



*Cycle Ingénieur – 2<sup>ème</sup> année*  
*Département Informatique*

# Verification and Validation

## Part IV : Proof-based Verification

Burkhart Wolff

Département Informatique  
Université Paris-Sud / Orsay

# Difference between Validation and Verification

---

- ❑ Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

*Do we build the right system ?*

- ❑ Verification: Does the system meet the specification ?

*Do we build the system right ?*

*Is it « correct » ?*

# What are the limits of test-based verification

---

- ❑ Assumptions on „Testability“

(system under test must behave deterministically, or have controlled non-determinism, must be initializable)

- ❑ Assumptions like Test-Hypothesis

(Uniform / Regular behaviour is sometimes a „realistic“ assumption, but not always)

- ❑ Limits in perfection:

We know only up to a given “certainty” that the program meets the specification ...

# How to do Verification ?

---

- In the sequel, we concentrate on Verification by Proof Techniques ...

# Standard example

---

The specification in UML/OCL (Classes in USE Notation):

```
class Triangles inherits_from Shapes
```

```
attributes
```

```
  a : Integer
```

```
  b : Integer
```

```
  c : Integer
```

```
operations
```

```
  mk(Integer,Integer,Integer) :Triangle
```

```
  is_Triangle() : triangle
```

```
end
```

# Standard example : Triangle

---

The specification in UML/OCL (Classes in USE Notation):

**context** Triangles:

**inv** def : a.oclIsDefined() and b.oclIsDefined()...

**inv** pos :  $0 < a$  and  $0 < b$  and  $0 < c$

**inv** triangle :  $a + b > c$  and  $b + c > a$  and  $c + a > b$

**context** Triangle::isTriangle()

**post** equi :  $a = b$  and  $b = c$  implies result=equilateral

**post** iso :  $((a <> b$  or  $b <> c)$  and  
 $(a = b$  or  $b = c$  or  $a = c))$  implies result=isosceles

**post** default:  $(a <> b$  or  $b <> c)$  and  
 $(a <> b$  and  $b <> c$  and  $a <> c)$   
implies result=arbitrary

# Standard example: Triangle

---

```
procedure triangle(j,k,l : positive) is
  eg: natural := 0;
begin
  if j + k <= l or k + l <= j or l + j <= k then
    put("impossible");
  else if j = k then          eg := eg + 1; end if;
    if j = l then          eg := eg + 1; end if;
    if l = k then          eg := eg + 1; end if;
    if eg = 0 then put("quelconque");
    elsif eg = 1 then put("isocele");
    else          put("equilateral");
    end if;
end if;
end triangle;
```

# Standard example : Exponentiation

---

The specification in UML/OCL (Classes in USE Notation):

```
context OclAny:
```

```
def exp(x,n) = if n >= 0 then  
    if n=0 then 1  
    else x*exp(x,n-1)  
    endif  
else OclUndefined endif
```

```
context Integer :: exponent(n:Integer):Real
```

```
pre true
```

```
post result = if n>= 0 then exp(self,n)  
    else 1 / exp(self,-n) endif
```

# Program Example : Exponentiation

---

Program\_1 :

```
S:=1; P:=N;
```

```
while P >= 1 loop S:= S*X; P:= P-1; end loop;
```

Program\_2 :

```
S:=1; P:= N;
```

```
while P >= 1 loop
```

```
    if P mod 2 <> 0 then P := P-1; S := S*X; end if;
```

```
    S:= S*S; P := P div 2;
```

```
end loop;
```

These programs have the following characteristics:

- one is more efficient, but more difficult to test
- good tests for one program are not necessarily good for the other

# How to do Verification ?

---

- How to PROVE that the programs meet the specification ?



# Introduction to proof-based program verification

# The role of formal proof

---

- ❑ formal proofs are another technique for program validation
  - based on a model of the underlying programming language, the conformance of a concrete program to its specification can be established

**FOR ALL INPUT DATA AND ALL INITIAL STATES !!!**

- ❑ formal proofs as verification technique can:
  - verify that a more concrete design-model “fits” to a more abstract design model (construction by formal refinement)
  - verify that a program “fits” to a concrete design model.

