Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto

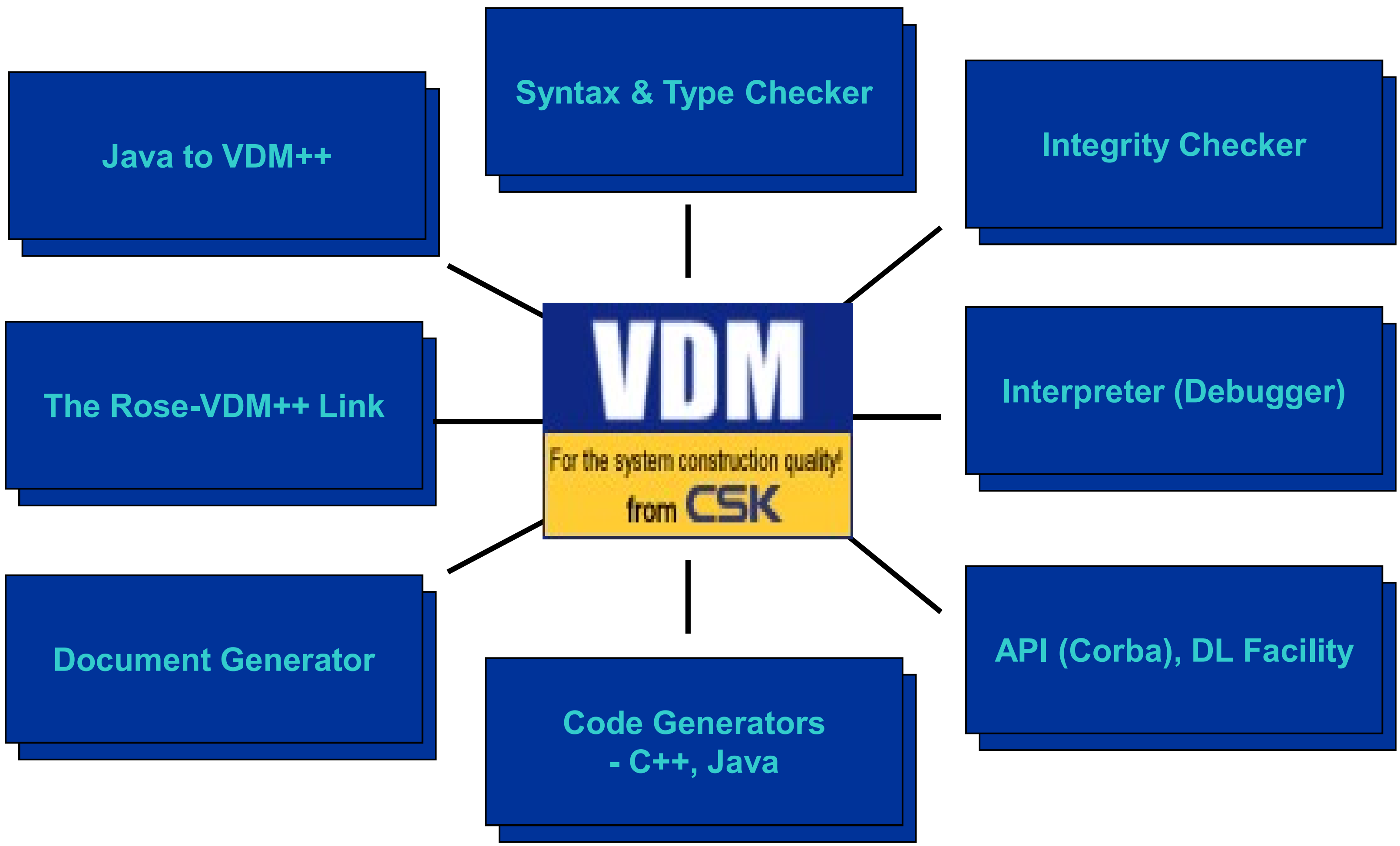
Faculdade de Engenharia

FEUP

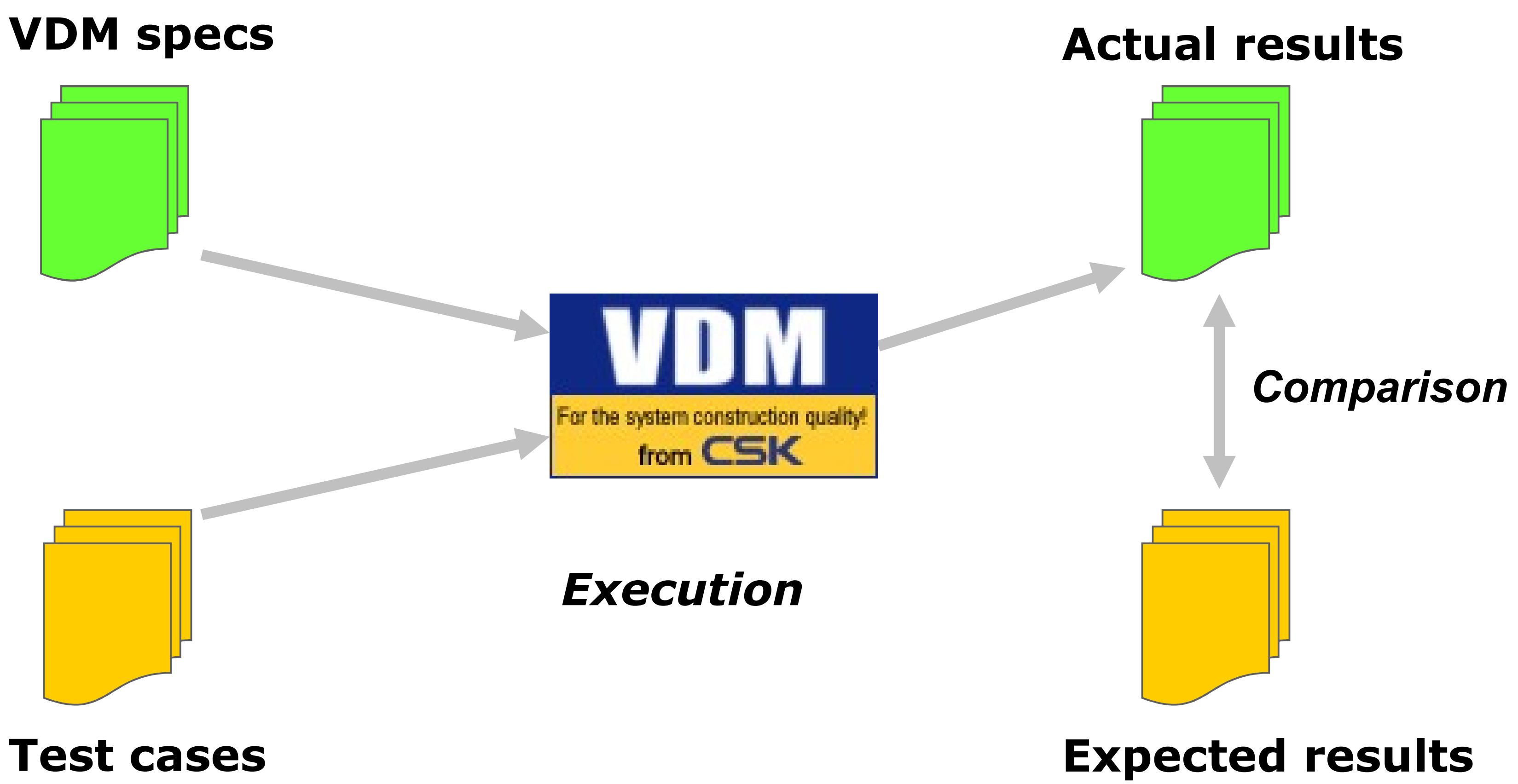
Agenda

- 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
- Example: Vending Machine
- Concurrency in VDM++

VDMTools - Overview

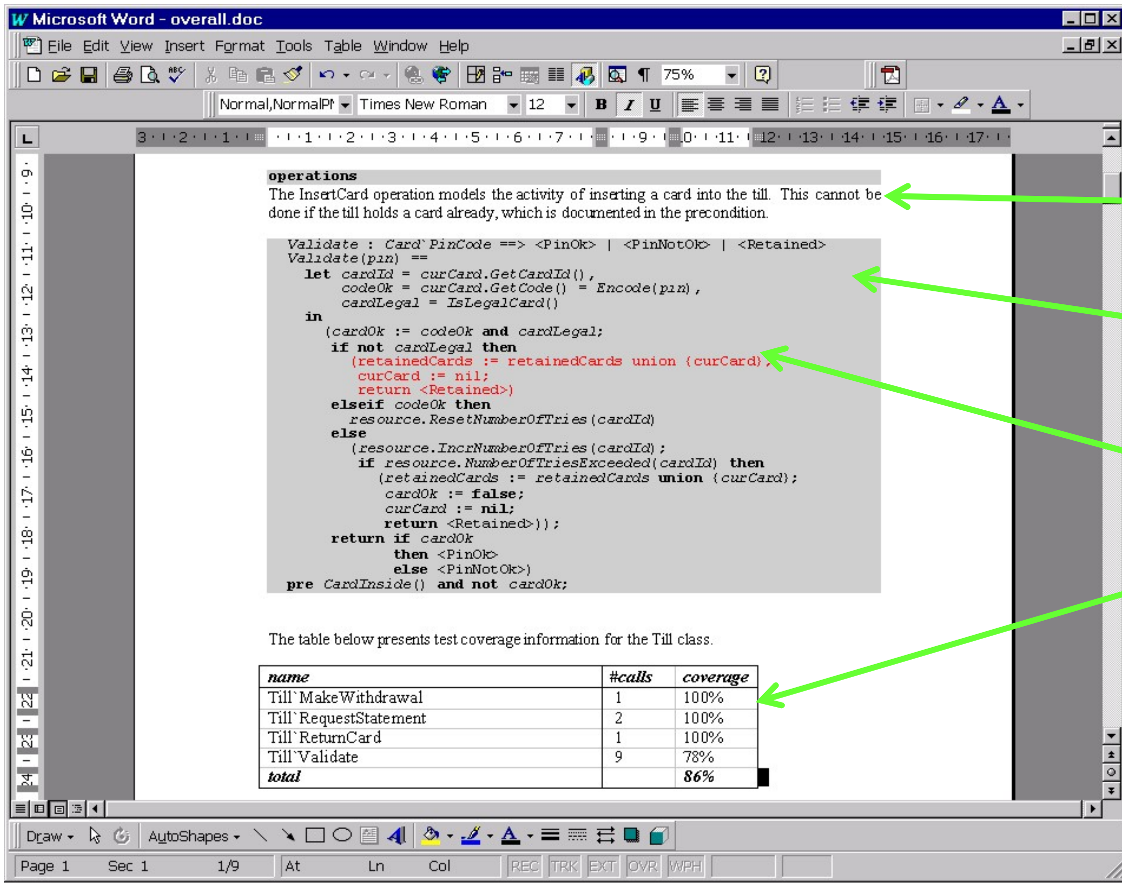


Validation with VDMTools



Documentation in MS Word/RTF

One compound document:


Microsoft Word - overall.doc

operations

The InsertCard operation models the activity of inserting a card into the till. This cannot be done if the till holds a card already, which is documented in the precondition.

```
  Validate : Card`PinCode ==> <PinOk> | <PinNotOk> | <Retained>
  Validate(pin) ==
    let cardId = curCard.GetCardId(),
        codeOk = curCard.GetCode() = Encode(pin),
        cardLegal = IsLegalCard()
    in
      (cardOk := codeOk and cardLegal;
       if not cardLegal then
         (retainedCards := retainedCards union {curCard};
          curCard := nil;
          return <Retained>)
       elseif codeOk then
         resource.ResetNumberOfTries(cardId)
       else
         (resource.IncrNumberOfTries(cardId);
          if resource.NumberOfTriesExceeded(cardId) then
            (retainedCards := retainedCards union {curCard};
             cardOk := false;
             curCard := nil;
             return <Retained>));
       return if cardOk
              then <PinOk>
              else <PinNotOk>)
pre CardInside() and not cardOk;
```

The table below presents test coverage information for the Till class.

name	*#calls*	*coverage*
Till`MakeWithdrawal	1	100%
Till`RequestStatement	2	100%
Till`ReturnCard	1	100%
Till`Validate	9	78%
total		***86%***

