

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

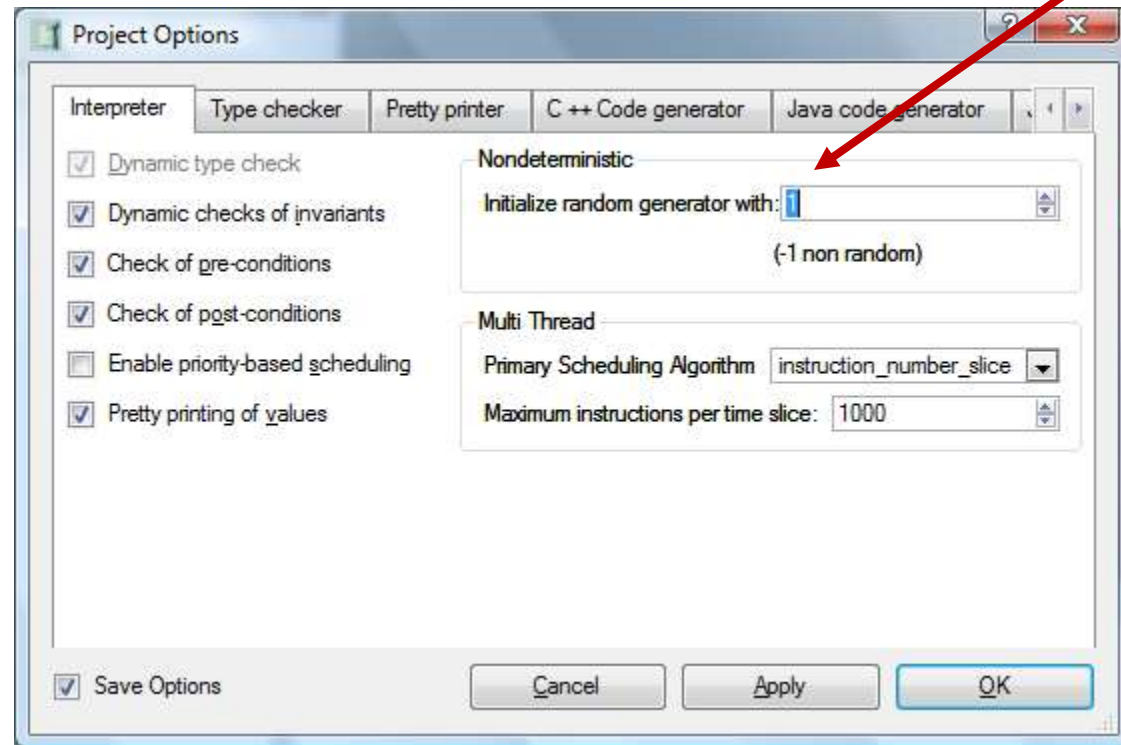
Expression	Example
Set enumeration	<code>{a,3,3,true}</code>
Set Comprehension	<code>{a+2 mk_(a,a) in set {mk_(true,1),mk_(1,1)}}</code>
Set: type binding	<code>{a a: nat & a<10}</code>
Set range	<code>{3, ..., 10}</code>
Seq enumeration	<code>[7.7, true, "l",true]</code>
Seq comprehension	<code>[i*i i in set {1,2,4,6}]</code>
Subsequence	<code>[4,true,"string",9,4](2,...,4)</code>
Map enumeration	<code>{1 ->true, 7 ->6}</code>
Map comprehension	Ex1: <code>{i ->mk_(i,true) i: bool}</code> Ex2: <code>{a+b -> b-a a in set {1,2}, b in set {3,6}}</code>
Tuple	<code>mk_(2, 7, true, { ->})</code>

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

To initialize
random
generator



Exercises

$\text{dom } \{\text{mk_}(1,2).\#1 \mapsto 3, \text{mk_}(2,3).\#2 \mapsto 4\}$

$\{1,3\}$

$[[5,6],[3,1,1],[5]] ++ \{2 \mapsto [5,5], 3 \mapsto [8]\}$

$[[5,6], [5,5], [8]]$

$\{\text{mk_}(x,y) \mid x \text{ in set elems } ([1,2,2,1] \wedge [2]), y \text{ in set inds } [0,1] \ \& \ x \leq y\}$