# Who is using formal proofs in industry?

---

## ❑ Hardware Suppliers:

- INTEL: Proof of Floating Point Computation compliance to IEEE754
- INTEL: Correctness of Cash-Memory-Coherence Protocols
- AMD: Correctness of Floating-Point-Units against Design-Spec
- GemPlus: Verification of Smart-Card-Applications in Security

## ❑ Software Suppliers:

- MicroSoft: Many Drivers running in „Kernel Mode“ were verified
- MicroSoft: Verification of the Hyper-V OS (60000 Lines of Concurrent, Low-Level C Code ...)
- . . .

# Who is using formal proofs in industry?

---

- ❑ For the highest certification levels along the lines of the Common Criteria, formal proofs are
  - recommended (EAL6)
  - mandatory (EAL7)

There had been now several industrial cases of EAL7 certifications ...

- ❑ For lower levels of certifications, still, formal specifications were required. Recently, Microsoft has agreed in a Monopoly-Lawsuit against the European Commission to provide a formal Spec of the Windows-Server-Protocols. (The tools validating them use internally automated proofs).

# Pre-Prerequisites of Formal Proof Techniques

---

- ❑ A Formal Specification (OCL, but also Z, VDM, CSP, B, ...)
  - know-how over the application domain
  - informal and formal requirements of the system
- ❑ Either a formal model of the programming language or a trusted code-generator from concrete design specs
- ❑ Tool Chains to generate, simplify, and solve large formulas (decision procedures)
- ❑ Proof Tools and Proof Checker: proofs can also be false ...

*Nous, on le fera à la main ;-(*

# Proof Systems

---

- An Inference System (or *Logical Calculus*) allows to infer formulas from a set of *elementary facts* (axioms) and inferred facts by rules:

$$\frac{A_1 \quad \dots \quad A_n}{A_{n+1}}$$

“from the *assumptions*  $A_1$  to  $A_n$ , you can infer the conclusion  $A_{n+1}$ .” A rule with  $n=0$  is an elementary fact. Variables occurring in the formulas  $A_n$  can be arbitrarily substituted.

# Proof Systems

---

- An Inference System for the equality operator (or “Equational Logic”) looks like this:

$$\frac{}{x = x} \qquad \frac{x = y}{y = x} \qquad \frac{x = y \quad y = z}{x = z}$$

$$\frac{x = y \quad P(x)}{P(y)}$$

(where the first rule is an elementary fact).

# Proof Systems

---

- A series of inference rule applications is usually displayed as *Proof Tree* (or : *Derivation*)

$$\frac{\frac{f(a,b) = a \quad \frac{f(a,b) = a \quad f(f(a,b),b) = c}{f(a,b) = c}}{a = c} \quad \frac{}{g(a) = g(a)}}{g(a) = g(c)}$$

- The non-elementary facts are the *global assumptions* (here  $f(a,b) = a$  and  $f(f(a,b),b) = c$ ).

# Proof Systems

---

- As a short-cut, we also write for a derivation:

$$\{f(a, b) = a, f(f(a, b), b) = c\} \vdash g(a) = g(c)$$

... or generally speaking: from global assumptions  $A$  to a **theorem** (in theory  $E$ )  $\phi$ :

$$A \vdash_E \phi$$

This is what theorems are: derivable facts from assumptions in a certain logical system ...

---

# A Proof System for Propositional Logic

- Propositional Logic (PL) in so-called natural deduction:

$$\begin{array}{c}
 \frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad \frac{A \vee B \quad \begin{array}{c} [A] \\ \vdots \\ Q \end{array} \quad \begin{array}{c} [B] \\ \vdots \\ Q \end{array}}{Q} \\
 \\
 \frac{A \quad B}{A \wedge B} \qquad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B} \qquad \frac{A \wedge B \quad \begin{array}{c} [A, B] \\ \vdots \\ Q \end{array}}{Q}
 \end{array}$$

# A Proof System for Propositional Logic

---

- Propositional Logic (PL) in so-called natural deduction:

$$\begin{array}{c}
 \frac{\textit{False}}{A} \\
 \\
 \frac{\begin{array}{c} [A] \\ \vdots \\ \neg A \end{array}}{B} \\
 \\
 \frac{\neg\neg A}{A} \\
 \\
 \frac{A}{\neg\neg A} \\
 \\
 \frac{P \rightarrow Q \quad P}{Q} \\
 \\
 \frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B}
 \end{array}$$

