2009-2010

POLYTECH
PARIS-SUD

Cycle Ingénieur – 2ème année

Département Informatique

Verification and Validation

Part IV : Proof-based Verification

Burkhart Wolff

Département Informatique

Université Paris-Sud / Orsay

Difference between Validation and

- 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 specifiation ...

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 god for the other

How to do Verification ?

- How to PROVE that the programs meet the specification ?

POLYTECH
PARIS-SUD

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 againt 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-Rerquisites 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 ;-(

Foundations: 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 occuring in the formulas A_n can be arbitraryly substituted.

Foundations: 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).

Foundations: Proof Systems

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

$$\dfrac{f(a,b)=a \qquad \dfrac{f(a,b)=a \quad f(f(a,b),b)=c}{f(a,b)=c}}{a=c} \qquad \dfrac{}{g(a)=g(a)}$$
$$\overline{g(a)=g(c)}$$

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

Foundations: 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) ϕ:

$$A \vdash_E \phi$$

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:

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad \frac{A \vee B \quad \begin{matrix}[A]\\ \vdots \\ Q\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ Q\end{matrix}}{Q}$$

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

A Proof System for Propositional Logic

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

$$\frac{False}{A} \qquad \frac{\begin{array}{c}[A]\\ \vdots \\ \neg A\end{array}}{B} \qquad \frac{\neg\neg A}{A} \qquad \frac{A}{\neg\neg A}$$

$$\frac{P \to Q \quad P}{Q} \qquad \frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}$$

A Proof System for Propositional Logic

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

$$\frac{}{1 + x \neq x} \qquad \frac{}{(1 + x = 1 + y) \rightarrow x = y}$$

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

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

$$\frac{}{x + y = y + x} \qquad \frac{}{x + (y + z) = (x + y) + z}$$

Hoare – Logic: A Proof System for Programs

- Now, can we build a

Logic for Programs ???

Hoare – Logic: A Proof System for Programs

- Now, can we build a

  Logic for Programs ???

  Well, yes !

  There are actually lots of possibilities ...

- We consider the Hoare-Logic (Sir Anthony Hoare ...), technically an inference system PL + E + A + Hoare

Hoare – Logic: A Proof System for Programs

- Basis: IMP, (following Glenn Wynskell's Book)

  We have the following commands (*cmd*)

  - the empty command SKIP
  - the assignment $x :== E \quad (x \in V)$
  - the sequential compos. $c_1 ; c_2$
  - the conditional IF cond THEN c_1 ELSE c_2
  - the loop WHILE cond DO c

  where c, c_1, c_2, are cmd's, V variables,
  E an arithmetic expression, cond a boolean expr.

Hoare – Logic: A Proof System for Programs

- Core Concept: A Hoare Triple consisting ...

  - of a pre-condition *P*
  - a post-condition *Q*
  - and a piece of program *cmd*

  written:

$$\vdash \{P\}\ cmd\ \{Q\}$$

P and Q are formulas over the variables V, so they can be seen as set of possible states.

Hoare Logic vs. Symbolic Execution

- HL is also based notion of a *symbolic state.*

$$state_{sym} = V \rightarrow Set(D)$$

As usual, we denote sets by

$$\{ x \mid E \}$$

where E is a boolean expression.

Hoare Logic vs. Symbolic Execution

- However, instead of:

  $\vdash \{\sigma::state_{sym} \mid Pre(\sigma(X_1), \ldots, \sigma(X_n))\}$

  cmd

  $\{\sigma::state_{sym} \mid Post(\sigma(X_1), \ldots, \sigma(X_n))\}$

  where Pre and Post are sets of states.
  we just write:

  $\vdash$ {Pre} cmd {Post}

  where Pre and Post are expressions over program variables.

Hoare Logic vs. Symbolic Execution

- Intuitively:

  |– {Pre} cmd {Post}

  means:

  If a program *cmd* starts in a state admitted by *Pre* if it terminates, that the program must reach a state that satisfies *Post*.

Hoare – Logic: A Proof System for Programs

- PL + E + A + Hoare (simplified binding) at a glance:

$$\frac{}{\vdash \{P\}\ \mathrm{SKIP}\ \{P\}} \qquad \frac{}{\vdash \{P[x \mapsto E]\}\ \mathrm{x :== E}\{P\}}$$

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

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

$$\frac{P \to P' \qquad \vdash \{P'\}\ cmd\ \{Q'\} \qquad Q' \to Q}{\vdash \{P\}\ cmd\ \{Q\}}$$

Verification : Test or Proof

Test

- Requires Testability of Programs (initialitzable, 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 !

Summary

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 (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

Hoare – Logic: Outlook

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

    Well, no.
    But when can we ever be entirely sure that we know what we have in mind ?
    But at least, we can gain confidence validating specs, i.e. by animation and test, thus, by experimenting with them ...