$\{ \text{mk_}(1, 1), \text{mk_}(1, 2), \text{mk_}(2, 2) \}$

$\{x \mapsto y \mid x \text{ in set dom } (\{1 \mapsto 2, 2 \mapsto 3\} \Rightarrow \{3\}), y \text{ in set rng } \{1 \mapsto 4\} \ \& \ y = x^2\}$

$\{ 2 \mapsto 4 \}$

$\text{conc } ([[1,2],[2],[3,2]] ++ \{1 \mapsto [3]\})$

$[3,2,3,2]$

$\{1 \mapsto 2, 2 \mapsto 1, 4 \mapsto 4\} \text{ munion } (\{1 \mapsto 1, 2 \mapsto 2\} ++ \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 1\})$

$\{ 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 4 \}$

Expressions: examples

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

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

Instructions: examples

Instrução	Exemplo
block	(dcl a: nat := 5; dcl b: bool; stmt1; ...; stmtn) -- Se <i>stmt1</i> retorna 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 exceção

Instructions: examples

Instrução	Exemplo
exception handling	<p>Ex1: trap pat with ErrorAction(pat) in (dcl mem: Memory; always Free(mem) in (mem := Allocate(); Command(mem, ...)))</p> <p>Ex2:</p> <pre> 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() </pre>

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*



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


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

Breaks invariant after 1st attribution 

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



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

wrong: uses old value of the variable 

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(parameters-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

.	&	\mapsto	\mapsto	$\overset{m}{\longleftrightarrow} \dots$	inmap ... to ...
\times	*	\triangle	==	μ	mu
\leq	<=	\uparrow	**	\mathbb{B}	bool
\geq	>=	\dagger	++	\mathbb{N}	nat
\neq	<>	\mathbb{E}	munion	\mathbb{Z}	int
$\overset{o}{\rightarrow}$	\Rightarrow	\triangleleft	<:	\mathbb{R}	real
\rightarrow	->	\triangleright	:>	\neg	not
\Rightarrow	=>	\triangleleft	<-:	\cap	inter
\Leftrightarrow	<=>	\triangleright	:->	\cup	union
		\cup	psubset	\in	in set
		\cup	subset	\notin	not in set
		\cup	^	\wedge	and
		\mathcal{F}	dinter	\vee	or
		...-set	dunion	\forall	forall
		...*	power	\exists	exists
		...+	set of ...	$\exists!$	exists1
		$\overset{m}{\rightarrow} \dots$	seq of ...	λ	lambda
			seq1 of ...	ι	iota
			map ... to ...	\dots^{-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 attributes' 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 |

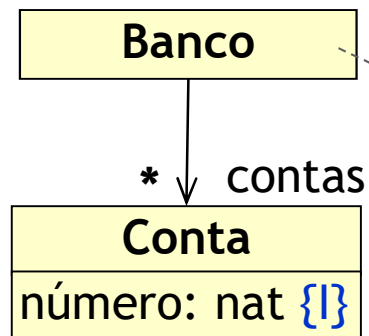
```
class Empréstimo
types
 Percentagem = real
 inv p == p >= 0 and p <= 100;
instance variables
 taxaJuros: Percentagem;
end Empréstimo
```

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;