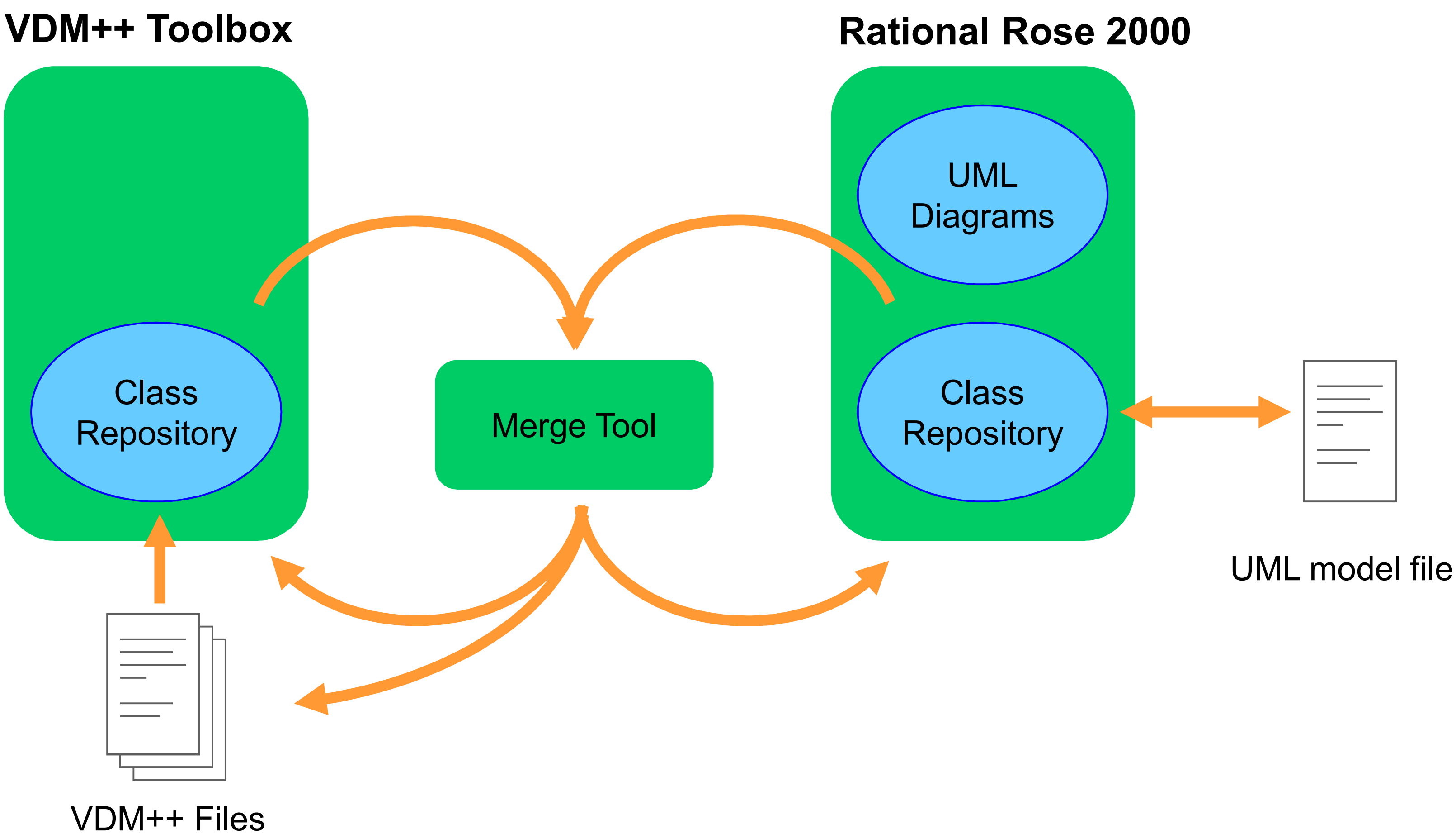
Documentation

Specification

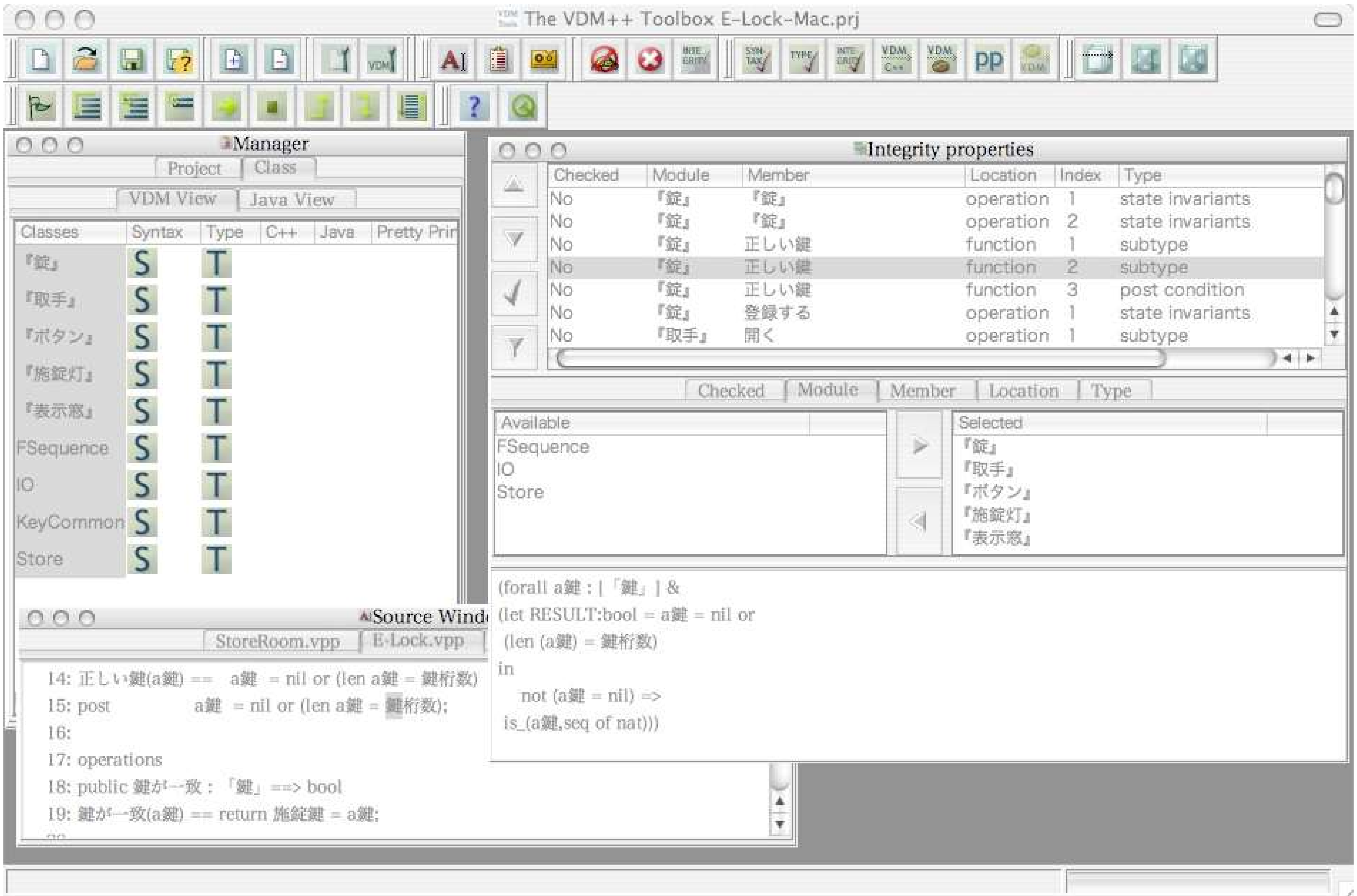
Test coverage

Statistics

Architecture of the Rose VDM++ Link

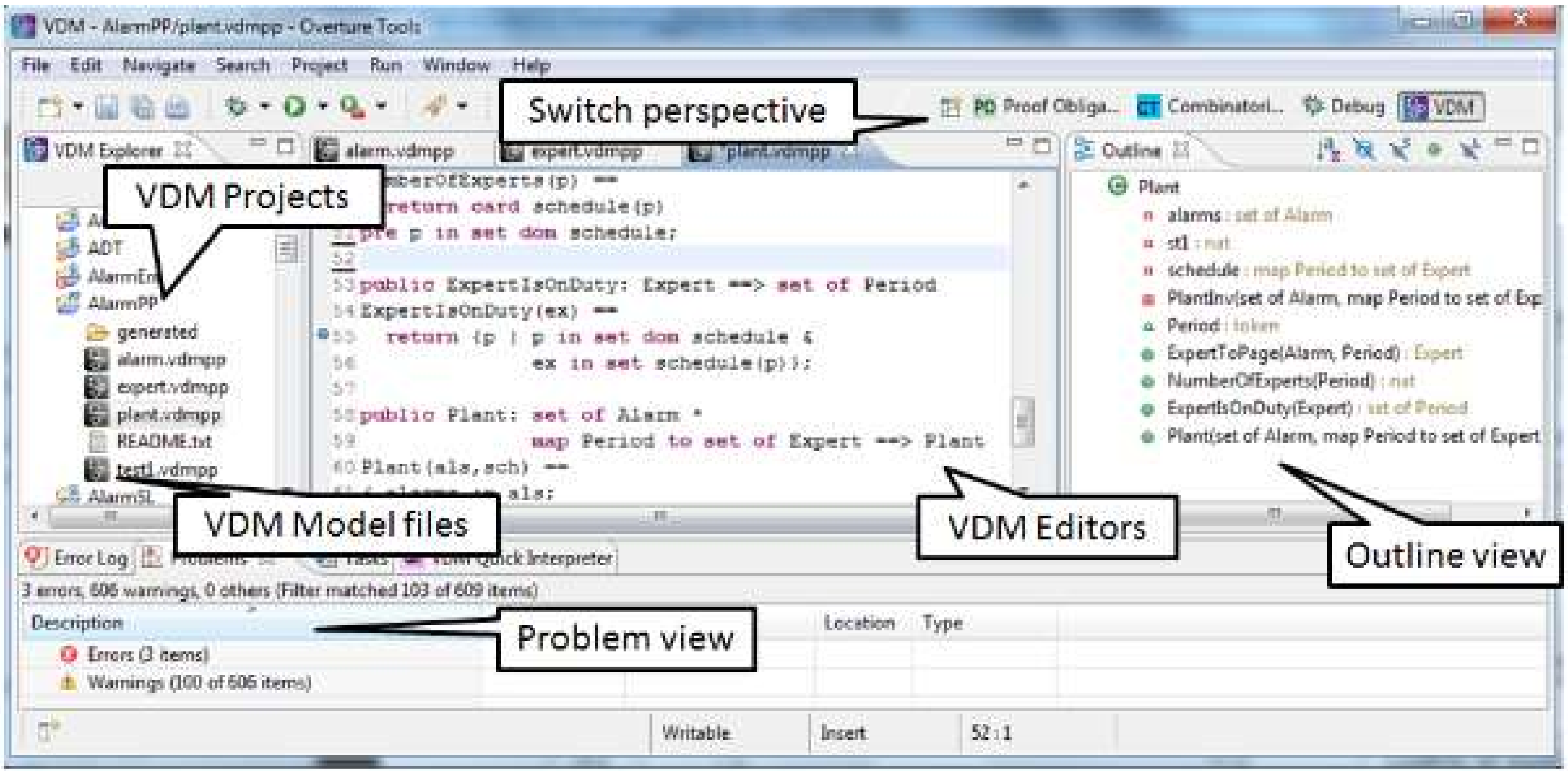


Integrity checker

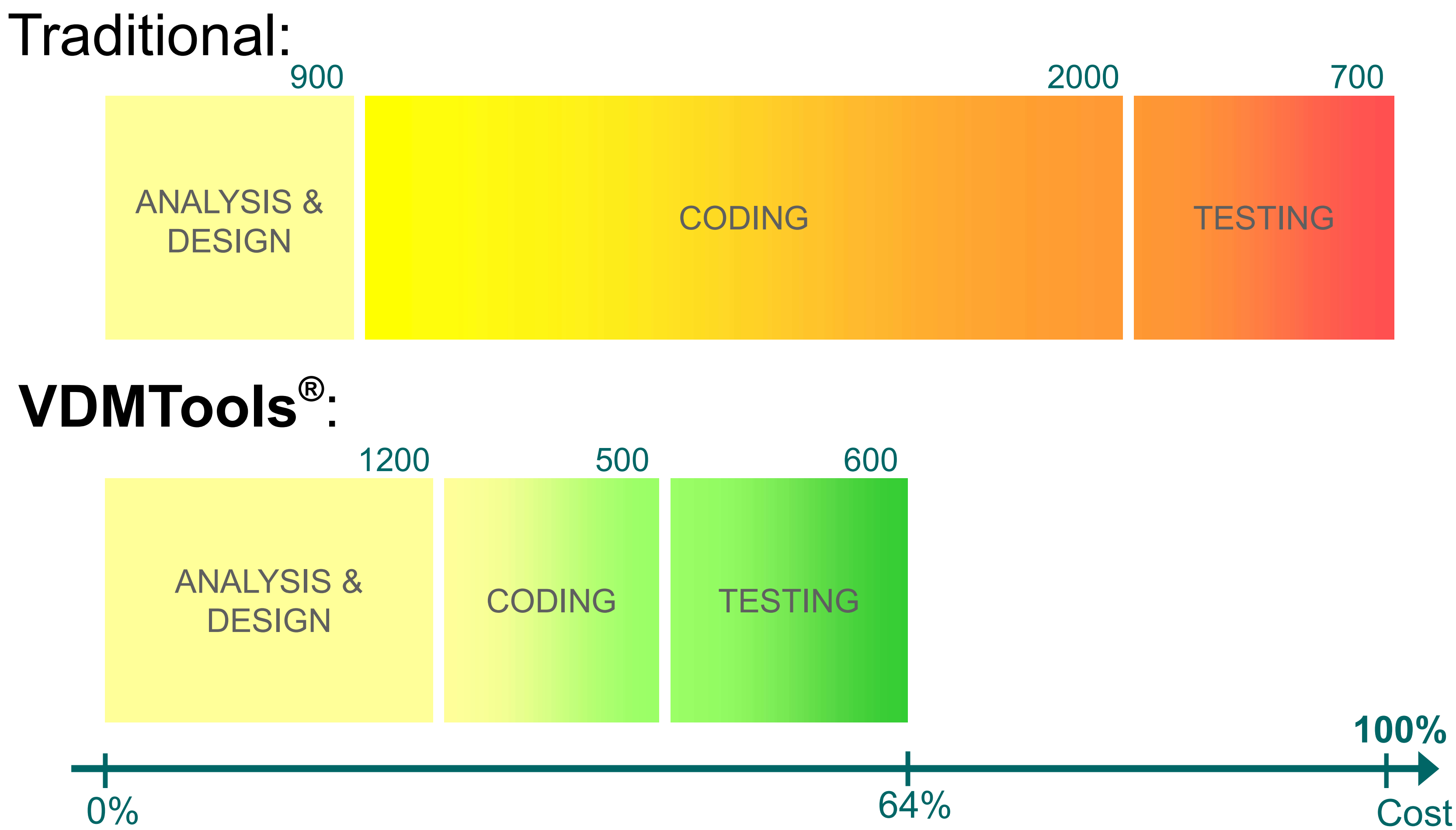


Overture project

- http://www.overturetool.org/



Process


Traditional:
900
2000
700
ANALYSIS & DESIGN
CODING
TESTING
VDMTools®:
1200
500
600
ANALYSIS & DESIGN
CODING
TESTING
100%
0%
64%
Cost

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
- Example: Vending Machine
- 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

```
class <class-name>

  instance variables
    ...

  types
  values
  functions
  operations
    ...

  thread
    ...

  sync
    ...

end <class-name>
```

OO Paradigm (instance variables) — Internal object state

Functional paradigm (types, values, functions)

OO Paradigm (operations)

Definitions (types, values, functions, operations)

Dynamic behaviour (thread)

Synchronization control (sync)

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.).

