Slide 1

An introduction to formal methods

Guy Tremblay

Dépt. d'informatique, UQAM

Montréal, Qué.

Michigan Technological University

February 9, 2000

Slide 2

Outline:

- What is a specification?
- What is a formal method?
- What is a formal specification language?
- Why are there many specification languages and methods?
- When can formal methods be used?
- What are the major benefits of formal specifications?
- Are there examples of *real* applications developed with FMs?

Slide 3

1 Prelude: Analysis, specification, design

Many possible levels of description:

(a) The needs and requirements of users;

(b) The set (the space) of possible solutions;

(c) One specific solution;

(d) The high-level, internal structure of the selected solution;

(e) The detailed description of the modules making up the solution;

(f) The algorithms used by each module;

(g) The final code.

Slide 4

Analysis and specification

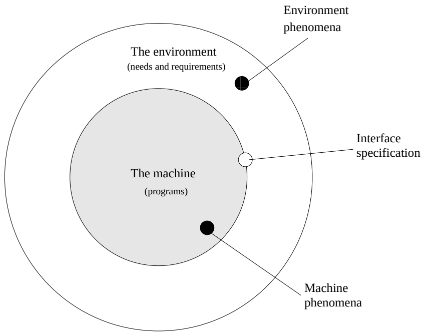
The **analysis and specification** stage consists in:

1. Analysis of the problem = Analyze and understand the problem to be solved, the needs and requirements of users, the constraints to satisfy = (a)+(b)
   - (a) Needs and requirements
   - (b) Space of possible solutions
2. Specification of a solution = Describe the software system that will fulfill the needs and requirements and that will satisfy the constraints = (c)
   - (c) Description of a specific solution

Slide 5

Requirements and needs vs. specifications (according to M. Jackson):

- Requirements and needs = phenomena *in the environment*
- Programs = phenomena *in the machine*
- Specifications = phenomena at the interface between the environment and the machine



Slide 6

Design

- Architectural design:
  - What are the major modules?
  - How are they organized and structured?
  - What are the interfaces between these modules?
- Detailed (procedural) design:
  - What is the exact behavior of each of the modules?

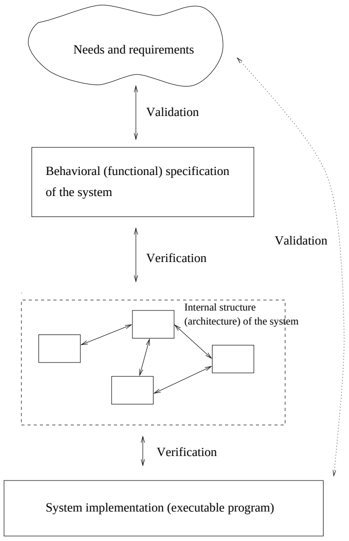
Analysis and specification vs. design

Thus:

- Analysis and specification: Concepts and interfaces for using the system (*user*'s point of view);
- Design: Concepts and interfaces for building the system (*builder*'s point of view).

Slide 7

Why is a specification important?



Specification = *contract* ... that must be fulfilled

- Validation: *Are we building the right product?*
- Verification: *Are we building the product right?*

Slide 8

2 What is a "formal method"?

> *"[Formal methods are] mathematically based techniques used to describe the properties of computing systems. They provide a framework in which one can specify, develop, and verify systems in a systematic and rigorous manner, rather then in* ad hoc *way."* [Wing90]

Key elements of a formal method:

- *Formal* language for writing specifications
- Rules to check the validity and quality of the specifications
- Strategies and rules to refine the specifications and verify these refinements

Foundation on which *everything* rests = Formal Specification!

Slide 9

3 What is a formal *specification* language?

Formal language => well-defined syntax and semantics:

- Syntax = EBNF; syntax diagrams; etc.
- Semantic = algebras; automatas and transition systems; functions, relations and predicates; etc.

Specification lang. => describe software components's external *behavior*

- by describing its major properties
- in an abstract way, without unneeded implementation details
- without saying how it is going to be implemented (non-algorithmic)

Thus, a programming language *is not* a specification language:

- algorithmic (operational)
- not abstract (arrays, pointers, etc.)

