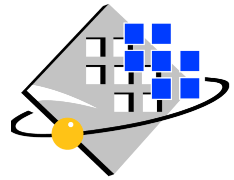
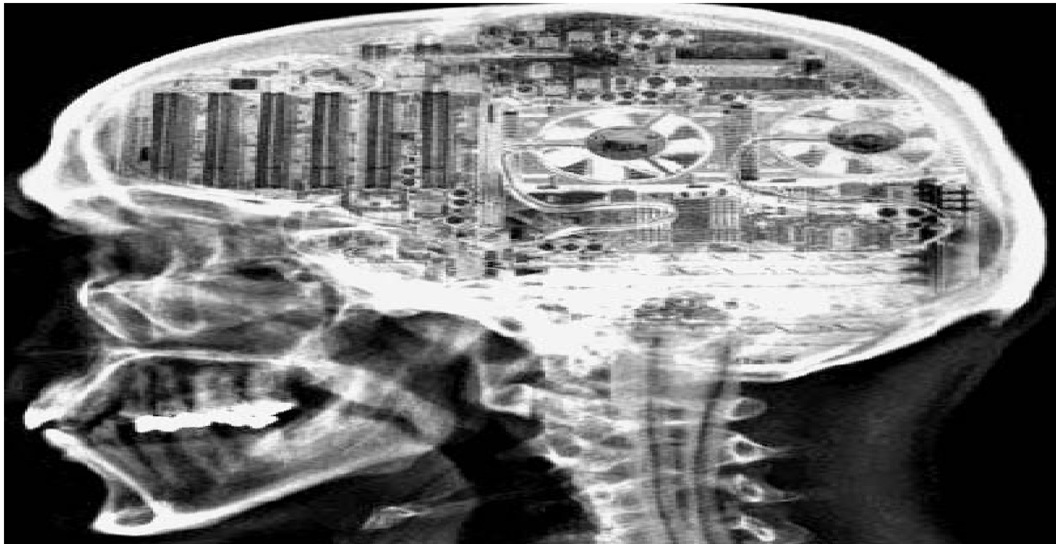


Licence Mention Informatique - L3/S6 - 2009/10

Introduction au Génie Logiciel

Partie3: Verification by Proof (I)



Burkhart Wolff
Département Informatique

Why Proof ?

- Test Procedures have theoretical limits in certifying Quality Assurance …
- Test Procedures have practical limits in certifying Quality Assurance …
  (low coverage of feasible paths,
  complexity of DNF-computations, ...
- Random-Testdata generators are usually
  very ineffective
- In some problem domains, usual Testing Hypothesises
  are not justifiable (stuck-at errors in Hardware)
- In some problem domains, usual Testability Hypothesis
  are not justifiable (SUT is *not* a function; non-determ. Sys.)

Difficulties with Testing so far

- So, for:
  - non-deterministic
    operating system level systems
  - large distributed code
  - high-assurance, safety-critical systems
    (Common Criteria EAL 7 level)

  ... Testing techniques can not be applied
  (at least : alone).

- Anyway, systematic testing involves specification as well, so, modeling work has to be done in both cases

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

- Software Suppliers:
  - GemPlus: Verification of Smart-Card-Applications in Security (First EAL7 level ceritification ...)
  - MicroSoft: Many Drivers running in „Kernel Mode" were verified
  - MicroSoft: Verification of the Hyper-V OS (60000 Lines of Concurrent, Low-Level C Code ...)
  - . . .

Roles of Proof

- Validation possible ?
  - … to a certain extent. Proofs — if done interactively — may help to deepen both the understanding of code & specification.
  - … however, this tends to be costly.
- Verification possible ?
  - Yes — Both for internal properties, for the conformance of design to analysis specs as well as the conformance of design to code.
  - for low-level, concurrent code without alternative …

Automation of Proof

- Can Program Verification be automated ?
  - … to a large extent

    (this is what the rest
    of the course is about)

  - … still, some very nasty parts of verification, in particular finding loop-invariants or delicate framing conditions, require a lot of thought and human user interaction.