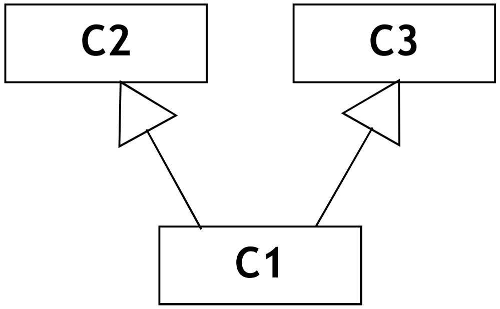
Inheritance

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

  class C1 is subclass of C2, C3

  ...

  end C1



- Usual semantics
- Polymorphism

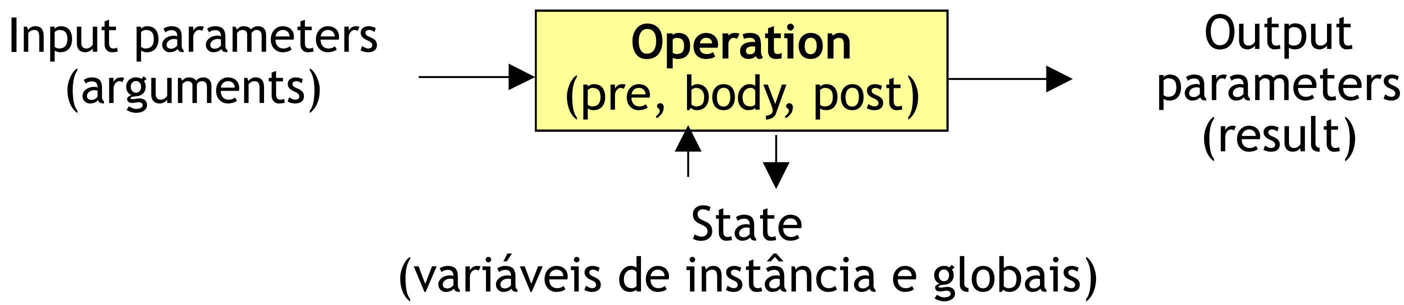
Instance variables

- 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:

argument type result type

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

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:

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

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

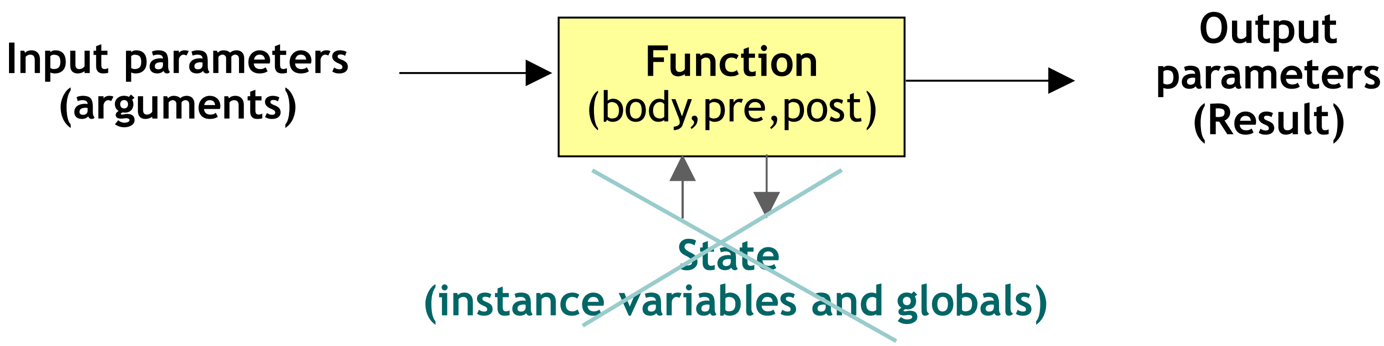
Predefined name for the return value

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 2.
- 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
```

Advanced functions in VDM++

- Polymorphic functions
- Higher-order functions
- The type function
- The lambda expression

Polymorphic functions

nomeFunção[@*TypeParam1*, @*TypeParam2*, …] …

- Are generic functions that can be used with different values
- They have special parameters (type parameters) that must be replaced by names of specific types using the function
- Names of these parameters start with "@" and are indicated in brackets after the function name
- Like function templates in C++

Polymorphic functions

- Example: utility function, which checks whether a sequence of elements of some kind has doubled:

```
public static HasDuplicates[@T](s: seq of @T) res: bool ==
      exists i, j in set inds s & i <> j and s(i) = s(j);
```

- Example of its use:

```
class Publication
  instance variables
    private autores: seq of Autor := [];
    inv not HasDuplicates[Autor](autores);
```

A concrete type

The function type

- Total function: *arg1Type * arg2Type * …* **+>** *resultType*
- Partial function: *arg1Type * arg2Type * …* **->** *resultType*

  It can be seen as a single package type argument tuple (instance calls of the product of types)

- The type of a Function is defined by the types of arguments and result
- Instances of a Function type (i.e., concrete function) can be passed as argument or as return, and saved (by reference) in data structures

Higher order functions

- Are functions that take other functions as arguments, or (Curried functions) that return functions as a result
- I.e. have arguments or a result of type function
- Example: a function that finds an approximate zero of a function between specified limits, with maximum error specified by the method of successive bisections:

```
findZero( f: real -> real , x1, x2, err: real) res: real ==
   if abs(x1 - x2) <= err and abs(f(x1) – f(x2)) <=  err then x1
   else let m = (x1 + x2) / 2
          in if sinal(f(m)) = sinal(f(x1)) then findZero(m,x2)
              else findZero(x1,m)
  pre sinal(f(x1)) <> sinal(f(x2));
```

The function type

Operator	Name	Type
f1 comp f2	Function composition	(B->C) * (A->B) -> (A->C)
f ** n	Function iteration	(A->A) * nat -> (A->A)

Operator name	Semantics Description
Function composition	It yields the function equivalent to applying first f2 and then applying f1 to the result.
Function iteration	Yields the function equivalent to applying *f* *n* times. *n*=0 yields the identity function which just returns the values of its parameter; *n*=1 yields the function itself. For *n*>1, the result of *f* must be contained in its parameter type.

The lambda expression

- lambda *patternArg1*: *Type1*, …, *patternArgN*: *TypeN* & *expr* λ

- Constructs a function on the fly
- Patterns are usually identifiers of arguments
- Normally used to pass as argument to another function (higher order)

Example: finding a real zero of a polynomial

```
findZero(lambda x: real & 5 * x**3 -  x**2 - 2 , 0, 1, 0.0000001)
```

Types

- Types are value types
  - Instances are immutable values pure
  - Comparison and assignment operate with their own values
  - Variable of type T name (a type) has its own data
- Subdivided into:
  - Basic types - bool, nat, real, char, ...
  - Constructed types (collections, etc..) - set of T, seq of T, map T1 to T2, ...
- New types can be defined within classes in the "types“ section
- The definition may include invariant for restricting valid instances
- Use to model types of values of attributes (data types)

Basic types

Symbol	Description	Examples of values
bool	Boolean	true, false
nat1	Natural number different from 0	1, 2, 3, …
nat	Natural number	0, 1, 2, …
int	Integer	…, -2, -1, 0, 1, …
rat **real**	Rational number Real number (the same as "rat" because only rational numbers can be represented in the computer)	-12.78, 0, 3, 16.23
char	Character	'a', 'b', '1', '2', '+', '-', …
token	Encapsulates a value (argument mk_token) of any type (useful if you know little about the type)	mk_token(1)
<identificador>	*Quotes* (literal names, typically used to define enumerated types)	<white>, <Black>

Constructed types - collections

Description	Sintax	Example of an instance
Set of elements of type A	set of A	{1 , 2}
Sequence of elements of type A	seq of A	[1, 2, 1]
Not empty sequence	seq1 of A	
Mapping elements of type A to type B elements (function finite set of key-value pairs)	map A to B	{ 0 |-> false, 1 |-> true }
Injected mapping (different key values correspond to different values)	inmap A to B	

More constructed types

Description	Sintax	Example of an instance
Products of types A, B, … (instances are tuples)	A * B * …	mk_(0, false)
Record T with fields *a*, *b*, etc. of types *A*, *B*, etc. (*)	T :: a : A b : B …	mk_T(0, false)
Union of types A, B, … (type A or type B or …)	A | B | …	
Opcional type (allows **nil**)	[A]	

(*) Alternative definitions :

```
T :: a  : A
     b :- B    -- field with ":-" is ignored in the comparison of records
```

Fields can be accessed by: **mk_T(x,y).b**

```
T ::  A  B      -- anonymous fields
```

Fields can be accessed by: **mk_T(x,y).#2**

Strings

- Not predefined type string, but can easily be defined as string (seq of char)
- All operations on sequences can be used with strings
- String literals can be indicated with quotation marks
  - "I am" is equivalent to ['I', ' ', 'a', 'm']

Example of a type definition

```
class Pessoa
  types
    public String = seq of char;
    public Date :: year : nat
                   month: nat
                   day  : nat;
    public Sexo = <Masculino> | <Feminino>;
  instance variables
    private nome: String;
    private sexo: Sexo;
    private dataNascimento: Date;
    ...
end Pessoa
```

sequence

record

Enumerated type (defined with *union* and *quote*)

attribute

Data type

The type reference

- Reference to class object
- Allows the modeling of associations between classes and work with objects of classes
- Example: :

```
class Pessoa
  instance variables
    private conjuge : [Pessoa];
    private filhos : set of Pessoa;
 ...
```

Guard reference to an object of class Person, or nil

Guard set of 0 or more references to objects of class Person

Symbolic constants

- Are constants which is given a name in order to make the specification more readable and easy to change
- Are declared in the section *values* with the syntax:

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

- Example:

