Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Agenda

- VDMTools
- Características da linguagem VDM++
  - Classes; Variáveis de instância; Operações; Funções (polimórficas, de ordem superior, lambda, …); Tipos; Operadores; Expressões
  - Design-by-contact:
    - Definição de invariantes; pré e pós-condições
    - Ligação do VDM++ ao UML
- Exemplo da Vending Machine
- **Consistência da especificação: obrigações de prova e teste**
- Concorrência em 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
```

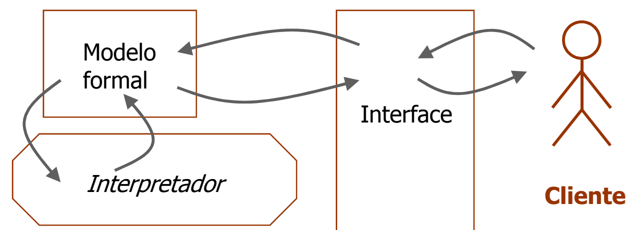
If it is not possible to find a result of type `nat` which satisfies `post_f` then `f` is not satisfiable

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

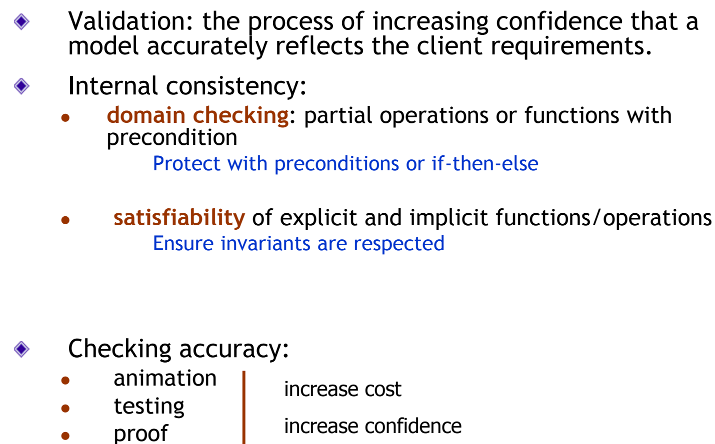
- Systematic testing and animation are only as good as the tests and scenarios used. Proof allows the modeller to assess the behaviour of a model for whole classes of inputs in one analysis.
- In order to prove a property of a model, the property has to be formulated as a logical expression (like a proof 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



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, preconditions and postconditions
- 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