Slide 10

Really difficult part of writing specifications:

is not mathematics (sets, functions)

but is modeling (understand the problem and describe a solution)

The underlying mathematics for specifications (e.g., using an abstract modeling approach) are relatively simple:

- Propositional logic:

```
 p => q  <=>  ~p | q
~(p & q) <=>  ~p | ~q
```

- Predicate logic:

```
 SOME( i: nat :: 2*i = i )
~SOME( i: nat SUCH THAT i ~= 0 :: 2*i = i )
 ALL( i: nat :: i < i+1 )
 ALL( i: nat, j: nat SUCH i < j :: i+1 < j+1 )
```

Slide 11

Example (M. Jackson) showing how formalization using logic can help make *explicit* certain implicit properties = Suppose we have the following two signs near an escalator:

Shoes Must Be Worn	Dogs Must Be Carried

```
ALL( p: person
     SUCH THAT wants_to_use_escalator(p) ::
          SOME( s: shoes :: must_wear(p, s) ) )

ALL( p: person, d: dog
     SUCH THAT wants_to_use_escalator_with(p, d) ::
          must_carry(p, d) )
```

Slide 12

- Set = unordered collection of elements

```
e1: set{nat}
e1 = {20, 30}
e1 U {30..32} = {20, 30, 31, 32}
intersection(e1, {30..40}) = {30}
{x: nat SUCH THAT 1 <= x <= 10 :: x-1} = {0..9}
```

- Sequence = ordered, indexable collection of elements:

```
s1: sequence{nat}
s1 = [20, 30]
length(s1) = 2
length(s1 || s1) = 4
add(10, s1) = [10, 20, 30]
s1 || [20..22] = [20, 30, 20, 21, 22]
```

Slide 13

- Map = function which binds a unique definition to each key:

```
d1: map{string, nat}
d1 = {["John", 23], ["Mary", 35]; 0}
d1["Mary"] = 35
d1["Paul"] = 0
"John" IN d1
domain(d1) = {"John", "Mary"}
range(d1) = {0, 35, 66}
```

Slide 14

Formal specifications are non-algorithmic descriptions (what? instead of how?)

Two examples:

- A function to compute the square root of a real number
- A function to sort a sequence of values

Slide 15

```
FUNCTION square_root
         {precision: real SUCH THAT precision > 0.0}

  MESSAGE root( x: real )
    WHEN x >= 0.0
      REPLY( y: real )
        WHERE y >= 0.0 & almost_equal( y^2.0, x )
    OTHERWISE
      REPLY EXCEPTION imaginary_root

  CONCEPT almost_equal( r1 r2: real )
    VALUE( b: boolean )
      WHERE b <=> abs(r1 - r2) <= precision
END
```

Slide 16

```
FUNCTION sort{t: type SUCH THAT Subtype(t, total_order{t})}
 MESSAGE sort( s1: sequence{t} )
   REPLY( s2: sequence{t} )
     WHERE is_permutation(s1, s2),
           in_order(s2)


 CONCEPT is_permutation( s1 s2: sequence{t} )
         VALUE( b: boolean )
   WHERE b <=> ALL(x: t :: nb_occs(x, s1) = nb_occs(x, s2))

 CONCEPT nb_occs( x: t, s: sequence{t} ) VALUE( n: nat )
   WHERE n = NUMBER( i: nat SUCH THAT i IN domain(s)
                                    & s[i] = x :: i )

 CONCEPT in_order( s: sequence{t} ) VALUE( b: boolean )
   WHERE b <=> ALL( i: nat SUCH THAT 1 <= i < length(s) ::
                    s[i] <= s[i+1] )
END
```

Slide 17

4 Why are there many specification languages and methods?

Many different *styles* of specifications:

- Abstract modeling (VDM, Z, Spec, etc.)
- Algebraic specification (Larch, ACT ONE, etc.)
- Process algebras (CCS, CSP, LOTOS, etc.)
- Temporal logic
- ...

Similar to programming languages:

- Various application domains
- Various styles and paradigms
- Varying expressive power

Slide 18

Example: a small library system

```
MACHINE LibSys{ books: set{Book} }
  STATE( available: set{Book},
         borrowed:  map{Book, User} )
  INVARIANT
    available U domain(borrowed)              = books,
    intersection(available, domain(borrowed)) = {}
  INITIALLY
    available        = books,
    domain(borrowed) = {}

  MESSAGE CheckOut( u: User, b: Book )
    WHEN b IN available
      REPLY OK
      TRANSITION
        available = *available - {b},
        borrowed  = bind(b, u, *borrowed)
    OTHERWISE
      REPLY EXCEPTION Book_not_available
```

