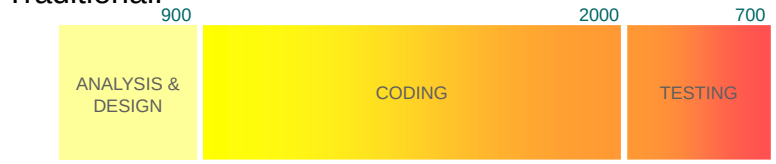
Overture project

- http://www.overturetool.org/

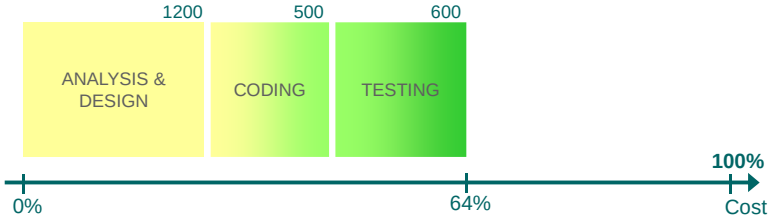


Process

Traditional:



VDMTools®:



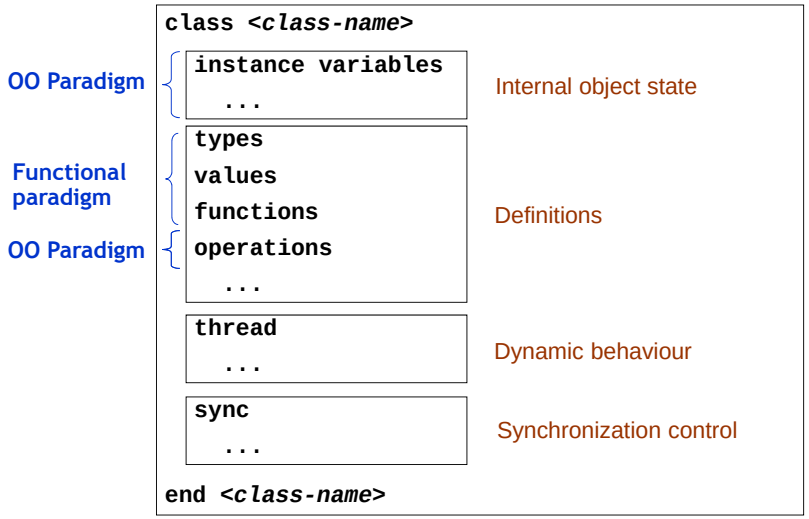
Agenda

- VDMTools
- Characteristics of the VDM++ language
  - Classes; Instance variables; Operations; Functions (polymorphic, Higher-order functions, lambda, …); Types; Operators; Expressions
  - Design-by-contact:
    - Definitions of invariants; pre and postconditions
    - Link between VDM++ and UML
- Internal consistency: proof obligations
- Concurrency in VDM++

Main characteristics of the VDM++

- Based on the standard VDM-SL (Vienna Development Method)
- Formal model based specification language (i.e., explicit representation of the state) object oriented
- Combination of two paradigm
  - Paradigm **functional**: types, functions and values
  - Paradigm **OO**: classes, instance variables, operations and objects
- Suported by VDMTools that allow:
  - The execution of an VDM++ specification
  - Specification testing and test coverage analysis
  - Synchronize with UML class diagrams in Rational Rose
  - Code generation to Java and C++
- Two notations available: ASCII and math symbols

VDM++ Class Outline



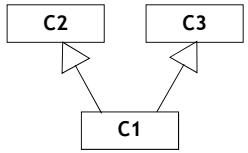
Classes

- A specification written in VDM++ is organized into classes
- Classes are reference types
  - Like what happens in several OO languages
  - Instances are mutable objects accessible by a reference
  - Variable of type C, where C is a class, contains a reference to the object with the data and not the data itself
  - Comparison and assignment operate with **references**
- Use to model the system state
  - State is represented by the set of existing objects and values of its instance variables
  - Classes represent types of physical entities (person, book room, ...), roles (teacher, student, ...), events (class, ...), documents (invoice, contract, ..., etc.).