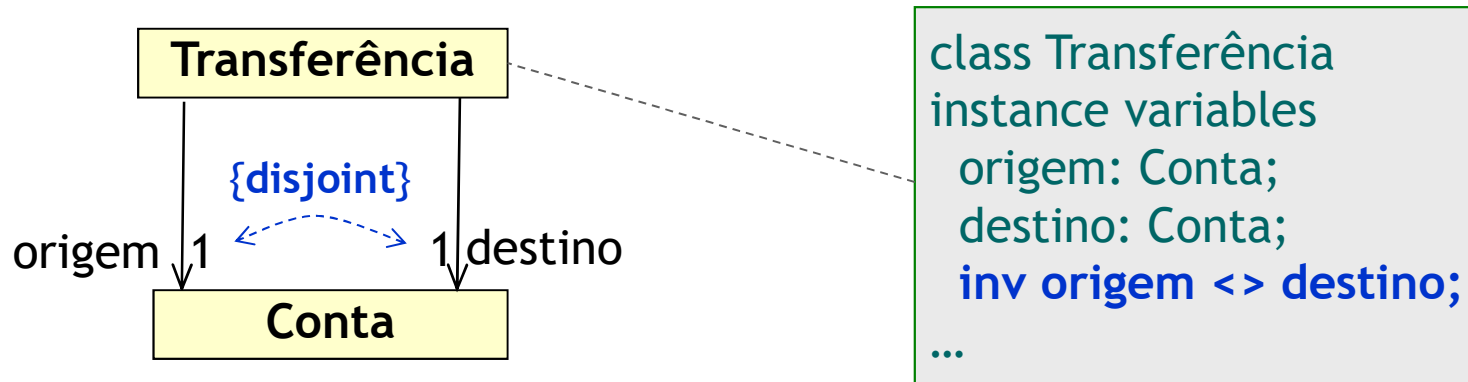
 inv not exists c1, c2 in set contas &
 c1 <> c2 and c1.número = c2.número;

 ...
```

# Common types of invariants

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

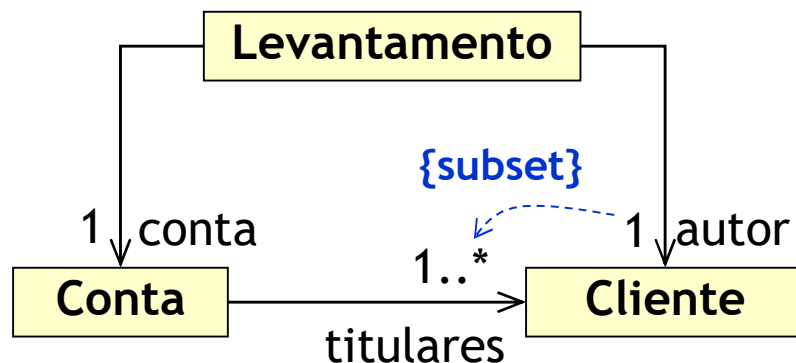
*A transfer must be made between different accounts*



# Common types of invariants

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

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

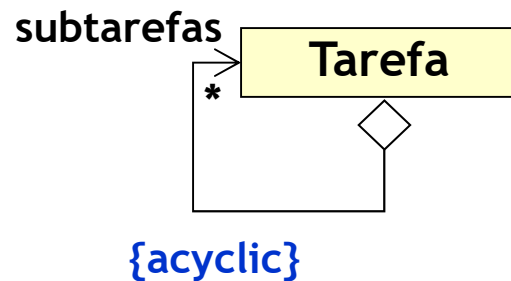


```
class Levantamento
instance variables
 conta: Conta;
 autor: Cliente;
 inv autor in set conta.titulares;
...
```

# 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();

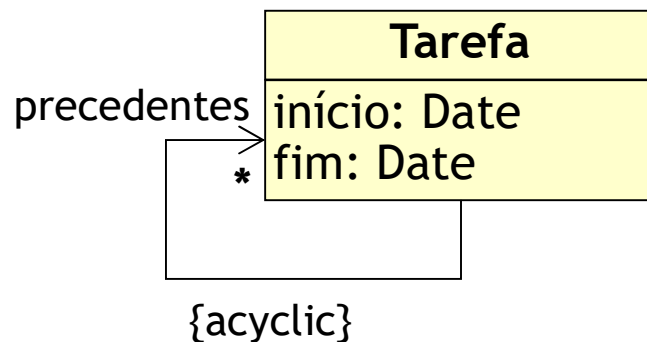
operations
 fechoTransitivoSubTarefas() : set of Tarefa == (
 dcl fecho : set of Tarefa := subtarefas;
 dcl visitadas : set of Tarefa := {};
 while visitadas <> fecho do
 let t in set (fecho \ visitadas) in (
 fecho := fecho union t.subtarefas;
 visitadas := visitadas union {t}
)
 return fecho
);
...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

# 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;

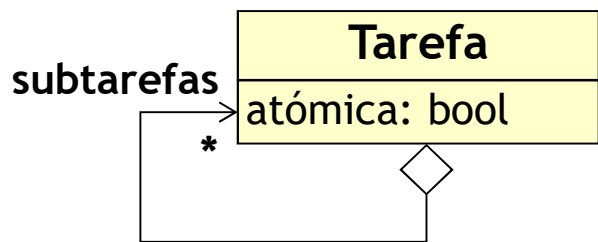
 inv fim >= início;
 inv forall p in set precedentes &
 self.início >= p.fim;

 ...
```