# A Proof System for Propositional Logic

---

- PL + E + Arithmetics (A) in so-called natural deduction:

$$\overline{1 + x \neq x}$$

$$\overline{(1 + x = 1 + y) \rightarrow x = y}$$

$$\frac{P(0) \quad \forall x. P(x) \rightarrow P(1 + x)}{\forall x. P(x)}$$

$$\overline{(1 + x) + y = 1 + (x + y)}$$

$$\overline{x + y = y + x}$$

$$\overline{x + (y + z) = (x + y) + z}$$

# Hoare – Logic: A Proof System for Programs

---

- PL + E + A + Hoare (simplified binding):

$$\frac{}{\vdash \{P\} \text{ SKIP } \{P\}} \quad \frac{}{\vdash \{P[x \mapsto E]\} \text{ x } ::= E \{P\}}$$

$$\frac{\vdash \{P \wedge \text{cond}\} c \{Q\} \quad \vdash \{P \wedge \neg \text{cond}\} d \{Q\}}{\vdash \{P\} \text{ IF } \text{cond} \text{ THEN } c \text{ ELSE } d \{Q\}}$$

$$\frac{}{\vdash \{P \wedge \text{cond}\} c \{P\}}$$

$$\frac{}{\vdash \{P\} \text{ WHILE } \text{cond} \text{ DO } c \{P \wedge \neg \text{cond}\}}$$

$$\frac{P \rightarrow P' \quad \vdash \{P'\} \text{ cmd } \{Q'\} \quad Q' \rightarrow Q}{\vdash \{P\} \text{ cmd } \{Q\}}$$

# Hoare – Logic: A Proof System for Programs

---

- PL + E + A + Hoare (simplified binding):

... in the essence, the Hoare Calculus is an entirely syntactic game that constructs a **labelling** of the program with assertions  $P$ ,  $Q$ , etc ...

# Hoare – Logic: A Proof System for Programs

---

- PL + E + A + Hoare (simplified binding):

... however, there is the correctness theorem that relates „this game“ to the semantics:

$$\vdash \{P\} \text{ cmd } \{Q\} \rightarrow \models \{P\} \text{ cmd } \{Q\}$$

where the „validity of a Hoare Triple“ is defined by:

$$\models \{P\} \text{ cmd } \{Q\} \equiv \forall \sigma, \sigma'. (\sigma, \sigma') \in C(\text{cmd}) \rightarrow P(\sigma) \rightarrow Q(\sigma')$$

# Proof of Programs

---

- Note: Validity is a « partial correctness notion »

proof under condition that the program terminates.  
For non-terminating programs, the calculus allows to prove anything

- The Proof-Method is therefore two-staged:
  - verify termination (find measures for loops and recursive calls that strictly decrease for each iteration)
  - prove partial correctness of the spec for the program via a Hoare-Calculus (or a wp-calculus)



***total correctness = partial correctness + termination ...***

# Validation : Test or Proof (1)

---

## Test

- Requires Testability of Programs (initializable, reproducible behaviour, sufficient control over non-determinism)
- Can be also Work-Intensive !!!
- Requires Test-Tools
- Requires a Formal Specification
- Makes Test-Hypothesis, which can be hard to justify !

# Validation : Test or Proof (2)

---

## Formal Proof

- Can be very hard – up to infeasible  
(no one will probably ever prove correctness of MS Word!)
- Proof Work typically exceeds Programming work by a factor 10!
- Tools and Tool-Chains necessary
- *Makes assumptions on language, method, tool-correctness, too !*

# Validation : Test or Proof (3)

---

- ❑ Can we be sure, that the logical systems are consistent

Well, yes, practically.

(See Hales Article in AMS: "Formal Proof", 2008.

<http://www.ams.org/ams/press/hales-nots-dec08.html>)

- Can we be sure, that a specification "means" what we intend ?

Well, that's a really hard problem.

But when can we ever be entirely sure that we know what we have in mind ?

At least, we can gain confidence by animation and test, thus, by **experimenting** with them ...

# Validation : Test or Proof (end)

---

## Test and Proof are Complementary ...

- ❑ ... and extreme ends of a continuum : from static analysis to formal proof of “deep system properties”
- ❑ In practice, a good “verification plan” will be necessary to get the best results with a (usually limited) budget !!!
  - detect parts which are easy to test
  - detect parts which are easy to prove
  - good start: maintained formal specification
    - ☞ this leaves room for changes in the conception
    - ☞ ... and for different implementation of sub-components