Automation of Proof

- Can we do Automated Verification
  for UML/OCL ?

  - ... no. There are tools
    (like HOL-OCL, see)
    but they require too much knowledge on
    both proof and meta-theory of OCL for
    a course.

  - We will do most of it by hand ;-(

Requirements of Proof

- A Formal Specification (OCL or JML, but also Z, VDM, CSP, B, ..., ACDL or VCC)
  - know-how over the application domain
  - informal and formal requirements of the system (design-level, with framing conditions ...)
- a domain theory; a number of predicates describing foundational data-structures from the problem domain
- a semantics of the underlying programming language ...

Revision

STATE AND SYMBOLIC STATES ...

SEE SLIDES ON WHITE BOX TESTING ...

Foundation : What is a Formal Proof ?

- A Tree, where:
  - the nodes are formulas
  - the branches correspond to inference rules (so we need a set of these rules, the *inference system*)
  - rules should be logically valid

    (whatever this means ...)
  - the root of the tree is called the *conclusion*, the leaves the *assumptions*

Foundation : Proof Systems

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

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

"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.

Foundation : 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).

Foundation : Proof Systems

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

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

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

Foundation : 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 ...

Foundation : A Proof System for Propositional Logic

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

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

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

Foundation : A Proof System for Propositional Logic

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

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

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

Foundation : 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 (x ∈ 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: A Proof System for Programs

- The Inference System Hoare (simplified binding):

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

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

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

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

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:

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

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

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

Proof pf 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 mesures 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 (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 !

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 ...

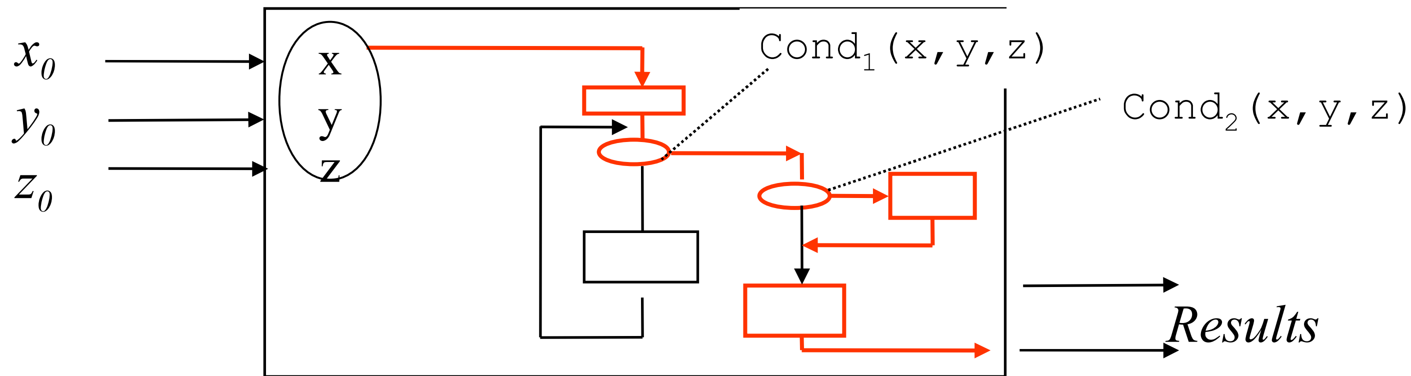
Validation : Test or Proof (end)

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

Static Structural ("white-box") Tests

- we select "critical" paths
- specification used to verify the obtained resultats



what the program does and how …

A path corresponds to one logical expression over x_0, y_0, z_0 .
corresponding to one test-case (comprising several test data ...)

$$\neg\, Cond_1(x_0,\ y_0,\ z_0) \wedge \neg\, Cond_2(x_0,\ y_0,\ z_0)$$

We are interested either in edges (control flow), or in nodes (data flow)

A Program for the triangle example

```
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(“arbitrary”);
     elsif    eg = 1  then put(“isocele”);
     else     put(“equilateral”);
     end if;
end if;
end triangle;
```

What are tests adapted to this program ?