```
values
    public PI = 3.1417;
```

Boolean Operators

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**

Numeric operators

-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
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

Operators on sets (set)

Operador	Nome	Descrição	Tipo
e in set s1	In	$e \in s1$	A * set of $A \to$ bool
e not in set s1	Not in	$e \notin s1$	
s1 union s2	Union	$s1 \cup s2$	set of A * set of $A \to$ set of A
s1 inter s2	Intersection	$s1 \cap s2$	
s1 \ s2	Difference	$s1 \setminus s2$	
s1 subset s2	subset	$s1 \subseteq s2$	set of A * set of $A \to$ bool
s1 psubset s2	proper subset	$s1 \subset s2$ $(s1 \subseteq s2 \wedge s1 \neq s2)$	
s1 = s2	equal	$s1 = s2$	
s1 <> s2	Not equal	$s1 \neq s2$	
card s1	Cardinal	$\# s1$	set of $A \to$ nat
dunion ss	Distributed union	$\bigcup_{s_i \in ss} s_i$	set of set of $A \to$ set of A
dinter ss	Distributed intersection	$\bigcap_{s_i \in ss} s_i$	
power s1	Set of sets	$\mathcal{P}(s1)$	set of $A \to$ set of set of A

Exercises (sets)

- {1,…,6}
- {1,…,1}
- {4,…,1}
- {x | x in set {2,3,4,5} & x > 2}
- {x | x in set {2,3,4,5} & 22 < x}
- {} in set power {1,3,6}
- dunion {{1,2},{1,5,6},{3,4,6}}
- dinter {{1,2},{1,5,6},{3,4,6}}
- {1,2,3} psubset {1,2}

Operators on sequences (seq)

Operador	Nome	Descrição	Tipo
hd l	Cabeça (*head*)	Dá o 1º elemento de l, que não pode ser vazia	seq of *A* → *A*
tl l	Cauda (*tail*)	Dá a subsequência de l em que o 1º elemento foi removido. l não pode ser vazia	seq of *A* → seq of *A*
len l	Comprimento	Dá o comprimento de l	seq of *A* → nat
elems l	Elementos	Dá o conjunto formado pelos elementos de l (sem ordem nem repetidos)	seq of *A* → set of *A*
inds l	Índices	Dá o conjunto dos índices de l, i.e., {1, ..., len l}	seq of *A* → set of nat1
l1 ^ l2	Concatenação	Dá a sequência formada pelos elementos de l1 seguida pelos elementos de l2	(seq of *A*) * (seq of *A*) → seq of *A*
conc ll	Concatenação distribuída	Dá a sequência formada pela concatenação dos elementos de ll (que são por sua vez sequências)	seq of seq of *A* → seq of *A*
l ++ m	Modificação de sequência	Os elementos de l cujos índices estão no domínio de m são modificados para o valor correspondente em m. Deve-se verificar: dom m subset inds l.	(seq of *A*) * (map nat1 to *A*) → seq of *A*
l(i)	Aplicação de sequência	Dá o elemento que se encontra no índice i de l. Deve-se verificar: i in set inds l.	seq of *A* * nat1 → *A*
l(i, ..., j)	Subsequência	Dá a subsequência de l entre os índices i e j, inclusive. Se i < 1, considera-se 1. Se j > len s, considera-se len(s).	seq of *A* * nat * nat → seq of *A*

Exercises (seq)

- Which of the following expressions are true?
- 6 in set elems [3,6,8,10,0]
- [] = tl [4]
- 6 in set inds [3,6,8,10,0]

Exercises (seq)

- 2.2 What are the results of the following expressions:
- tl [1,2,3]
- len [[1,2],[1,2,3]]
- hd [[1,2],[1,2,3]]
- tl [[1,2],[1,2,3]]
- elems [1,2,2,3,3,4]
- elems [[1,2],[2],[3],[3],[3,4]]

Exercises (seq)

- 2.3 What is the value of the following expressions
- len []
- len [1,2,3] + len [3]
- [hd [<A>,<B>]] ^ [hd [<C>,<D>]]
- tl [1,2,3,4,5] ^ [hd [1,2,2]]
- tl ([1,2]^[1,2])

Operators on finite functions (maps)

<table>
<tr><th>Operador</th><th>Nome</th><th>Descrição</th><th>Tipo</th></tr>
<tr><td>dom m</td><td>Domain</td><td>Gives the domain (key set) of m</td><td>map A to B → set of A</td></tr>
<tr><td>rng m</td><td>Co-domain (range)</td><td>Gives the co-domain (set of values corresponding to keys) of m</td><td>map A to B → set of B</td></tr>
<tr><td>m1 munion m2</td><td>Merge</td><td>Makes a union of key-value pairs exist in m1 and m2, which must be compatible (they can not match different values to equal keys)</td><td rowspan="2">(map A to B) * (map A to B) → map A to B</td></tr>
<tr><td>m1 ++ m2</td><td>Override</td><td>Union with unrestricted compatibility. In case of dispute, m2 prevails.</td></tr>
<tr><td>merge ms</td><td>Distributed union</td><td>Does the union of the mappings contained in ms that should be compatible.</td><td>set of (map A to B) → map A to B</td></tr>
</table>

Operators on finite functions (maps)

Operador	Nome	Descrição	Tipo
s <: m	Domínio restrito a	Dá o mapeamento constituído pelos elementos de `m` cuja chave está em `s` (que não tem de ser um subconjunto de `dom m`)	(set of A) * (map *A* to *B)* → map *A* to *B*
s <-: m	Domínio restrito por	Dá o mapeamento constituído pelos elementos de `m` cuja chave não está em `s` (que não tem de ser um subconjunto de `dom m`)	
m :> s	Contra-domínio restrito a	Dá o mapeamento constituído pelos elementos de `m` cujo valor de informação está em `s` (que não tem de ser um subconjunto de `rng m`)	(map *A* to *B*) * (set of *B*) → map *A* to *B*
m :-> s	Contra-domínio restrito por	Dá o mapeamento constituído pelos elementos de `m` cujo valor de informação não está em `s` (que não tem de ser um subconjunto de `rng m`)	

Operators on finite functions (maps)

Operador	Nome	Descrição	Tipo
m(d)	Aplicação de mapeamento	Dá o valor corresponde à chave `d` por `m`. A chave `d` deve existir no domínio de `m`.	(map *A* to *B*) * *A* → *B*
m1 comp m2	Composição de mapeamentos	Dá `m2` seguido de `m1`. O mapeamento resultante tem o mesmo domínio que `m2`. O valor correspondente a cada chave é obtido aplicando primeiro `m2` e depois `m1`. Restrição: `rng m2 subset dom m1`.	(map *B* to *C*) * (map *A* to *B*) → map *A* to *C*
m ** n	Iteração	Composição de `m` consigo próprio `n` vezes. Se `n=0`, dá a função identidade, em que cada elemento do domínio é mapeado para si próprio. Se `n=1`, dá `m`. Se `n>1`, `rng m` deve ser um subconjunto de `dom m`.	(map *A* to *A*) * nat → map *A* to *A*
inverse m	Mapeamento inverso	Dá o inverso de `m`, que deve ser injectivo.	inmap *A* to *B* → inmap *B* to *A*

Exercises (maps)

- dom {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
- rng {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
- {1000 |-> 3, 1005 |-> 4, 1002 |-> 1} ++ {1002 |-> 6}
- {1008 |-> 3, 1065 |-> 4, 1012 |-> 1} ++ {1011 |-> 6}
- {128} <: {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}
- {128} <-: {100 |-> <TIM>, 10 |-> <ROB>, 12 |-> <DAVE>}

Instructions/Expressions

- **Expressions:** return values
- **Instructions:** change system state, i.e., create, delete or change the state of objects (or the state of static variables) (e.g., assignment)
- For the model to be executable, you must write the body of transactions in the form of a statement or block of statements
- The body is also called "algorithmic body" because, while the postcondition specifies the "what" (effect), it is stated in the body "as" (algorithm)
- The VDM++ language allows to describe and test the algorithm to a high level of abstraction, refine it to the desired level, and generate an executable program in Java or C++ with the VDM Tools

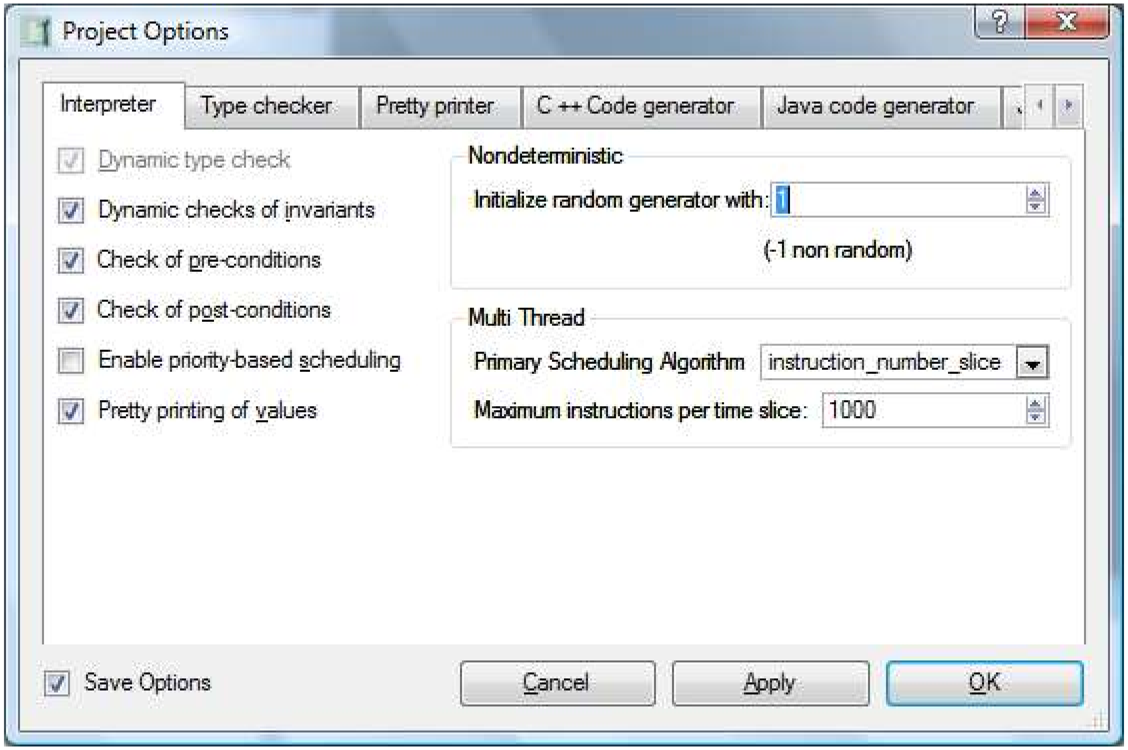
Expressions: examples

<table>
<tr><th>Expression</th><th>Example</th></tr>
<tr><td>Set enumeration</td><td>{a,3,3,true}</td></tr>
<tr><td>Set Comprehension</td><td>{a+2 | mk_(a,a) in set {mk_(true,1),mk_(1,1)}}</td></tr>
<tr><td>Set: type binding</td><td>{a | a: nat & a<10}</td></tr>
<tr><td>Set range</td><td>{3, …, 10}</td></tr>
<tr><td>Seq enumeration</td><td>[7.7, true, “I”,true]</td></tr>
<tr><td>Seq comprehension</td><td>[i*i | i in set {1,2,4,6}]</td></tr>
<tr><td>Subsequence</td><td>[4,true,”string”,9,4](2,…,4)</td></tr>
<tr><td>Map enumeration</td><td>{1|->true, 7|->6}</td></tr>
<tr><td>Map comprehension</td><td>Ex1: {i|->mk_(i,true) | i: bool}
Ex2: {a+b|-> b-a | a in set {1,2}, b in set {3,6}}</td></tr>
<tr><td>Tuple</td><td>mk_(2, 7, true, {|->})</td></tr>
</table>

Expressions: examples

Expression	Example
lambda	Ex1: **lambda** n: **nat** & n * n Ex2: **lambda** s: **nat**, b: **bool** & **if** b **then** a **else** 0
skip	**if** a <> [] **then** str := str ^ a **else skip** **--** Para indicar que nenhuma acção foi executada.
error	**if** a = <OK> **then** DoSomething() **else error** **--** O resultado é indefinido pelo que ocorreu um erro.
nondeterministic	|| (stmt1, stmt2, ..., stmtn)
iota	**iota** bind **&** expression -- it returns the unique value which satisfies the body expression e.g., iota x in set {sc1,sc2,sc3,sc4} & x.team = <France>

Nondeterministic



Exercises

dom {mk_(1,2).#1 |-> 3, mk_(2,3).#2 |-> 4}

[[5,6],[3,1,1],[5]] ++ {2 |-> [5,5],3 |-> [8]}

{mk_(x,y) | x in set elems ([1,2,2,1] ^ [2]), y in set inds [0,1] & x<=y}

{x|->y | x in set dom ({1|->2,2|->3} :> {3}), y in set rng {1|->4} & y = x*2}

conc ([[1,2],[2],[3,2]] ++ {1|->[3]})

{1|->2,2|->1,4|->4} munion ({1|->1,2|->2} ++ {1|->2,2|->1,3|->1})

Expressions: examples

Expression	Example
new	new C()
self	Create: **nat** ==> C Create (n) == (a := n; **return self**) -- Inicializa um objecto de uma classe com uma variável de instância -- *a* de tipo *nat* e guarda a referência para esse objecto
is	**Ex1:** is_nat(5) **Ex2:** is_C(mk_C(5))
isofclass	**isofclass**(Class_name,object_ref) -- Retorna true se *object_ref* é da classe *Class_name* ou uma -- subclasse da classe *Class_name*.
isofbaseclass	**isofbaseclass**(Class_name,object_ref) -- Para que o resultado seja true, *object_ref* tem que ser da classe -- *Class_name*, e *Class_name* não pode ter superclasses.
sameclass	**sameclass**(obj1, obj2) -- true se e só se *obj1* e *obj2* são instâncias da mesma classe
samebaseclass	**samebaseclass**(obj1, obj2)
Object reference	o operador **=** só retorna true se os dois objectos são a mesma instância; o operador **<>** retorna true quando os objectos não são a mesma instância, mesmo que tenham os mesmos valores nas variáveis de instância.

Expressions: examples

Expression	Example
forall	**forall** bind list **&** expression e.g., forall x in set elems l & m<=n
exists	**exists** bind list **&** expression e.g., exists x in set elems & x<5
exists1	**exists1** bind list **&** expression e.g., exists1 x in set elems & x<5

Instructions: examples

<table>
<tr><th>Instrução</th><th>Exemplo</th></tr>
<tr><td>let</td><td>**let** cs’ = {c |-> cs(c) **union** {s}}, ct’ = {s |-> ct(s) **union** {c}}
in sub_stmt1</td></tr>
<tr><td>let be</td><td>**let** i **in set inds** l **be st** Largest (**elems** l, l(i))
in sub_stmt2</td></tr>
<tr><td>define</td><td>**def mk_**(r,-) = OpCall()
in (x := r / 2; **return** x)
-- Allows binding of the result of an operation call to a pattern</td></tr>
<tr><td>if-then-else</td><td>**if** i = 0
then return <Zero>
elseif 1 <= i **and** i <= 9
then return <Digit>
else return <Number></td></tr>
<tr><td>cases</td><td>**cases** a:
mk_A(a’,-,a’) -> Expr(a’),
mk_A(b.b.c) -> Expr2(b,c),
others -> Expr3()
end</td></tr>
<tr><td>assign</td><td>x := 5</td></tr>
</table>

Instructions: examples

Instrução	Exemplo
block	(**dcl** a: **nat** := 5;**dcl** b: **bool**; stmt1; ...; stmtn) -- Se *stmt1* returna um valor, a execução do bloco termina e esse -- valor é retornado como resultado de todo o bloco.
loop	Ex1: **for** id = lower **to** upper [**by** step] **do** stmt Ex2: **for all** pat **in set** setexpr **do** stmt Ex3: **for all** pat **in seq** seqexpr **do** stmt
while	**while** expr **do** stmt

always	(**dcl** mem: Memory;
	  always Free(mem) **in**
	    (mem := Allocate();
	      Command(mem, …)))

return	**return** expr or **return**
exit	**exit** expr -- para sinalizar excepção

Instructions: examples

<table>
<tr><th>Instrução</th><th>Exemplo</th></tr>
<tr><td>exception handling</td><td>Ex1: **trap** pat **with** ErrorAction(pat) **in**
(**dcl** mem: Memory;
always Free(mem) **in**
(mem := Allocate();
Command(mem, ...)))
Ex2:
DoCommand : () ==> int
DoCommand () ==
(dcl mem : Memory;
always Free(mem) in
(mem := Allocate();
Command(mem, ...)
)
);
Example : () ==> int
Example () ==
tixe
{ <NOMEM> |-> return -1,
<BUSY> |-> DoCommand(),
err |-> return -2 }
in
DoCommand()</td></tr>
</table>

Blocks and variable declarations

```
(
      dcl id1 : tipo1 [:= expr1],  id2 : tipo2 [:= expr2], ...;
      dcl ... ;
      ...
      instruction1;
      instruction2;
      ...
)
```

- A block must have at least one instruction
- Variables may be declared only at the beginning of the block
- Last statement does not need ";"
- The 1st statement that return a value (even without a "return", just call one operation that returns a value) is finishing the block

Assignment

state variable := expression

- State variable name
  - Instance variable of the object in question
  - Static variable (static)
  - Local variable of the transaction (stated with dcl)
- Part of variable of type map, record or seq
  - map_var(key) := valor
  - seq_var(indice) := valor
  - record_var.field := valor

STOP An identifier introduced with let , forall, etc. is not a variable in this sense

Multiple assignment

atomic (*sd1* := *exp1*; *sd2* := *exp2*; …)

- First evaluates all expressions on the right side and then assigns them to the variables at the left side at once!
- Checks invariants at the end of all attributions (otherwise, it would check invariants after each attribution)

- Useful in the presence of invariants involving more than one instance variable (of the same object)
- It does not solve the problem of inter-object invariants, i.e., involving multiple objects (why?)

Multiple assignment - example

```
instance variables
  private quantidade : real;
  private precoUnitario : real;
  private precoTotal : real;
  inv precoTotal = quantidade * precoUnitario;
operations
```

Breaks invariant after 1st atribution

```
public SetQuantidade(q: real) ==
  (quantidade := q;  precoTotal := precoUnitario * q);
```

```
public SetQuantidade(q: real) ==
  atomic(quantidade := q; precoTotal := precoUnitario * q);
```

wrong: uses old value of the variable

```
public SetQuantidade(q: real) ==
  atomic(quantidade:=q; precoTotal:=precoUnitario * quantidade);
```

Instructions “let” and “def”

let *definition1*, *definition2*, … in *instruction*

let *identifier* in set *Set* [be st *condition*] in *instruction*

def *definition1*, *definition2*, … in *instruction*

Have the same form as expressions "let" and "def", with instruction rather than expression in the “in” part

Using "def" instead of "let", when in the definitions part are invoked operations that change state

Identifiers introduced in the definition are not variables that can change the value (can not appear on the left side of assignments)!

Instructions “if” and “cases”

```
if condition then instruction1 [else instruction2]

cases expression:
        pattern11, pattern12, ..., pattern1N -> instruction1,
        ... -> ...,
        patternM1, patternM2, ..., patternMN -> instructionM,
        others -> instructionM1
      end
```

- Have the same form as the expressions "if" and "cases", with instructions instead of expressions

- In the "if" statement, the party of "else" is optional

Instruction “return”