# Common types of invariants

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

*An atomic task cannot have subtasks*



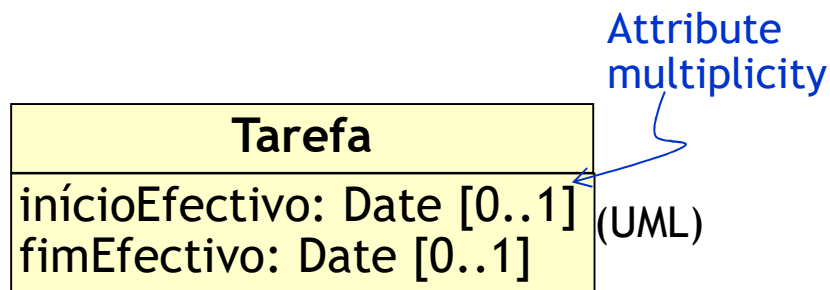
```
class Tarefa
instance variables
 atómica: bool;
 subtarefas: set of Tarefa;

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

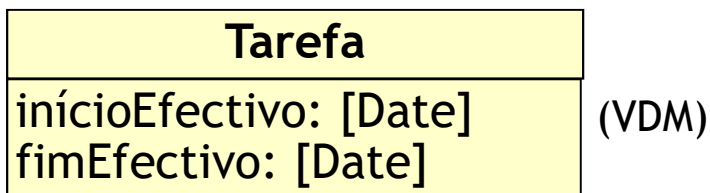
# Common types of invariants

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

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



ou



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

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

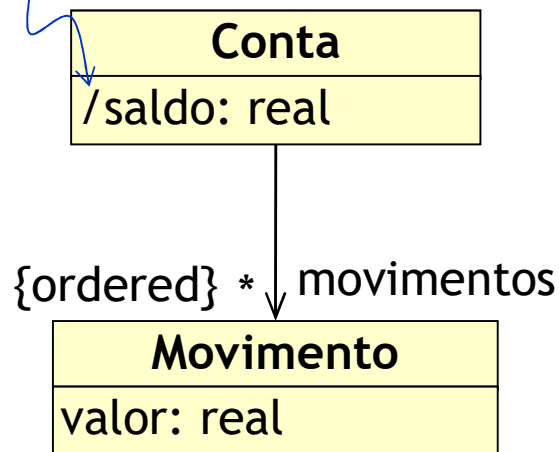


# Common types of invariants

## ◆ 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)*

Derived  
element



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

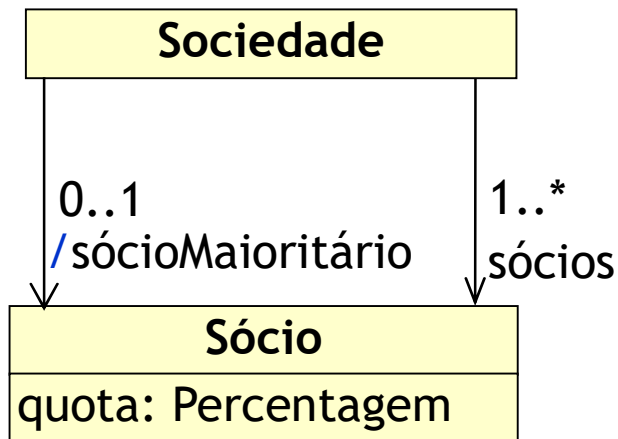
 inv saldo = sum(movimentos);

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;

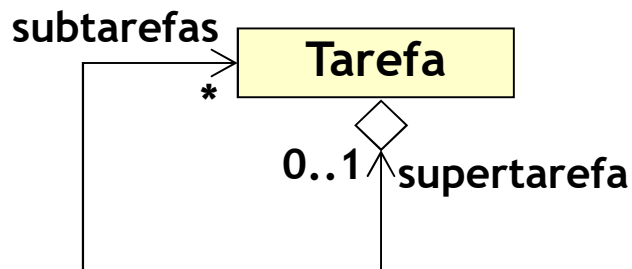
 inv saldo >= 0;

 ...
```

# Common types of invariants

## ◆ Idiomatic restrictions (1)

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



Both invariants are needed  
(why?)

