Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Agenda

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

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)

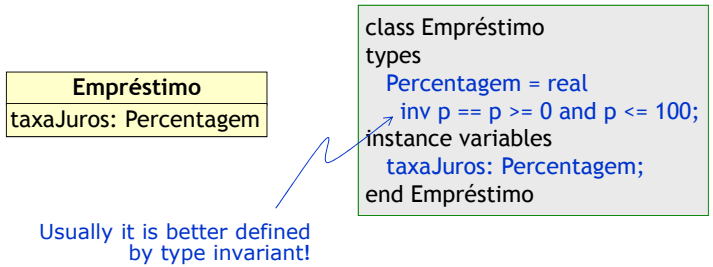
Common types of invariants

- 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

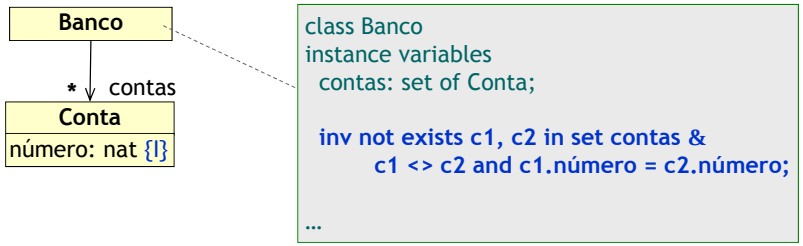
The interest rate on a loan is a percentage between 0 and 100%.



Common types of invariants

- Unique key constraint

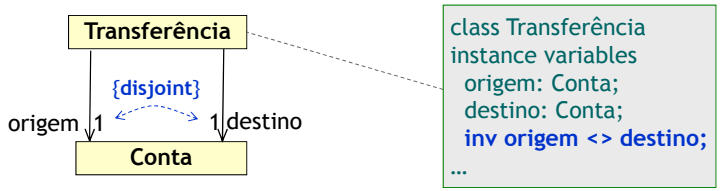
A bank can not have two accounts with the same number



Common types of invariants

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

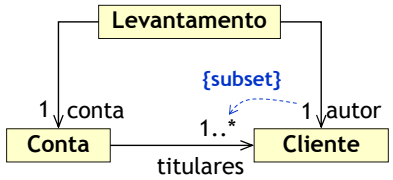
A transfer must be made between different accounts



Common types of invariantes

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

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

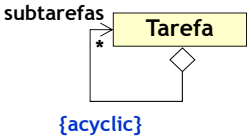



```
class Levantamento
instance variables
  conta: Conta;
  autor: Cliente;
  inv autor in set conta.titulares;
...
```


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!

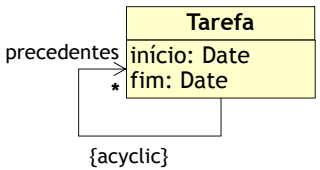
How to generalize to reuse?

```
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

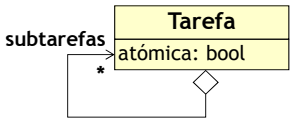



```
class Tarefa
types
  Date = nat; -- YYYYMMDD
instance variables
  início: Date;
  fim: Date;
  precedentes: set of Tarefa;

  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

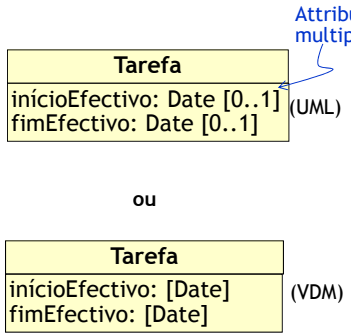



```
class Tarefa
instance variables
  atómica: bool;
  subtarefas: set of Tarefa;

  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

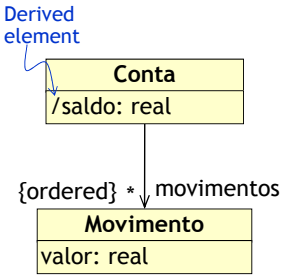



```
class Tarefa
instance variables
  inícioEfectivo: [Date];
  fimEfectivo: [Date];

  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)

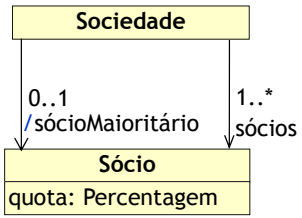



```
class Conta
instance variables
  saldo: real;
  movimentos: seq of Movimento;

  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%

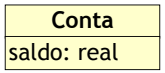



```
class Sociedade
instance variables
  sócios: set of Sócio;
  sócioMaioritário: [Sócio];

  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

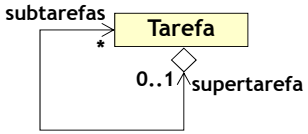



```
class Conta
instance variables
  saldo: real;

  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?)

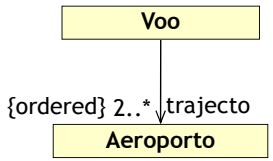
It can be seen as a case of cycle associations

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!

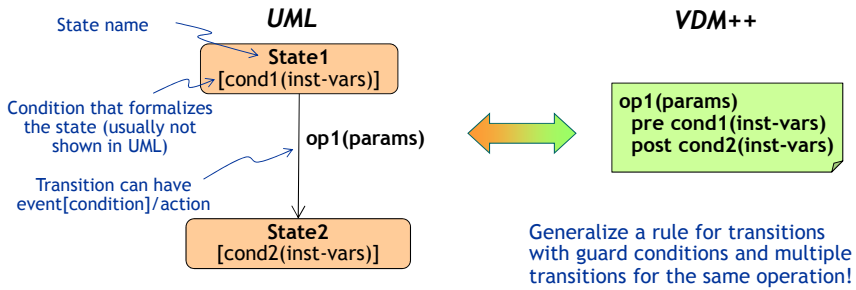
Other languages (OCL, Spec #, etc..) solve the problem of verifying an invariant only within the limits of method calls!

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