```
return
```

- Used to end operations that do not return any value

```
return expression
```

- Used to complete transactions that return a value

Beware of return implicit: the 1st instruction to return a value (just call operation that returns value) does end the block

Expression: “new”

- Create object:
  - new *name-of-the-class*(*parmeters-constructor*)
- Delete object: automatic, like in Java and C#
  - Are automatically deleted when no longer referenced
  - What we can do is to explicitly remove an object or a collection by assigning nil dereference (obj_ref: = nil)
  - Prevents errors and simplifies the specification
  - In contrast, prevents to know that there are instances of a given class at any given time (in OCL is ClassName.allInstances)
- Modify state of the subject: see assignment operator

Syntactic aspects

- Comments begin with "--" and go to the ends of the line
- Distinction of uppercase and lowercase letters (case sensitive)
- Accents are partially supported, it is preferable not to use them
- To cite an instance member (instance variable or operation) of an object, we use the usual notation "object.member"
- To refer to a static member (variable, operation or static function), type or constant defined in another class, we use the notation “class`member”, not “classe.membro”
- Use "nil" and not "null"
- Use "self" and not "this“

Mathematical notation vs ASCII

$\cdot$	&
$\times$	*
$\leq$	<=
$\geq$	>=
$\neq$	<>
$\xrightarrow{o}$	==>
$\rightarrow$	->
$\Rightarrow$	=>
$\Leftrightarrow$	<=>

$\mapsto$	|->
$\underline{\triangle}$	==
$\uparrow$	**
$\dagger$	++
$\uplus_{m}$	munion
$\lhd$	<:
$\rhd$	:>
$⩤$	<-:
$⩥$	:->
$\subset$	psubset
$\subseteq$	subset
$\curvearrowright$	^
$\bigcap$	dinter
$\bigcup$	dunion
$\mathcal{F}$	power
$\ldots$-set	set of ...
$\ldots^{*}$	seq of ...
$\ldots^{+}$	seq1 of ...
$\ldots \xrightarrow{m} \ldots$	map ... to ...

$\ldots \xleftrightarrow{m} \ldots$	inmap ... to ...
μ	mu
$\mathbb{B}$	bool
$\mathbb{N}$	nat
$\mathbb{Z}$	int
$\mathbb{R}$	real
$\neg$	not
$\cap$	inter
$\cup$	union
$\in$	in set
$\notin$	not in set
$\wedge$	and
$\vee$	or
$\forall$	forall
$\exists$	exists
$\exists!$	exists1
λ	lambda
ι	iota
$\ldots^{-1}$	inverse ...

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
- Example: Vending Machine
- Concurrency in VDM++

Type invariants

- Following the definition of a type we can define an invariant, to restrict the valid instances (valid values)
  inv *pattern == predicate*
  - pattern does match with the value of the type in question
  - predicate is the restriction that the value must satisfy
- Usually the pattern is simply a variable, as in

```
types
    public Date :: year : nat1
                   month: nat1
                   day  : nat1
      inv d == d.month <= 12 and
               d.day <=  DaysOfMonth(d.year, d.month);
```

- But we can use more complex patterns, for example

```
inv mk_Date(y,m,d) == m <= 12 and d <=  DaysOfMonth(y, m);
```

State invariants

- Defined in section "instance variables", following the variable instance definition, with sintaxe

  inv *boolean_expression_in_instance_variables*;

- Restrict the valid values of instance variables
- In VDM++, the invariants are checked after each assignment
  - Assignment to instance variable of the same class of invariant!
- You can also group multiple tasks in a single atomic block, and check the invariant at the end
  - Necessary for invariants that relate different instance variables
- Are inherited by subclasses, which may include further restrictions
- The expression of an invariant should not have side effects (may invoke query operations but no state change)

Common types of invariants

- Restriction of the attributs' domain
- Unique key constraints
- Restrictions associated with cycles in the associations
- Time constraints (with dates, times, etc.).
- Restrictions due to elements derived (calculated or replicated)
- Rules (conditions) existence (of values or objects)
- Generic business restrictions
- Idiomatic constraints (UML structurally guaranteed but not guaranteed when transforming to VDM ++)

Common types of invariants

- Restriction of the attributes' domain

The interest rate on a loan is a percentage between 0 and 100%.

Empréstimo
taxaJuros: Percentagem

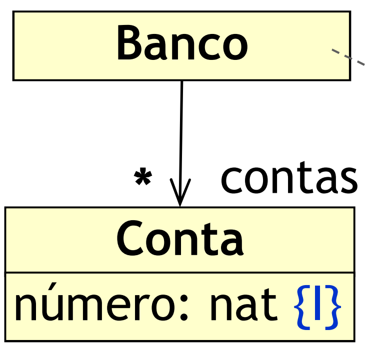
```
class Empréstimo
types
   Percentagem = real
     inv p == p >= 0 and p <= 100;
instance variables
   taxaJuros: Percentagem;
end Empréstimo
```

Usually it is better defined
by type invariant!

Common types of invariants

- Unique key constraint

A bank can not have two accounts with the same number

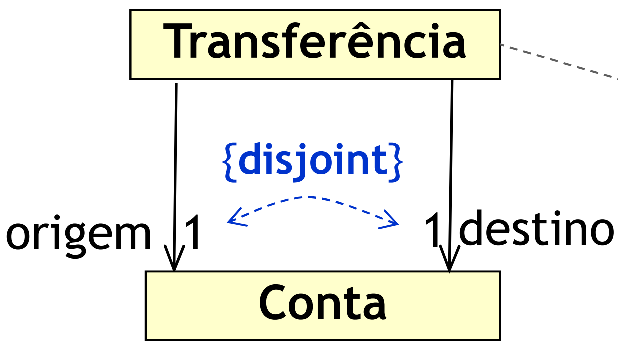



```
class Banco
instance variables
  contas: set of Conta;

  inv not exists c1, c2 in set contas &
          c1 <> c2 and c1.número = c2.número;

...
```

Common types of invariants

- Restrictions associated with cycles in the associations: *disjoint*

A transfer must be made between different accounts

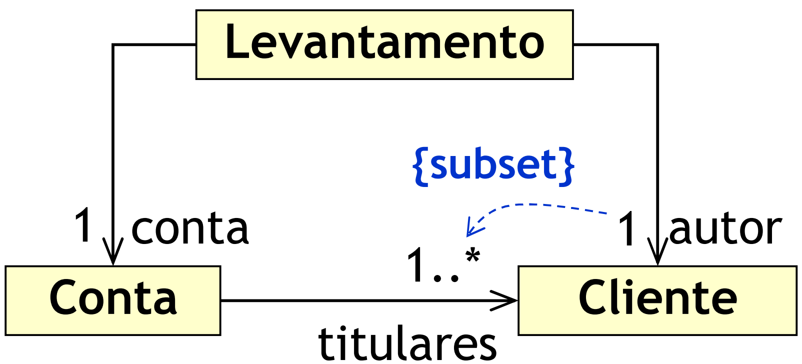



```
class Transferência
instance variables
  origem: Conta;
  destino: Conta;
  inv origem <> destino;
...
```

Common types of invariantes

- Restrictions associated with cycles in the associations: *subset*

  A withdraw can only be done by one of the account holders

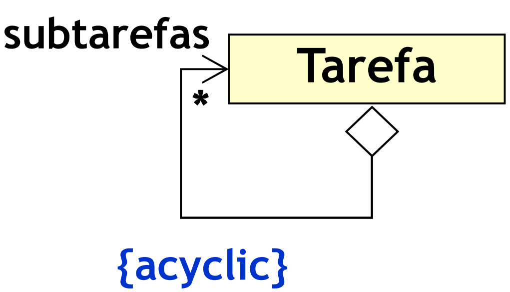



```
class Levantamento
instance variables
  conta: Conta;
  autor: Cliente;
  inv autor in set conta.titulares;
...
```

Common types of invariants

- Restrictions associated with cycles in the associations : *acyclic*

  A task can not be the subtask itself



Defined so as not to get into infinite loop if there are cycles!

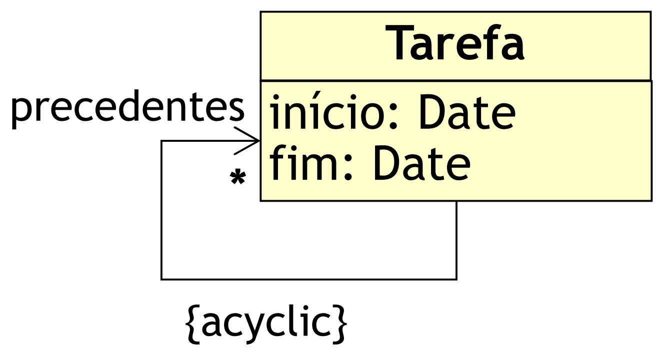
How to generalize to reuse?

```
class Tarefa
instance variables
  subtarefas: set of Tarefa;
  inv self not in set fechoTransitivoSubTarefas();

operations
  fechoTransitivoSubTarefas() : set of Tarefa ==  (
      dcl fecho : set of Tarefa := subtarefas;
      dcl visitadas : set of Tarefa := {};
      while visitadas <> fecho do
           let t in set (fecho \ visitadas) in (
                fecho := fecho union t.subtarefas;
                visitadas := visitadas union {t}
           )
      return fecho
  );
...
```

Common types of invariants

- Temporal restrictions

(1) A task can not finish before starting
(2) A task can not begin before the previous finishes

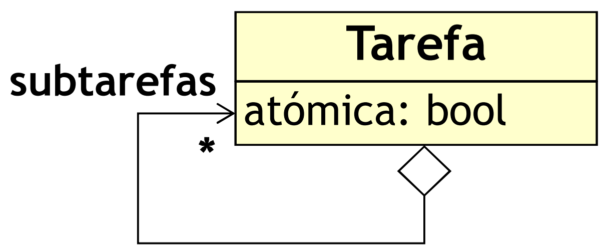



```
class Tarefa
types
  Date = nat; -- YYYYMMDD
instance variables
  início: Date;
  fim: Date;
  precedentes: set of Tarefa;

  inv  fim >= início;
  inv  forall p in set precedentes &
           self.início >= p.fim;
...
```

Common types of invariants

- Rules (conditions) existence (of values or objects)

An atomic task cannot have subtasks




```
class Tarefa
instance variables
  atómica: bool;
  subtarefas: set of Tarefa;

  inv  atómica => subtarefas = {};
...
```

Common types of invariants

- Rules (conditions) existence (of values or objects)

  You can not set the effective end of a task without defining its actual start

Attribute multiplicity

Tarefa
inícioEfectivo: Date [0..1] fimEfectivo: Date [0..1]

(UML)

ou

Tarefa
inícioEfectivo: [Date] fimEfectivo: [Date]

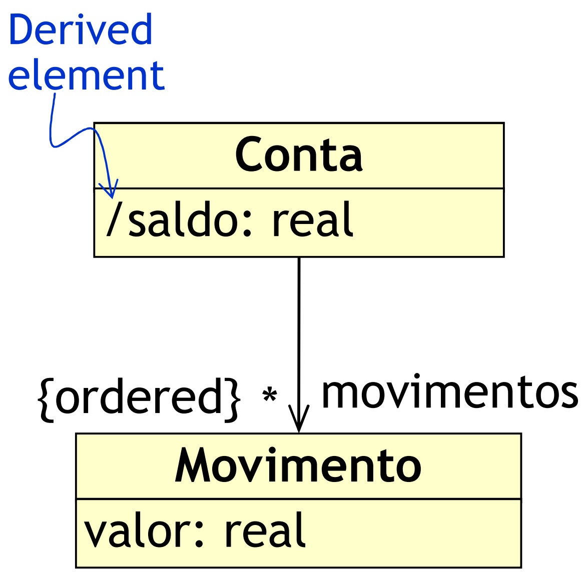
(VDM)

```
class Tarefa
instance variables
  inícioEfectivo: [Date];
  fimEfectivo: [Date];

  inv  fimEfectivo <> nil  =>
            inícioEfectivo <> nil;
...
```

Common types of invariants

- Restrictions on derived elements: Attributes

The account balance is equal to the sum of the movements from the opening of the account (negative in the withdraws)

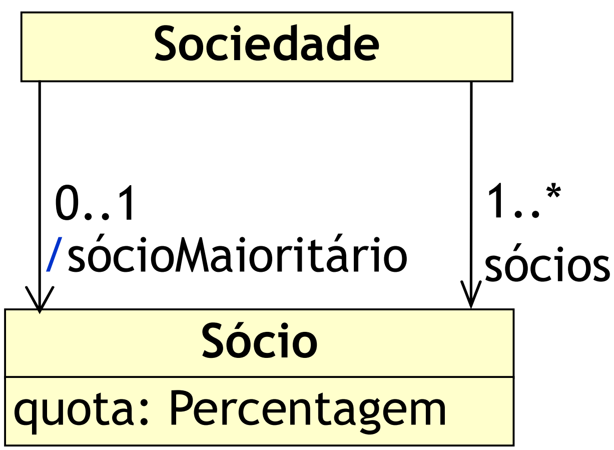



```
class Conta
instance variables
  saldo: real;
  movimentos: seq of Movimento;

  inv saldo = sum(movimentos);

functions
   sum(s: seq of real) res : real ==
      if s = [] then 0
      else let x = hd s in x.valor + sum(tl s);
...
```

Common types of invariants

- Restrictions on derived elements: associations

The main shareholder is the one with a market share exceeding 50%




```
class Sociedade
instance variables
  sócios: set of Sócio;
  sócioMaioritário: [Sócio];

  inv sum(sócios) = 100;

  inv if exists1 s in set sócios & s.quota > 50
        then sócioMaioritário =
                     iota s in set sócios & s.quota > 50
        else sócioMaioritário = nil;
...
functions
 public sum(s: set of Sócio) res: nat1 ==
  if s = {} then 0 else
    let x in set s in x.quota + sum(s\{x})
...
```

Common types of invariants

- Generic business rules

The account balance can not be negative

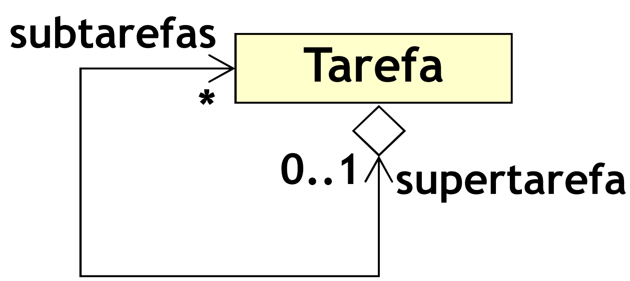
Conta
saldo: real

```
class Conta
instance variables
  saldo: real;

  inv saldo >= 0;

...
```

Common types of invariants

- Idiomatic restrictions (1)

A bidirectional association is represented in VDM++ for two unidirectional associations with associated integrity constraints