It can be seen as a case of  
cycle associations

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

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

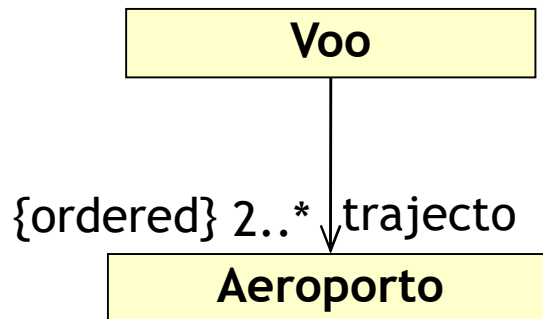
inv supertarefa <> nil =>
 self in set supertarefa.subtarefas;
```

# Common types of invariants

## ◆ Idiomatic restrictions (2)

*VDM++ does not have ordered collections without repetition (OrderedSet em OCL)*

*Restrictions on multiplicity may be the source of invariants*



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

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

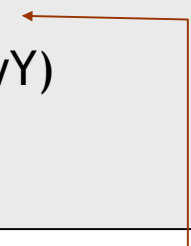
inv len trajecto >= 2;
...
```

## 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

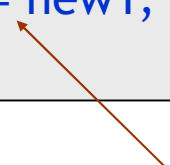
# Limitation of VDM++: invariants inter-object

```
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
```



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

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



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!