Slide 19

```
  MESSAGE Return( u: User, b: Book )
    WHEN borrowed[b] = u
      REPLY OK
      TRANSITION
        available = add(b, *available),
        borrowed  = remove(b, *borrowed)
    OTHERWISE
      REPLY EXCEPTION Invalid_return

  MESSAGE CheckedOutBooks( u: User )
    REPLY( books: set{Book}, count: nat )
     WHERE books = {b: Book SUCH THAT borrowed[b] = u :: b},
           count = size(books)

  CONCEPT Book: type
  CONCEPT User: type
END
```

Slide 20

Example: An algebraic specification (in Larch) of a stack

```
Stack(T, Stack) : trait
  introduces
    empty:             -> Stack
    isEmpty: Stack     -> Bool
    push:    Stack, T -> Stack
    pop:     Stack     -> Stack
    top:     Stack     -> T
  asserts
    Stack generated by empty, push
    Forall  x: T, p: Stack
      isEmpty(empty)       == true;
      isEmpty(push(p, x)) == false;
      top(push(p, x))      == x;
      pop(push(p, x))      == p;
```

Slide 21

Example: A process algebraic specification (in LOTOS) of a reactive program that computes the maximum of two numbers:

```
PROCESS Max2[inx, iny, outmax]: NOEXIT :=
  inx ?x: nat;  iny ?y: nat;
              ComputeMax[inx, iny, outmax](x, y)
  []
  iny y?: nat;  inx ?x: nat;
              ComputeMax[inx, iny, outmax](x, y)
WHERE
  PROCESS ComputeMax[inx, iny, outmax]
                  (x: nat, y: nat): NOEXIT :=
    [x <= y] -> outmax !y; Max2[inx, iny, outmax]
    []
    [x >= y] -> outmax !x; Max2[inx, iny, outmax]
  ENDPROC
ENDPROC
```

Slide 22

Applying a formal *method* (for ex., VDM) also require that we verify the *intrinsic* quality of our specifications. For example:

- Check that the initial state *establishes* the invariant
- Check that each operation *preserves* the invariant
- Check that the indicated responses can indeed be computed

This last step requires knowledge about the application domain:

```
MESSAGE square_root( n: nat )
  REPLY( r: nat ) WHERE r^2 = n
```

One other important . . . but *informal* step = validate the specification with the clients/user

⇒ Translate it back in natural language and/or using various graphical notations

Subsequent steps of development = verify that the design and implementation satisfy the specification

Slide 23

5 When can formal methods be used?

- Analysis and specification:
  - Description of the application domain (conceptual modeling)
  - Specification of the behavior of the solution (system to be developed)
  - Prototyping (animation of specification)
- Architectural and detailed design:
  - Formal specification of the modules' interfaces
  - Rigorous refinements
- Coding:
  - Code synthesis
  - Program proofs
- Tests
  - Automatic generation of tests

Slide 24

6 What are the major benefits of formal specifications?

> *"Having to better understand the* specificand *by compeling the analyst to be abstract yet precise about the properties of the system can be more rewarding than having the specification itself." [Wing90]*

- Positive impact of the *formalization effort* = help find and identify errors, ambiguities, and problems early.
- Specifications are more explicit, precise, with less ambiguity.
- Provide a better basis for implementation and formal verifications.
- Tools (manipulation, analysis, simulation).
- Basis for developing tests.

Slide 25

7 Are there examples of *real* applications developed with formal methods?

- Re-engineering of the CICS system (IBM).
- Hardware components (Inmos).
- Electronic devices (oscilloscopes) (Tektronik).
- Tools for structured analysis.
- Subway control system (Paris, France).
- Train control system (SNCF, France).
- Airline traffic control system (UK, USA).
- Nuclear reactor control systems (Ontario, USA).
- NASA's space shuttle (flight control, GPS).
- Radiotherapy control.

Slide 26

8 Conclusion

- Formal methods can be useful and can play an important role when developing critical systems
- They may appear to be costly but they can help reduce the costs of the subsequent phases (implementation and debugging, tests)
- . . . but they *are not* the *silver bullet*

- To know more about formal methods:
  - "The Object Constraint Language — Precise Modeling with UML", J. Warmer and A. Kleppe, Addison-Wesley, 1999.
  - Web sites:

    `http://www.afm.sbu.ac.uk/fm/`

    `http://www.fmeurope.org/`