```
class Tarefa
instance variables
  subtarefas: set of Tarefa;
  supertarefa: [Tarefa];

  inv forall t in set subtarefas &
          t.supertarefa = self;

  inv supertarefa <> nil =>
          self in set supertarefa.subtarefas;
```

Both invariants are needed (why?)

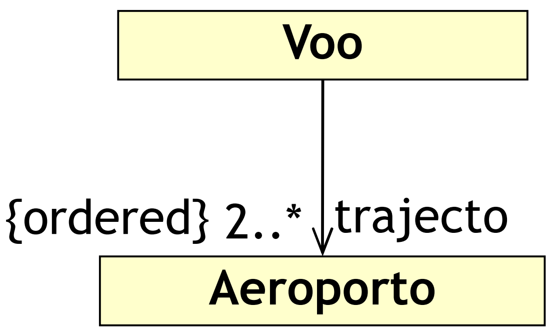
It can be seen as a case of cycle associations

Common types of invariants

- Restrições idiomáticas (2)

  VDM++ não tem nativamente colecções ordenadas sem repetições (OrderedSet em OCL)

  Restrições de multiplicidade podem originar invariantes




```
class Voo
instance variables
  trajecto: seq of Aeroporto;

  inv not exists i, j in set inds trajecto &
                 i <> j and trajecto(i) = trajecto(j);

  inv len trajecto >= 2;
...
```

To which class should associate each invariant?

- Both in VDM++ as OCL, the invariants have to be formalized within a class
- In the case of invariants which refer to only one class, the decision is trivial
- In the case of invariants involving more than one class, is a decision to "design" is not trivial
  - class where the expression is simpler
  - class where you have access to all information
  - class where there are operations that can violate the invariant

Limitation of VDM++: invariants inter-object

```
class A
  instance variables
    private x : nat;
    private b : B;
    inv x < b.GetY();
  operations
   public SetXY(newX, newY: nat) == (
      x := newX;
      b.SetY(newY)
   )
end A
```

```
class B
  instance variables
    private y : nat;
  operations
    public GetY() res: nat ==
       return y;
    public SetY(newY: nat) ==
       y := newY;
end B
```

1) Invariant is tested here (too soon), there is no way to delay check w / end of the block!

2) Invariant is not tested here, is set for another class!

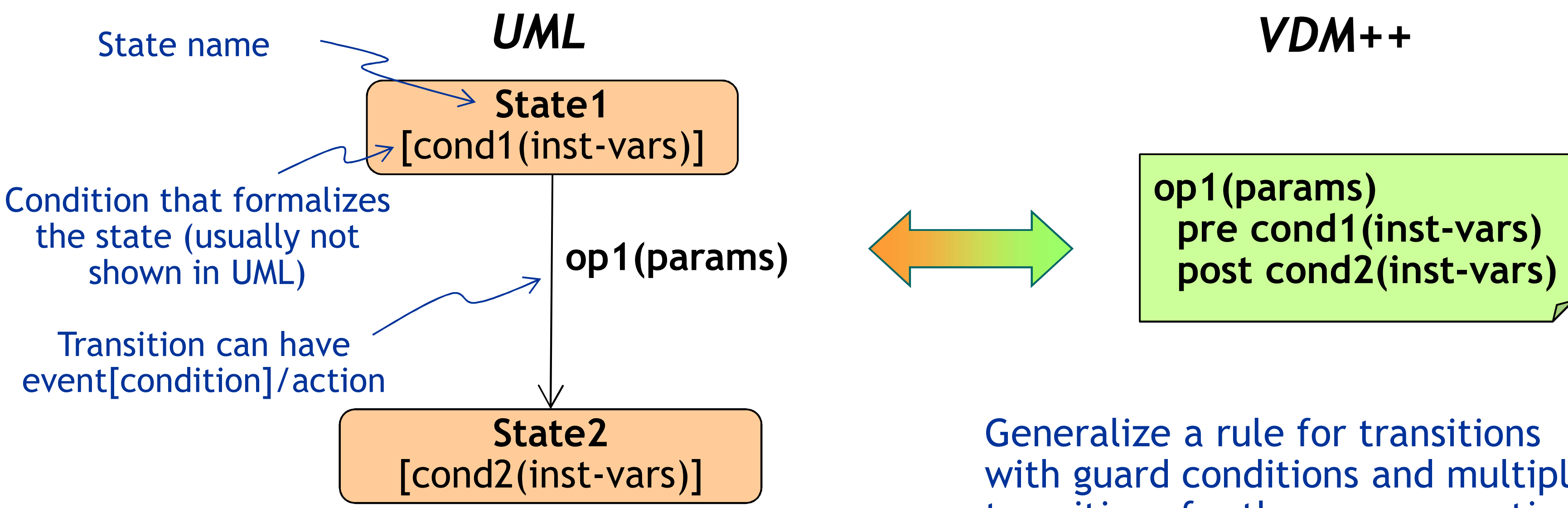
Other languages (OCL, Spec #, etc..) solve the problem of verifying an invariant only within the limits of method calls!

Pre and post conditions of operations

- Pre-condition: restricts the conditions call (values of instance variables and arguments of the object)
  - Correspond in the defensive lineup validations made at the beginning of the methods (with the possible launch of exceptions)
- Post-condition: formalizes the effect of the operation in a condition that relates the final values of instance variables and the value returned to the initial values of instance variables (indicated by ~) and the values of the arguments
- The pre-and post-conditions of the builder, along with default values of instance variables, must ensure the establishment of invariants, among other effects
- The pre-and post-conditions of operations, should ensure the preservation of invariants (assuming that the object checks the invariants at the beginning, it also checks at the end), among other effects

Relation to UML state diagrams

- State diagram is associated with a class and describes the life cycle and reactive behavior of each object class (in response to events call transactions or other to do later)
- Provides dynamic integrity constraints (valid transitions) for pre- and post-conditions of operations



Generalize a rule for transitions with guard conditions and multiple transitions for the same operation!

Limitations of VDM++

- You can only access the initial value of instance variables of the object (self)
- You can not access the baseline (old) of:
  - Instance variables of referenced objects
  - Instance variables inherited from superclasses
  - Query operations
  - Static variables

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
- Example: Vending Machine
- Concurrency in VDM++

Model validation

- **Validation** is the process of increasing confidence that the model is a faithful representation of the system under consideration. There are two aspects to consider:

1. Checking the internal consistency of the model.

2. Verify that the model describes the expected behavior of the system under consideration.

Properties of formal integrity

- **Satisfiability** (existence of solution)
  - ∃ combination of final values of instance variables and return value satisfying the postcondition, ∀ combinations of initial values of instance variables and arguments obeying the invariant and the precondition
- **Determinism** (uniqueness of solution)
  - If there is a requirement saying so, write a deterministic postcondition (which admits a unique solution)
  - But, for example, in a optimization problem, the postcondition can restrict admissible solutions without coming to impose a single solution
- **Preservation of invariants**
  - If initial values of instance variables and arguments obey to invariant and precondition, the postcondition ensures invariant at the end
  - After ensuring that all operations comply with the invariant, you can deactivate your check (heavier than incremental verification of pre / post conditions)
- **Protection of partial operators**
  - Inclusion of pre-conditions that define the value domain in which the operators can be called.

Internal consistency: proof obligations

- The collection of all verifications on a model are called VDM Proof Obligations. A proof obligation is a logical expression that should be true before considering the built VDM model formally consistent.

- We must consider three obligations of proof in VDM models:
  - Verification of domains (use of partial operators)
  - Satisfiability of explicit definitions
  - Satisfiability of implicit definitions

Related to the use of invariants

Domain verification

- The use of a partial operator outside its domain is considered an error performed by the modeler. There are two types of buildings that can not be automatically checked:
  - apply a function that has a pre-condition, and
  - apply a partial operator
- Some definitions:

  f:T1 * T2 * … * Tn -> R

  f(a1,…,an) == ...

  pre ...
- May refer the precondition of *f* as a Boolean function with the following signature:
  - pre_*f*:T1 * T2 * … * Tn -> bool

Domain verification

- if a function *g* uses an operator *f*: T1 * ... * Tn -> R in its body, occurring as an expression *f*(a1, ..., n), then it is necessary to show that the precondition of *f*

  pre-f(a1,…,an)

- is satisfied for all a1,…,an occurring in that position.
- Example:

```
AnalyseInput: Gateway -> Gateway
AnalyseInput(g) ==
  if Classify(hd g.input) = <High>
  then mk_Gateway(tl g.input,
                  g.outHi ^ [hd g.input],
                  g.outLo)
  else mk_Gateway(tl g.input,
                  g.outHi,
                  g.outLo ^ [hd g.input])
```

- Proof obligation for domain verification:
  - **forall g:Gateway & pre_AnalyseInput(g) => g.input <> []**

Domain verification

- The operators may be protected by partial pre-conditions :

```
AnalyseInput: Gateway -> Gateway
AnalyseInput(g) ==
   if Classify(hd g.input) = <High>
   then mk_Gateway(tl g.input,
                   g.outHi ^ [hd g.input],
                   g.outLo)
   else mk_ Gateway(tl g.input,
                   g.outHi,
                   g.outLo ^ [hd g.input])
pre g.input <> []
```

Now, the prove obligation

forall g:Gateway & pre_AnalyseInput(g) => g.input <> []

is verified

pre_AnalyseInput(g) == g.input <> []

Domain verification

- Alternatively, an operator can be partially protected including an explicit check in the function body, e.g.,:

```
AnalyseInput: Gateway -> [Gateway]
AnalyseInput(g) ==
  if g.input <> []
  then if Classify(hd g.input) = <High>
     then mk_Gateway(tl g.input,
                g.outHi ^ [hd g.input],
                g.outLo)
     else mk_ Gateway(tl g.input,
                   g.outHi,
                   g.outLo ^ [hd g.input])
  else nil
```

- If one includes this check, it must return a special value to indicate error and ensure that the return type of function is optional (to deal with return nil).

Domain verification

- It can be difficult to decide what to include in a pre-condition.
  - Some conditions are determined by requirements.
  - Many conditions are conditions to ensure the proper functioning of operators and partial functions.

When defining a function, you should read it systematically, highlighting the use of partial operators, and ensuring that there is no misuse of these operators by adding the appropriate set of preconditions

Invariant preservation

- All functions must ensure that the result is not only structurally of the correct type, but also that it is consistent with the invariant associated with its type.

- All operations must ensure that the invariants in the instance variables and in the result types are verified

- Formally, the preservation of invariant should be checked on all inputs that satisfy the preconditions of functions and operations

- Example

```
AddFlight: Flight ==> ()
AddFlight (f) ==
      journey := journey ^ f
pre journey(len journey).destination = f.departure
```

Satisfiability of explicit functions

- Explicit function without pre-condition set

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

said to be satisfiable if for all inputs, the result defined by the function body is of the correct type. Formally,

```
forall p1:T1,…,pn:Tn & f(p1,…,pn) : R
```

- An explicit function with precondition :

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

said to be satisfiable if for all inputs that satisfy the precondition, the result defined by the function body is of the correct type. Formally,

```
forall p1:T1,…,pn:Tn &
  pre_f(p1,…,pn) => f(p1,…,pn) : R
```

Satisfiability of implicit functions

- A function f defined implicitly as

```
f(a1:T1,…,an:Tn) r:R
pre ...
post ...
```

- said to be satisfiable if for all inputs that satisfy the precondition, there is a result of the correct type that satisfies the postcondition. Formally,

```
forall p1:T1,…,pn:Tn &
  pre_f(p1,…,pn) =>
    exists x:R & post_f(p1,…,pn,x)
```

E.g.,

```
f(x: nat) r:nat
pre x > 3
post r > 10 and r < 10
```

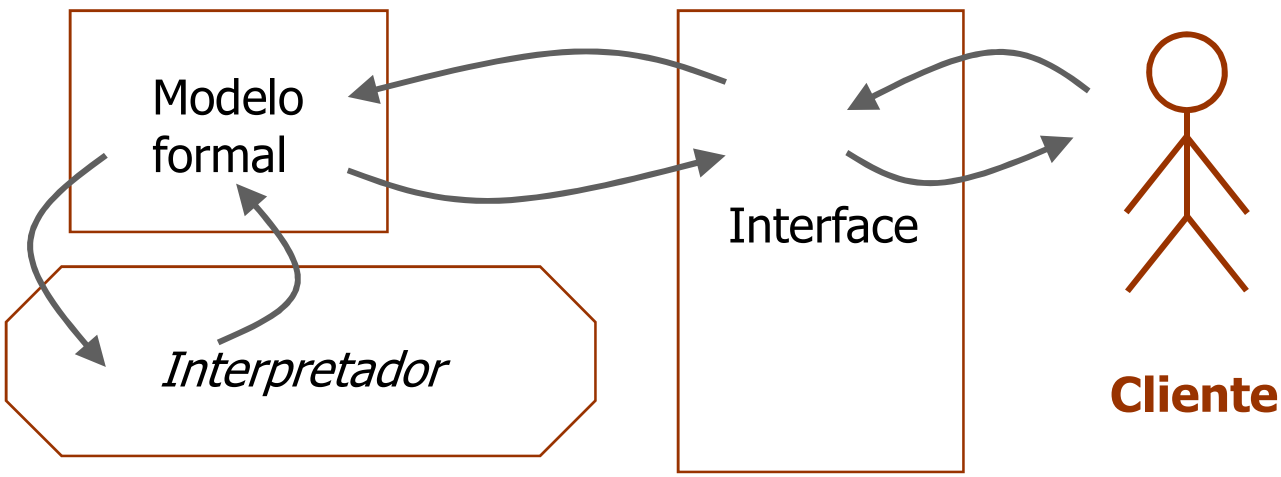
If it is not possible to find a result of type `nat` which satisfies `post_f` then `f` is not satisfiable

Behavior

- Another aspect of model validation is to ensure that it actually describes the expected behavior of the system under consideration.
- There are three possible approaches:
  - **Model animation** - works well with customers who are not familiar with modeling notations but requires a good user interface.
  - **Testing the model** - one can measure the coverage of the model but the results are limited to the quality of tests and the model has to be executable.
  - **Prove properties about the model** - ensures excellent coverage, does not require an executable model, but the tool support is limited.

Animation

- The model is animated via a user interface. The interface can be built in a programming language of choice considering it has the possibility of dynamic linking (Dynamic Link facility) for interconnection of the interface code to the model.



Testing

- The level of trust earned with the animation of the model depends on the particular set of scenarios that he decided to run on the interface.

  However, it is possible a more systematic test :
  - Define the collection of test cases
  - Perform these tests in a formal model
  - Compare the result with the expected

- Test cases can be generated manually or automatically. Automatic generation can produce a wide range of test cases.

- Techniques for generating test cases on functional programs can also be applied to formal models.

Proof

- Systematic testing and animation are only as good as the tests and scenarios used. Proof allows the modeller to assess the behaviour of a model for whole classes of inputs in one analysis.
- In order to prove a property of a model, the property has to be formulated as a logical expression (like a proof obligation). A logical expression describing a property which is expected to hold in a model is called a validation conjecture.
- Proofs can be time-consuming. Machine support is much more limited: it is not possible to build a machine that can automatically construct proofs of conjectures in general, but it is possible to build a tool that can check a proof once the proof itself is constructed. Considerable skill is required to construct a proof - but a successful proof gives high assurance of the truth of the conjecture about the model.

Proof levels

- **"Textbook"**: argument in natural language supported by formulae. Justifications in the steps of the reasoning appeal to human insight ("Clearly …", "By the properties of prime numbers …" etc.). Easiest style to read, but can only be checked by humans.
- **Formal**: at the other extreme. Highly structured sequences of formulae. Each step in the reasoning is justified by appealing to a formally stated rule of inference (each rule can be axiomatic or itself a proved result). Can be checked by a machine. Construction very laborious, but yields high assurance (used in critical applications)
- **Rigorous**: highly structured sequence of formulae, but relaxes restrictions on justifications so that they may appeal to general theories rather than specific inference rules.

Summary

- Validation: the process of increasing confidence that a model accurately reflects the client requirements.
- Internal consistency:
  - **domain checking**: partial operations or functions with precondition
    - Protect with preconditions or if-then-else
  - **satisfiability** of explicit and implicit functions/operations
    - Ensure invariants are respected
- Checking accuracy:
  - animation
  - testing
  - proof

increase cost

increase confidence

Pre/Post-conditions and inheritance

- When you reset an operation inherited from the superclass, you should not violate the contract (pre-and post-condition) established in the super-class
- The precondition can be weakened (relaxed) in the subclass, but not strengthened (can not be more restrictive)
  - any call that is promised to be valid on the precondition of the superclass, must continue to be accepted as a precondition of the subclass
  - pre_op_superclass => pre_op_subclass
- The postcondition can be strengthened in the subclass but not weaker
  - operation in the subclass must still ensure the effects promised in the superclass and may add other effects
  - post_op_subclass => post_op_superclass
- Behavioral subtyping

Pre/Post-conditions and inheritance