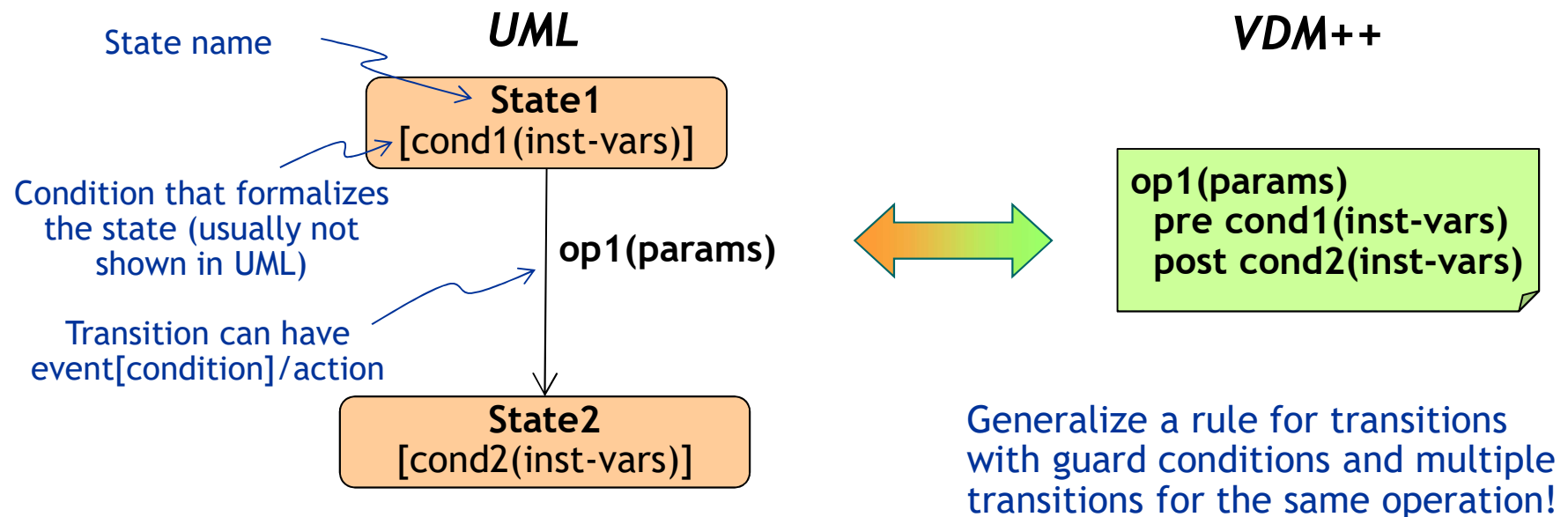
# Pre and post conditions of operations

- ◆ 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  $\sim$ ) 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



# 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 (1)

## ◆ Satisfiability (existence of solution)

- $\exists$  combination of final values of instance variables and return value satisfying the postcondition,  $\forall$  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

# Properties of formal integrity (2)

## ◆ 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:
$$f:T1 * T2 * \dots * Tn \rightarrow R$$
$$f(a1, \dots, an) == \dots$$
$$\text{pre } \dots$$
- ◆ May refer the precondition of  $f$  as a Boolean function with the following signature:
  - $\text{pre}_f:T1 * T2 * \dots * Tn \rightarrow \text{bool}$



# Domain verification

- ◆ if a function  $g$  uses an operator  $f: T_1 * \dots * T_n \rightarrow R$  in its body, occurring as an expression  $f(a_1, \dots, a_n)$ , then it is necessary to show that the precondition of  $f$   
 $\text{pre-}f(a_1, \dots, a_n)$
- ◆ is satisfied for all  $a_1, \dots, a_n$  occurring in that position.

- ◆ Example:

AnalyselInput: Gateway  $\rightarrow$  Gateway

AnalyselInput( $g$ ) ==

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 &  $\text{pre\_AnalyselInput}(g) \Rightarrow g.\text{input} \neq []$

# 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

$f:T_1*...*T_n \rightarrow R$   
 $f(a_1,...,a_n) == ...$

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

**forall  $p_1:T_1,...,p_n:T_n \ \& \ f(p_1,...,p_n) : R$**

## ◆ An explicit function with precondition

$f:T_1*...*T_n \rightarrow R$   
 $f(a_1,...,a_n) == ...$

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  $p_1:T_1,...,p_n:T_n \ \& \ pre\_f(p_1,...,p_n) \Rightarrow f(p_1,...,p_n) : R$**

# Satisfiability of implicit functions

- ◆ A function  $f$  defined implicitly as

$f(a_1:T_1, \dots, a_n:T_n) r:R$   
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  $p_1:T_1, \dots, p_n:T_n$  &  
pre\_f( $p_1, \dots, p_n$ ) =>  
exists  $x:R$  & post\_f( $p_1, \dots, p_n, x$ )