- try a certain number of execution “paths” (which ones ? all of them ?)
- find input values to stimulate these paths
- compare the results with expected values (i.e. the specification)

Functional-test vs. structural test?

Both are complementary and complete each other:

- Structural Tests have weaknesses in principle:
  - if you forget a condition, the specification will most likely reveal this !
- Structural Tests have weaknesses in principle:
  for a given specification, there are several possible implementations (working more or less differently from the spec):
  - *sorted arrays : linear search ? binary search ?*
  - *$(x, n) \rightarrow x^n$: successive multiplication ? quadratic multiplication ?*

  Each implementation demands for different test sets !

Equivalent programs ...

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;
```

Both programs satisfy the same spec but ...

- one is more efficient, but more difficult to test.
- test sets for one are not necessarily “good” for the other, too !

Control Flow Graphs

A graph with oriented edges root E and an exit S,

- the nodes be either "elementary instruction blocs" or "decision nodes" labelled by a predicate.
- the arcs indicate the control flow between the elementary instruction blocs and decision nodes (control flow)
- all blocs of predicates are accessible from E and lead to S (otherwise, dead code is to be supressed !)

elementary instruction blocs: a sequence of

- assignments
- update operations (on arrays, ..., not discussed here)
- procedure calls (not discussed here !!!)