```
class Figura
  types
    public Ponto :: x : real
                    y : real;

  instance variables
    protected centro : Ponto;

  operations

    public Resize(factor: real) ==
       is subclass responsibility
       pre factor > 0.0
       post centro = centro~;

 end Figura
```

```
class Circulo is subclass of Figura
 instance variables
   private raio : real;
   inv raio > 0;

 operations
   public Circulo(c: Ponto, r: real) res: Circulo
     == ( raio := r; centro := c; return self )
     pre r > 0;

   public Resize(factor: real) ==
     raio := raio * abs(factor)
     pre factor <> 0.0
     post centro = centro~ and
             raio = raio~ * abs(factor);
end Circulo
```

pre Figura`Resize(...) ⇒ pre Circulo`Resize(...)

post Figura`Resize(...) ⇐ post Circulo`Resize(...)

Specification testing

- A well-built specification already has built-in checks
  - Invariants, pre / post-conditions, other assertions (invariants of cycles, etc.).
- But it must be exercised in a repeatable manner with automated testing
  - The aim is to discover errors and gain confidence in the correctness of the specification
  - Later, the same tests can be applied to the implementation
- Testing with valid entries
  - Exercise all parts of the specification (measured coverage with VDMTools)
  - Use assertions to check return values and final states
  - (Op) Derive tests from state machines (test based on states)
  - (Op) Derive tests from usage scenarios (scenario-based test)
  - (Op) Derive tests axiomatic specifications (test based on axioms)
- Test with invalid entries
  - Break all invariants and pre-conditions to verify that work
  - …

Support for testing in VDM Tools

- Specification can be tested interactively with VDM++ interpreter, or based on test cases predefined
- You can enable automatic checking of invariants, pre conditions and post conditions
- For information on test coverage, you must define at least one test script
  - Each test script tsk is specified by two files:
    - tsk.arg file - the command to be executed by interpreter
    - tsk.arg.exp file - with the expected result of command execution
- VDMTools give information of the tests that have succeeded and failed
- Pretty printer "paints" the parts of the specification that were in fact executed and generates tables with % of coverage and number of calls

Simulation of assertions

Use

```
class TestPessoa is subclass of Test
    operations
        public TestNome() == (
            dcl j : Pessoa := new Pessoa("João", ...);
            Assert( j.GetNome() = "João")
        )
end TestPessoa
```

Definition

```
class Test
    operations
      protected Assert : bool ==> ()
        Assert(a) == return
      pre a
end Test
```

Assertion violation is reduced to violation of pre-condition (enable verification of pre-conditions in VDMTools)

Test-Driven Development com VDM++

- Principles :
  - Write tests before the implementation of the functionality (in each iteration)
  - Develop small iterations
  - Automate testing
  - Refactor to remove code duplication
- Advantages of TDD:
  - Ensuring quality of tests
  - Thinking in particular cases before considering the general case
    - test cases are partial specifications
  - Complex systems that work result from the evolution of simpler systems that work

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
- Example: Vending Machine
- Concurrency in VDM++

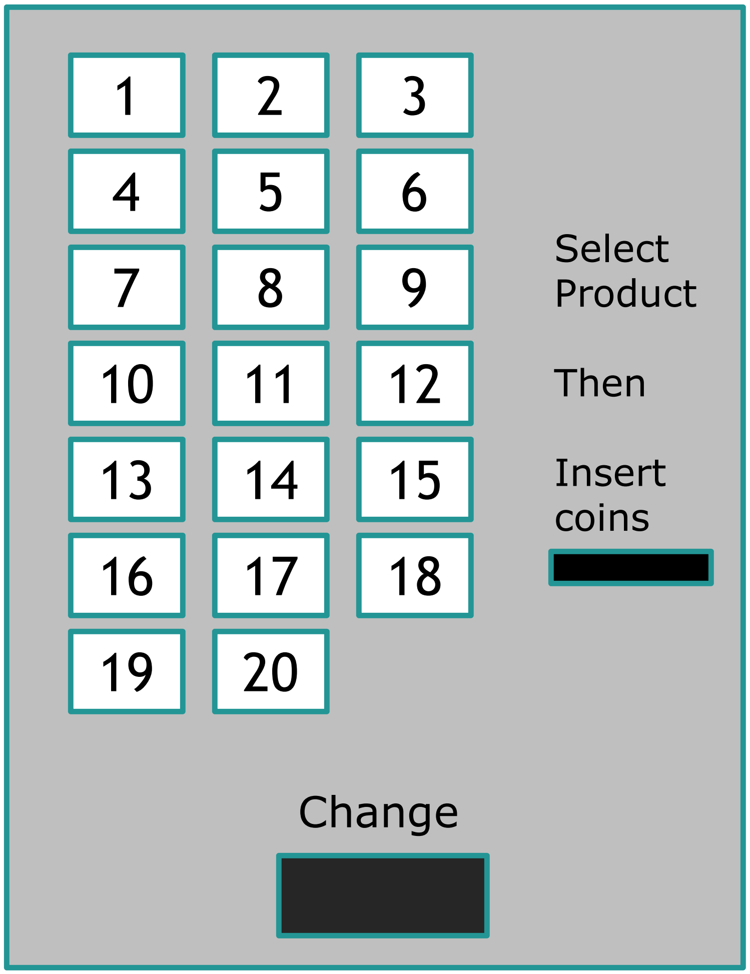
Vending Machine

- Build a VDM++ model of a vending machine for products like drinks, snacks, cookies, etc. This machine accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro.
- There is a stock of coins inside the vending machine which is used to give the due change to clients who may need it.
- The vending machine has also a stock of products and each one has its price.
- The Product Vending Machine provides different services in two different possible states: in configuration or waiting for user interaction;

While in configuration: it is possible to update the stock of products and coins;
While waiting:

- The vending machine should show the products available to the clients.
- The clients should select the product they want and then insert euro coins to pay for it;
- The vending machine should give change every time it is possible (there are coins in stock for that purpose); otherwise it should give back all the money introduced by the client without selling the product.

Vending machine



a) Accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro

b) Each of the 20 boxes can have at most 15 units.

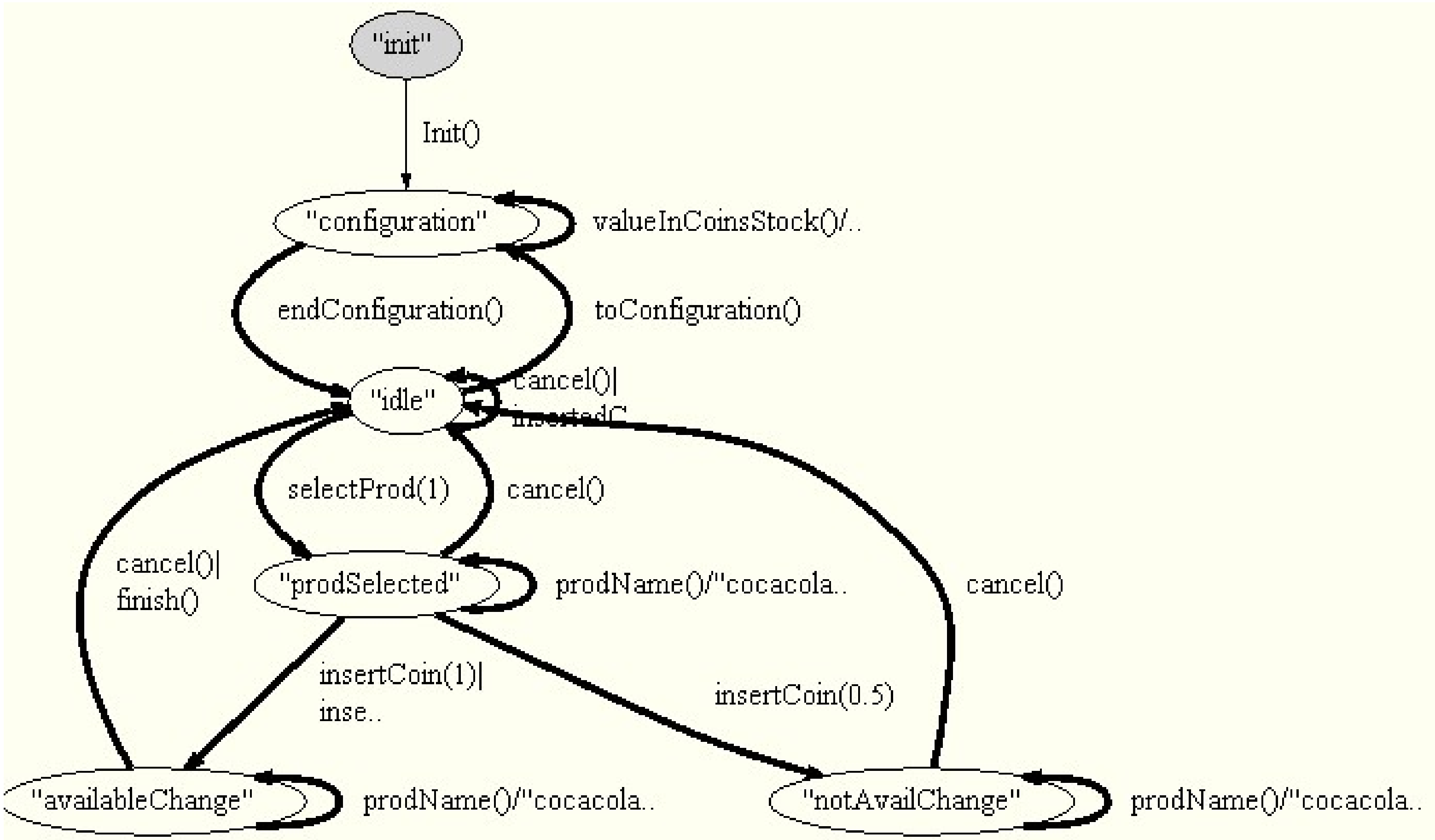
c) Each product has a price

d) different possible states: configuration, idle, init, prodSelected, availableChange, notAvailChange

e) Different operations available depending on the current state.

f) If there is no change, it is possible to cancel

Vending Machine



Vending Machine – constants and types

```
values
    public Capacity: real = 15;

types
    public String = seq of char;
    public Coins = nat
        inv c == c in set {100, 50, 20, 10, 5};
    public Boxes = nat
        inv b == b in set {1,...,20};

    public Products :: name: String
                       quantity : nat1
                       price : nat1;

    public ProductsInBoxes = map Boxes to [Products]
        inv pb ==
            forall b in set dom pb & pb(b) = nil or
              pb(b)<>nil => pb(b).quantity <= Capacity;

    public State = <init> | <configuration> | <idle> | <prodSelected> |
                   <availableChange> |  <notAvailChange>;
```

```
instance variables

  public stockProd: ProductsInBoxes;
  public stockCoins: map Coins to nat;
  public stateMachine: State := <init>;
  public prodSelected: [Boxes] := nil;
  public insertedCoins: seq of Coins := [];
  public coinsTroco: seq of Coins := [];
```

Vending Machine – operations/functions

operations

Construtor

```
public VendingMachine : () ==> VendingMachine
VendingMachine () ==
(
   stateMachine := <configuration>;
    stockProd := { |-> };
    stockCoins := { |-> };
)
pre stateMachine = <init>;
```

Refills machine with a product in a certain position. Replaces what was previously in the same position. Assume the machine in configuration.

```
public SetStockProducts (num: Boxes, name: String, price: nat1, quant: nat1) ==
   stockProd := stockProd ++ {num |-> mk_Products(name, quant, price)}
 pre stateMachine = <configuration> and
                         quant <= Capacity and price mod 5 = 0;
```

Vending Machine – operations/functions

Changes the stock of coins. Assume the machine in configuration.

```
public SetStockCoins(novoStockCoins: map Coins to nat) ==
    stockCoins := stockCoins ++ novoStockCoins
pre stateMachine = <configuration>;
```

Reads the quantity in stock of a product by number.

```
public GetStockProducts(number: Boxes) res : nat1 ==
   return stockProd(number).quantity
pre stateMachine = <configuration> and number in set dom stockProd;
```

Reads the price of a product by the number.

```
public GetPriceProduct(number: Boxes) res : nat1 ==
   return stockProd(number).price
pre stateMachine = <configuration> and number in set dom stockProd;
```

End of setup

```
public EndConfiguration() ==
  stateMachine := <idle>
pre stateMachine = <configuration>;
```

Vending Machine – operations/functions

Select a product by its number. After selecting a product you can insert coins.