**E.g.,**

$f(x: \text{nat}) r:\text{nat}$   
pre  $x > 3$   
post  $r > 10$  and  $r < 10$

**If it is not possible to find a result  
of type  $\text{nat}$  which satisfies  
post\_f**

**then  $f$  is not satisfiable**

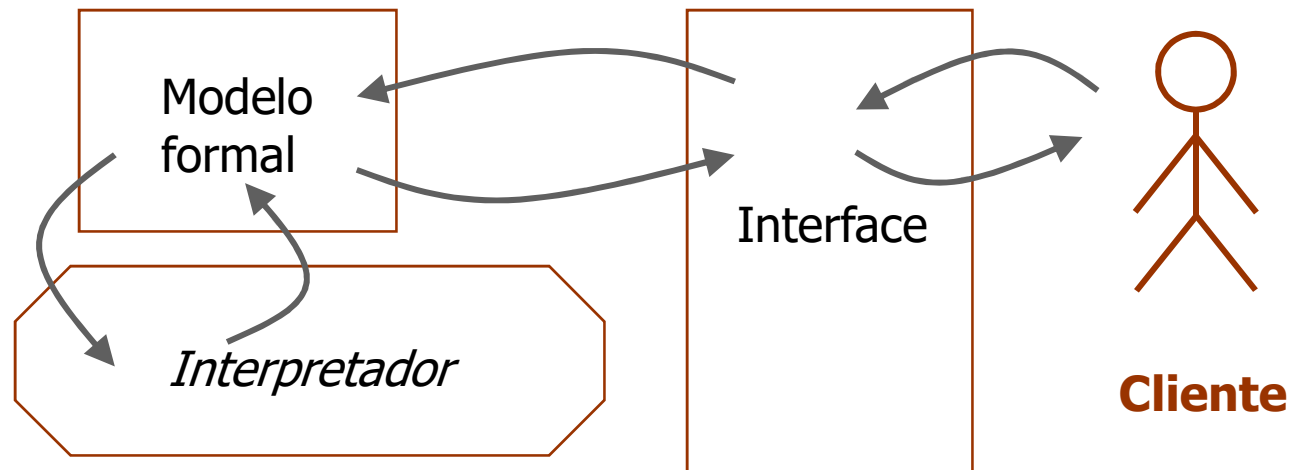
# Behavior

- ◆ 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
  - $\text{pre\_op\_superclass} \Rightarrow \text{pre\_op\_subclass}$
- ◆ 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
  - $\text{post\_op\_subclass} \Rightarrow \text{post\_op\_superclass}$
- ◆ 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(...)  $\Rightarrow$  pre Circulo `Resize(...)

post Figura `Resize(...)  $\Leftarrow$  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

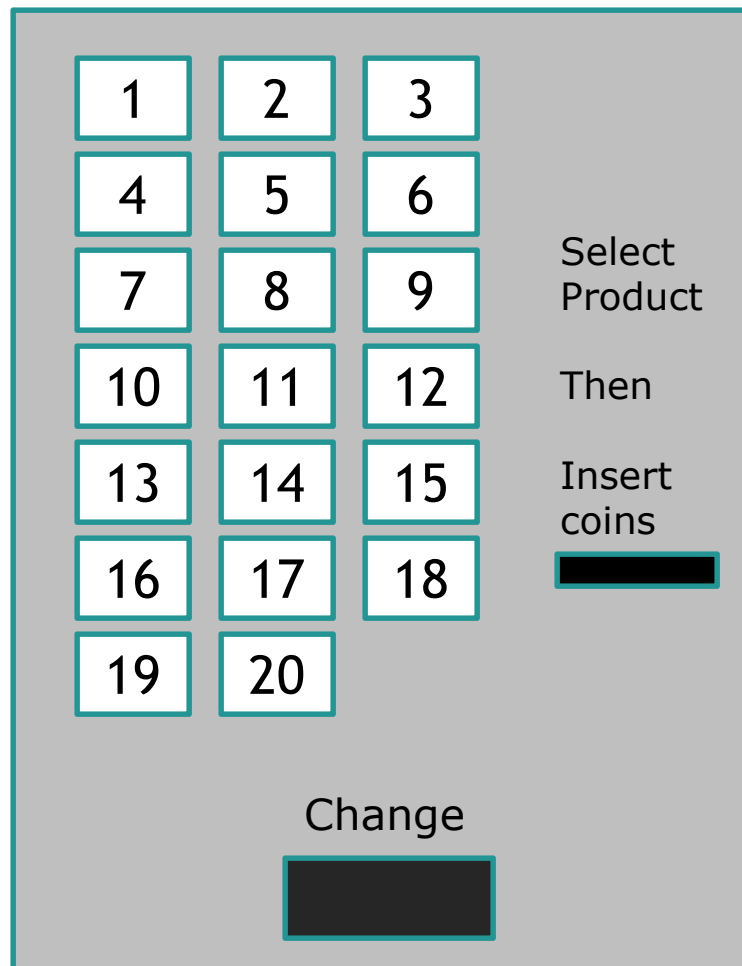
- ◆ Build a VDM++ model of a vending machine for products like drinks, snacks, cookies, etc. This machine accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro.
- ◆ There is a stock of coins inside the vending machine which is used to give the due change to clients who may need it.
- ◆ The vending machine 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:**