Inheritance

- A class may have several super-classes (multiple inheritance)
- Sintax:

```
class C1 is subclass of C2, C3
  ...
end C1
```



- Usual semantics
- Polymorphism

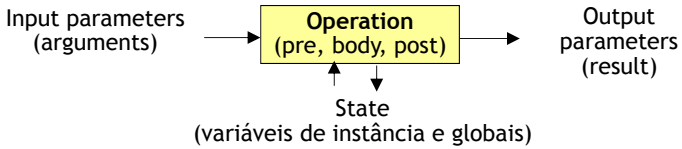
Instance variables

- Correspond to attributes in UML, and fields in Java and C#
- Can be private (by default), public or protected
- Can be static
- Declared in section “instance variables” with sintax:

  [private | public | protected] [static] *nome* : *tipo* [:= *valor_inicial*];

- Can define invariants (*inv*) that restrict the set of valid values for the instance variables

Operations



- Correspond to operations in UML and methods in Java or C#
- Can be private (by default), public or protected
- They can be static
- Can view or modify the state of objects (given by instance vars) or the overall state of the system (given by static vars)
- May have pre-condition, body (explicit definition, mandatory) and post-condition (implicit definition)

Operations - definition

- Style 1:

```
op(a: A, b: B, …, z: Z) r: R ==
   bodystmt
```

argument, type, result, type, omit when returns nothing

Variables read/written

```
ext rd instvarx, instvary, …
    wr instvarz, instvarw, …
pre preexpr(a, b, …, instvar1, instvar2, …)
post postexpr(a, b, …, r, instvar1, instvar2, …,
                  instvar1~, instvar2~, …) ;
```

- Style 2:

```
op: A * B * … ==> R
op(a,b,…) ==
  bodystmt
pre preexpr(a, b,…, instvar1, instvar2, …)
post postexpr(a, b,…, RESULT, instvar1, instvar2, …,
                  instvar1~, instvar2~, …) ;
```

when there aren't neither arguments nor results, write ()

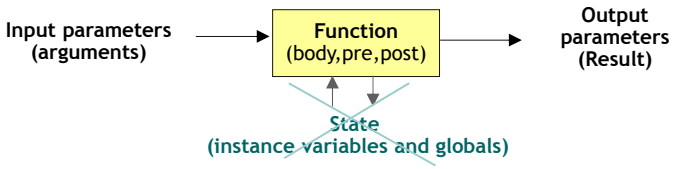
Predefined name for the return value

Variable state before the execution of the operation

Operation - definition

- Precondition (pre) - restriction on argument values and instance variables, to check the call
  - Can be omitted (even if true)
- Algorithmic body (bodysmt) – statement(s) between ()
  - Algorithm allows to express and execute the operation (explicit definition)
  - Imperative paradigm: c / assignments, variable declaration, etc..
  - Abstract operation "is subclass responsibility"
  - Operation to define: "is not yet specified," or omit "bodystmt ==" style 1.
- Postcondition (post) - the restriction on the values of the arguments, result, baseline and final vars ("~", *Tilde*) instance, to check on return
  - Check the result / effect of the transaction (implicit definition)
  - Can be omitted (even if true)
- Clause "ext" (externals) - lists the instance variables that can be read (rd) and updated (wr) in the body of the operation
  - Required to indicate the style 1, when shown the postcondition

Functions - definition



- Pure functions without side effects, convert inputs into outputs
- Not have access (either read or change) the state of the system represented by instance variables
- Are defined in section *functions*
- They can be *private* (default), *public* or *protected*
- They can be *static* (normal case)
- May have pre-condition, body (for explicit definition, functional paradigm) and post-condition (for implicit definition)

Functions - definition

- Style 1:

May have several output parameters

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

- Style 2:

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

(simple or tuple)

Functions

- Body - explicit definition of the result(s) of the function by an expression without side effects
  - Functional paradigm, executable (to calculate the result)
  - We omit it: write "is not specified yet" or omit "bodyexpr ==" style 1
- Precondition (pre) - restriction on the values of the arguments that must check in function call
  - Allows you to define partial functions (not defined for some values of the arguments)
  - Can be omitted (even if true)
  - The precondition of a function f is also a function called pre_f
- Postcondition (post) - Boolean expression that relates the function result w / arguments (the restriction that it must obey the result)
  - Implicit definition of function (lets you check but not to calculate the result)
  - Can be omitted (even if true)
  - The post-condition of a function f is also a function called post_f

Functions – examples

- Given an implicit function, for example:

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

- There are two additional functions, automatically created, which can be used in the specification:

```
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
```


Functions – examples

- Explicit definition (executable), total function

```
public static IsLeapYear(year: nat1) res : bool ==
  year mod 4 = 0 and year mod 100 <> 0 or year mod 400 = 0;
```

- Implicit definition (not executable), partial function

```
public static sqrt(x: real) res : real
 pre x >= 0
 post res * res = x and res >= 0;
```

- Recursive explicit function

```
fac: nat1 -> nat1
fac (n) ==
    if n > 1
    then n * fac(n-1)
    else 1
```

- Function with precondition

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