```
public SelectProduct(number : Boxes) ==
  prodSelected := number
pre stateMachine = <idle> and number in set dom stockProd and
      stockProd(number) <> nil;
```

To find out if you have not entered enough coins yet.

```
public InsertedCoinsValue() res : nat ==
(
   dcl sum:nat := 0;
   for all e in set inds insertedCoins do sum:=sum+insertedCoins(e); return sum;
)
pre prodSelected <> nil;
```

Inserts a currency of a certain value. There should be a selected product, and coins of sufficient value should not have been inserted.

```
public InsertCoin(c: Coins) ==
  insertedCoins := insertedCoins ^ [c]
pre prodSelected <> nil and InsertedCoinsValue() < stockProd(prodSelected).price;
```

Vending Machine – operations/functions

Asks / See the name of the selected product

```
public GetNomeProdSel() res : String ==  return stockProd(prodSelected).name
pre prodSelected <> nil;
```

Calculates the value of the coins that are part of the change

```
public Sum(troco: seq of Coins) res : nat ==
 (      dcl sum:nat := 0;
        for all e in set inds troco do sum := sum + troco(e);
        return sum;
);
```

Cancels the current purchase

```
public Cancelar() ==
 ( prodSelected := nil;  insertedCoins := []; coinsTroco := []  )
pre prodSelected <> nil;
```

Simulates user to pick the selected product

```
public RecolheProduto() == (
        if (stockProd(proSelected).quantity<>1) then
           stockProd(prodSelected).quantity := stockProd(prodSelected).quantity – 1 else
           stockProd := {prodSelected} <-: stockProd ;
        coinsTroco := [];
)
pre prodSelected in set dom stockProd;
```

Vending Machine – operations/functions

What is the total value of the coins in stock?

```
public SumMap(x:map Coins to nat) res:nat
        (dcl sum: nat := 0;
        for all e in set dom x do sum:=sum+e*x(e);
        return sum;)
```

Calculate the change (coins with equal value to change)

```
public calcularTroco () res: map Coins to nat
( ...
  exists m:map Coins to nat &
        SumMap(m) = InsertedCoinsValue()- stockProd(prodSelected).price
        and forall e in set dom m & e in set dom stockCoins and
                        stockCoins(e) >= m(e);
  ...
);
```

Not executable!

Vending Machine – operations/functions

If possible gives change to the client

```
public GiveChange() res: seq of Coins ==
(
   dcl sortedCoins: seq of Coins := [100,50,20,10,5];
   dcl troco : nat := InsertedCoinsValue() – stockProd(prodSelected).price;
   coinsTroco := [];
   while (sortedCoins <> []) do (
     if (hd sortedCoins in set dom stockCoins and stockCoins(hd sortedCoins) > 0 and
        hd sortedCoins <= troco)
     then (
        coinsTroco := coinsTroco ^ [hd sortedCoins];
        if (stockCoins(hd sortedCoins)=1) then stockCoins := {hd sortedCoins} <-: stockCoins
        else stockCoins(hd sortedCoins) := stockCoins(hd sortedCoins)-1;
        troco := troco – hd sortedCoins;
     )
     else sortedCoins := tl sortedCoins
   );
   if (troco <> 0) then (
      for all e in set elems coinsTroco do stockCoins(e) := stockCoins(e)+1;
      coinsTroco := [];
   );
   return coinsTroco;
 )
pre self.prodSelected <> nil and self.InsertedCoinsValue() > self.stockProd(self.prodSelected).price;
```

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
- Example: Vending Machine
- Concurrency in VDM++

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

Concurrency in VDM++

- 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.
- 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.

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.

Concurrency and sincronization in VDM++

- Concurrency: through the definition of ***active objects*** that can be owners of ***threads***
  - Classes of work objects and ***thread*** section where thread behavior is specified
  - ***Start*** statement starts ***thread*** on a previously created object
  - Two types of theads: ***simple*** and ***periodic***

Concurrency and sincronization in VDM++

- Synchronization: through **synchronization restrictions** on the access to shared objects (typically **passive**)
  - Synchronization constraints are defined declaratively
  - Allow limiting concurrency between active objects/threads
  - Constraints are indicated in the ***sync*** section of the class definition
  - Two types of restrictions/predicates: permission and mutual exclusivity

    sync per operation_name => guard-condition

    mutex (op1, op2)
  - Constraints are inherited by subclasses

Simple Threads

- **thread *statement(s)***
  - Section on the definition of the class that indicates the instruction (usually an operation) or instruction sequence to be performed by the thread
  - The thread dies when the execution of that instruction(s) is complete
- **start(*objRef*)**
  - Instruction used to start a thread on the indicated object
  - Thread does not start when you create the object to allow initialization
  - Called again (even without finishing previous), starts new thread
- **startlist(*objRefSet*)**
  - Statement used to start a thread pool
- **threadid**
  - Natural number that uniquely identifies the current thread

Periodic Threads

- **thread periodic (*timeinterval*) (*opname*)**
  - Repeatedly performs the operation according to the specified time interval (in "system time units").
  - The operation must run in less than the time interval
  - Not supported by VDMTools Light

```
-- timer that periodically increments its clock, at every 1000 system time units
class Timer
 instance variables
   private curTime : nat := 0;
  operations
   private IncTime() == curTime := curTime + 1;
   public GetTime() res: nat == return curTime;
 thread
    periodic(1000)(IncTime)
 end Timer
```

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.

- Ex.: Permission guards reflecting the bounding of the buffer :

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

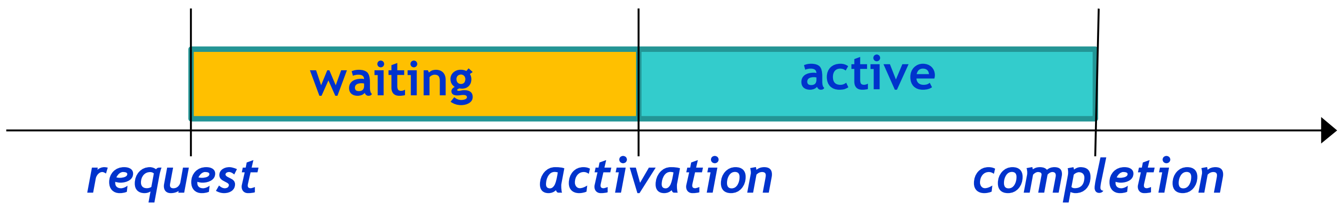
Guard conditions (1)

- **per operation-name => guard-condition**
  - Specifies condition to check in order to perform the operation
  - If it is not verified at the time of the operation's call, the operation is put on hold
- Guard condition can use instance variable values, as well as values of operations execution counters (see below)
- Guard condition is **different** from a precondition
  - No pre-condition satisfaction is an error
  - No guard condition satisfaction only puts the call on hold

Guard conditions (2)

- Interpreter detects and warns of deadlock situations
- You can only specify one permission predicate per operation
- Rules for reassessment of guard conditions:
  - Occurs when the execution of an operation ends (on the same object)
  - Tests the condition and (potential) activation of the operation performed atomically
  - It is not defined which is the object whose expression of guard is reevaluated 1^{st}

Operations Execution Counters



Expression	Description
#act(op- name)	Number of times the operation was activated (started running) on this object.
#fin(op- name)	Number of times the operation was completed (finished running) on this object.
#active(op- name)	Number of operation calls that are currently active on this object. #active(op-name) = #act(op-name) - #fin(op-name)
#req(op- name)	Number of operation calls on this object.
#waiting(op- name)	Number of calls that are currently on hold on this object. #waiting(op-name) = #req(op-name) - #act(op-name)

Mutual exclusive predicates (mutex)

- **mutex(*op-name1*, *op-name2*, …)**
  - Operations can not run simultaneously (on the same object)
- **mutex(all)**
  - "All" refers to all operations defined in the class and superclasses
- The same operation may appear in multiple mutex predicates (and in a permission predicate)
- Mutex predicates are implicitly translated into permission predicates

```
mutex(opA, opB);
mutex(opB, opC, opD);
per opD => someVariable > 42;
```

```
per opA => #active(opA) + #active(opB) = 0;
per opB => #active(opA) + #active(opB) = 0 and
           #active(opB) + #active(opC) + #active(opD) = 0;
per opC => #active(opB) + #active(opC) + #active(opD) = 0;
per opD => #active(opB) + #active(opC) + #active(opD) = 0
            and someVariable > 42;
```

Example

standard VDM++ IO library

```
class Worker
  operations
    public doit() == (
      dcl io : IO := new IO();
      dcl rc : bool;
      for i = 1 to 40 do
          rc := io.fwriteval[nat * nat]("out.txt",
                        mk_(threadid, i), <append>)
    );
    public wait_done() == skip;
    public static main() == (
      dcl w1 : Worker := new Worker();
      dcl w2 : Worker := new Worker();
      start(w1);  start(w2);
      w1.wait_done();  w2.wait_done()
    );
  thread doit()
  sync per wait_done => #fin(doit) > #act(wait_done)
end Worker
```

With w1 again, it would also work

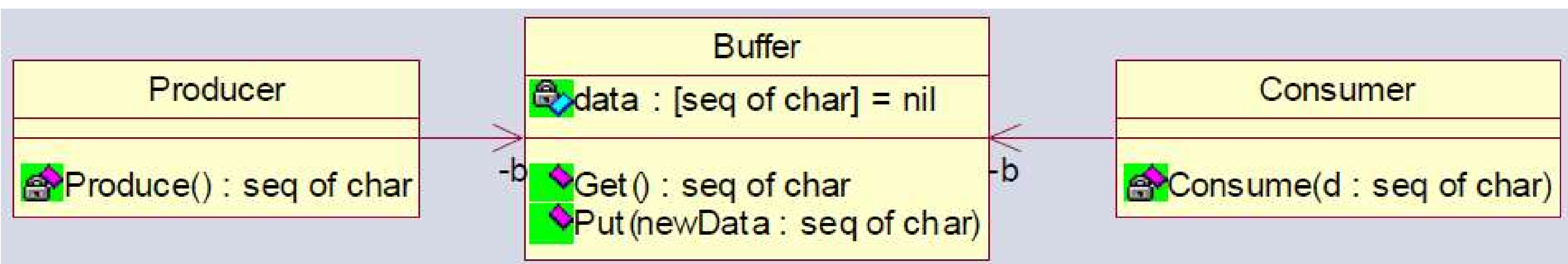
wait_done is called when ...

out.txt

mk_(2, 1)	mk_(3, 9)	mk_(2, 37)
mk_(2, 2)	mk_(3, 10)	mk_(2, 38)
mk_(2, 3)	mk_(3, 11)	mk_(2, 39)
mk_(2, 4)	mk_(3, 12)	mk_(2, 40)
mk_(2, 5)	mk_(3, 13)	mk_(3, 21)
mk_(2, 6)	mk_(3, 14)	mk_(3, 22)
mk_(2, 7)	mk_(3, 15)	mk_(3, 23)
mk_(2, 8)	mk_(3, 16)	mk_(3, 24)
mk_(2, 9)	mk_(3, 17)	mk_(3, 25)
mk_(2, 10)	mk_(3, 18)	mk_(3, 26)
mk_(2, 11)	mk_(3, 19)	mk_(3, 27)
mk_(2, 12)	mk_(3, 20)	mk_(3, 28)
mk_(2, 13)	mk_(2, 21)	mk_(3, 29)
mk_(2, 14)	mk_(2, 22)	mk_(3, 30)
mk_(2, 15)	mk_(2, 23)	mk_(3, 31)
mk_(2, 16)	mk_(2, 24)	mk_(3, 32)
mk_(2, 17)	mk_(2, 25)	mk_(3, 33)
mk_(2, 18)	mk_(2, 26)	mk_(3, 34)
mk_(2, 19)	mk_(2, 27)	mk_(3, 35)
mk_(2, 20)	mk_(2, 28)	mk_(3, 36)
mk_(3, 1)	mk_(2, 29)	mk_(3, 37)
mk_(3, 2)	mk_(2, 30)	mk_(3, 38)
mk_(3, 3)	mk_(2, 31)	mk_(3, 39)
mk_(3, 4)	mk_(2, 32)	mk_(3, 40)
mk_(3, 5)	mk_(2, 33)	
mk_(3, 6)	mk_(2, 34)	
mk_(3, 7)	mk_(2, 35)	
mk_(3, 8)	mk_(2, 36)	

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 Buffer class

```
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
```

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 Synchronized

Assuming the buffer does not lose data, there are two requirements:

1. It should only be possible to *get* data, when the producer has placed data in the buffer.
2. 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

- 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 counterintuitive. VDM++ provides the keyword mutex

  mutex(Put, Get)

- Shorthand for

  per Put => #active(Get) = 0

  per Get => #active(Put) = 0

* Standard IO library

- Include file **$TOOLBOXHOME/stdlib/io.vpp** in the project
- **writeval[type](value)**
  - Function that writes the value of the type indicated, in ASCII, on the standard output
  - Exemple: writeval[nat](20)
- fwriteval[tipo](ficheiro, valor, modo)
  - Function that writes the value of the indicated type, in ASCII, on the standard output
  - Mode can be <append> (add) or <start> (criate)
  - Exemple: fwriteval[nat]("output.txt", 20, <append>)
- echo(text)
  - Operation that writes the texto, possibly with escape sequences, on the standard output.
  - Exemple: echo(“ola\n")
- fecho(file, text, [mode])
  - The same but in file
- ferror()
  - All the previous functions/operations return false in case of error. This operation returns (and cleans) the string with the correspondent error message

Additional references

- *Applying Formal Specification in Industry*. P.G. Larsen, J. Fitzgerald and T. Brookes. Published in "IEEE Software" vol. 13, no. 3, May 1996
- *A Lightweight Approach to Formal Methods* S.Agerholm and P.G. Larsen. In Proceedings of the International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany, Springer-Verlag, October 1998.
- *Applications of VDM in Banknote Processing* P. Smith and P.G. Larsen. + *Application of VDM-SL to the Development of the SPOT4 Programming Messages Generator*, A. Puccetti and J.Y. Tixadou + *Formal Specification of an Auctioning System Using VDM++ and UML, M.Verhoef et. al*.

  Published at the First VDM Workshop: VDM in Practice with the FM'99 Symposium, Toulouse, France, September 1999.

Examples of VDM++ models on the web

- http://www.vdmportal.org/twiki/bin/view/Main/VDMPPexamples
- http://www.overturetool.org/?q=node/13
- http://www.vdmportal.org/twiki/pub/Main/VDMPPexamples/cashdispenser_a4.pdf