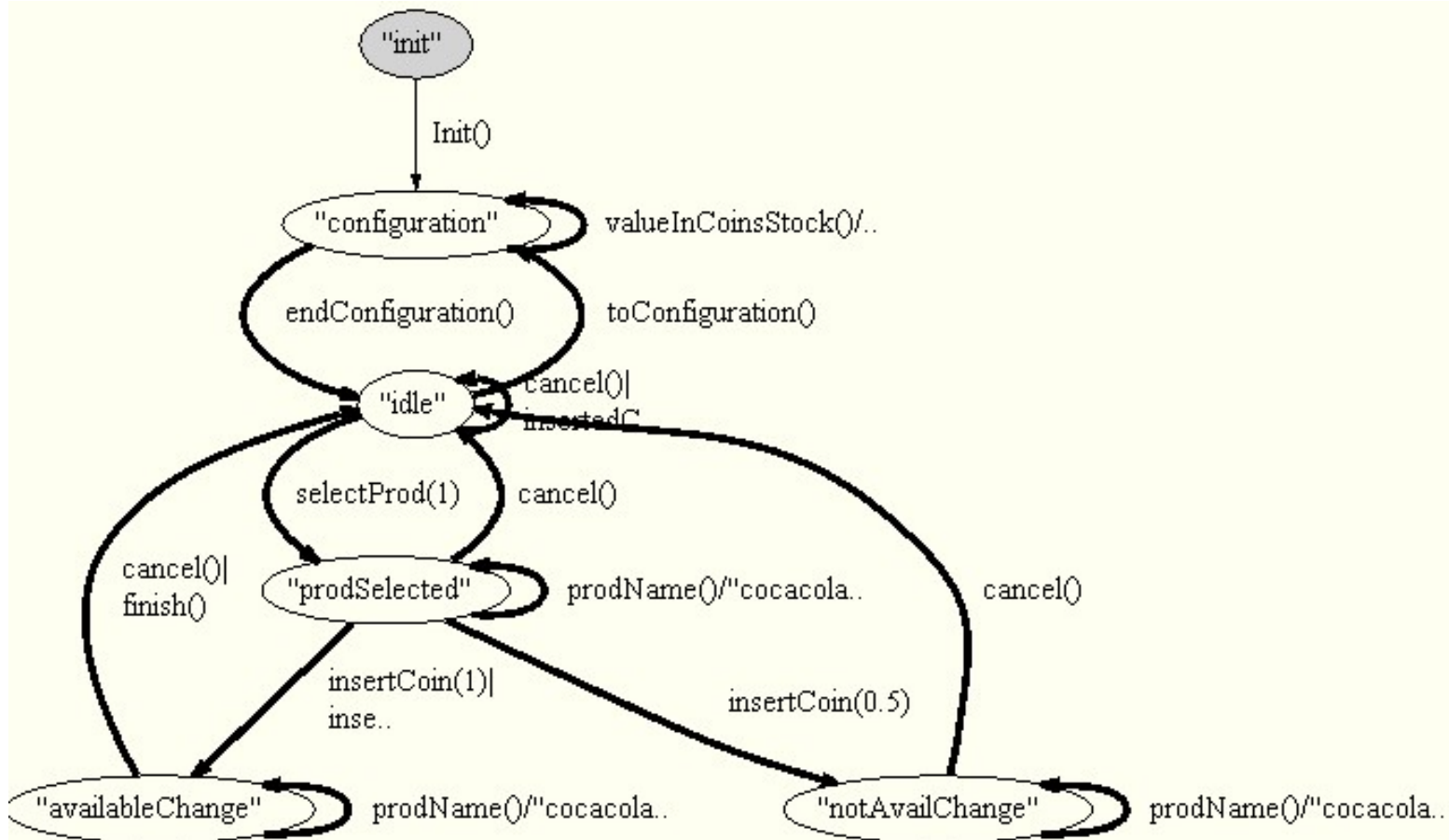
- The vending machine should show the products available to the clients.
- The clients should select the product they want and then insert euro coins to pay for it;
- The vending machine should give change every time it is possible (there are coins in stock for that purpose); otherwise it should give 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;
);
```

Cancel the current purchase

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

Simulates user to pick the selected product

```
public RecolheProduto() == (
 if (stockProd(prodSelected).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;
```