- conditions and expressions are assumed to be side-effect free

Computing Control Flow Graphs

- Identify longest sequences of assignments

Computing Control Flow Graphs

- Identify longest sequences of assignments

  Example:

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

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

Computing Control Flow Graphs

- Identify longest sequences of assignments

Example:

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

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

Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if_then_elses by branching

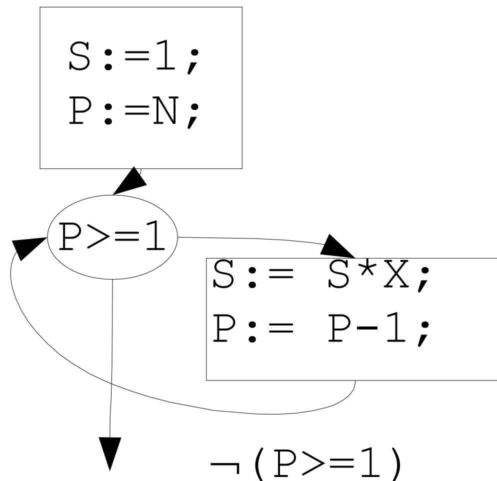
Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if_then_elses by branching
- Erase while_loops by loop-arc, entry-arc, exit-arc

Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if_then_elses by branching
- Erase while_loops by loop-arc, entry-arc, exit-arc

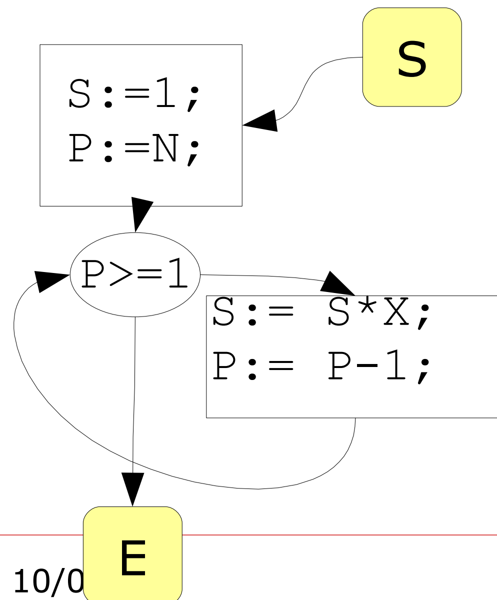
Example:



Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if_then_elses by branching
- Erase while_loops by loop-arc, entry-arc, exit-arc

Example:



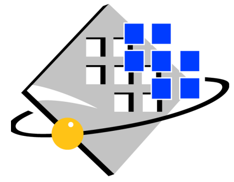
Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if_then_elses by branching
- Erase while_loops by loops
- Add entry node and exit loop-arc, entry-arc, exit-arc

A Control-Flow-Graph (CFG) is usually a by-product of a compiler ...

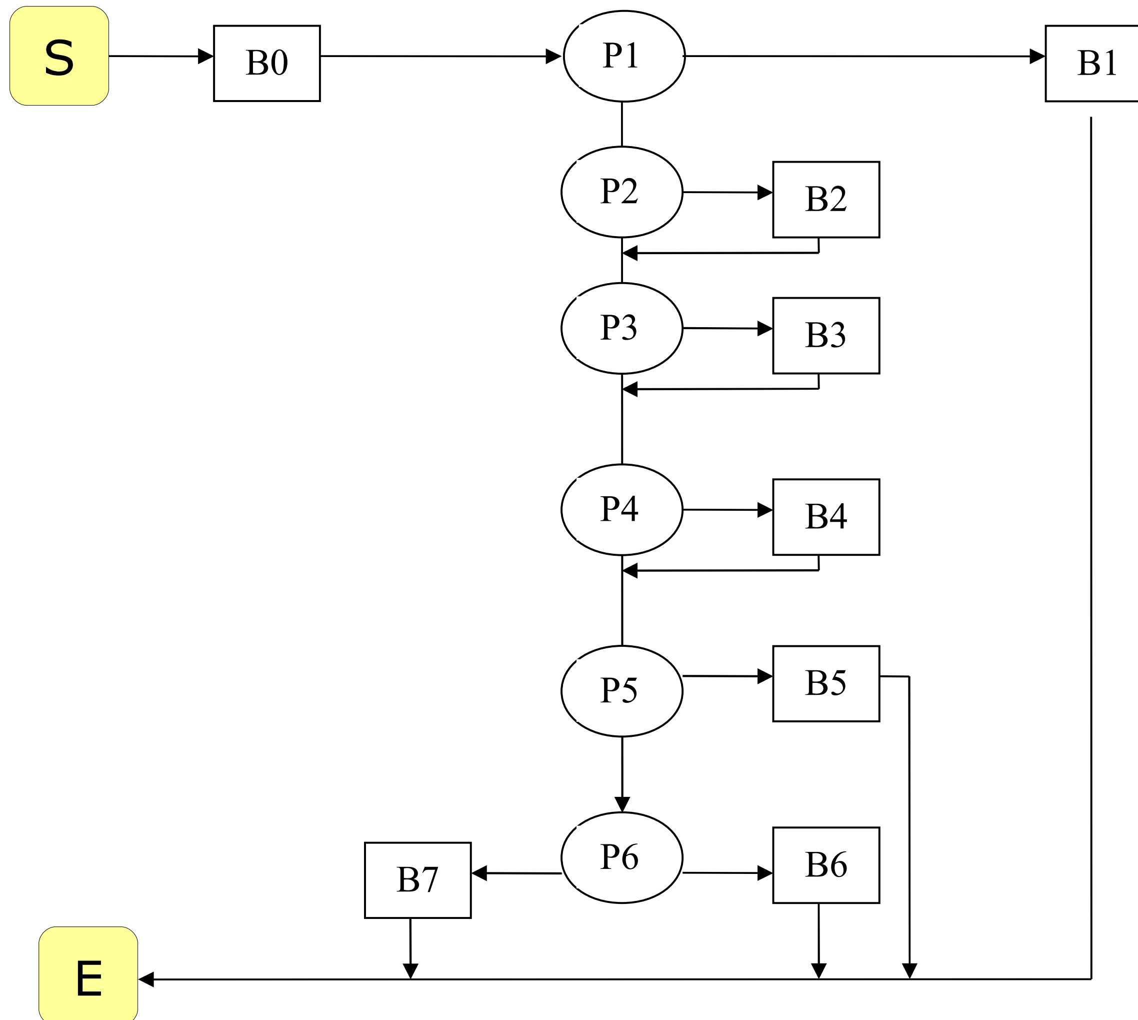
Revisiting our triangle example ...

```
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;
```



Q: What is the CFG of the body of triangle ?

Le graphe de flot de contrôle du programme



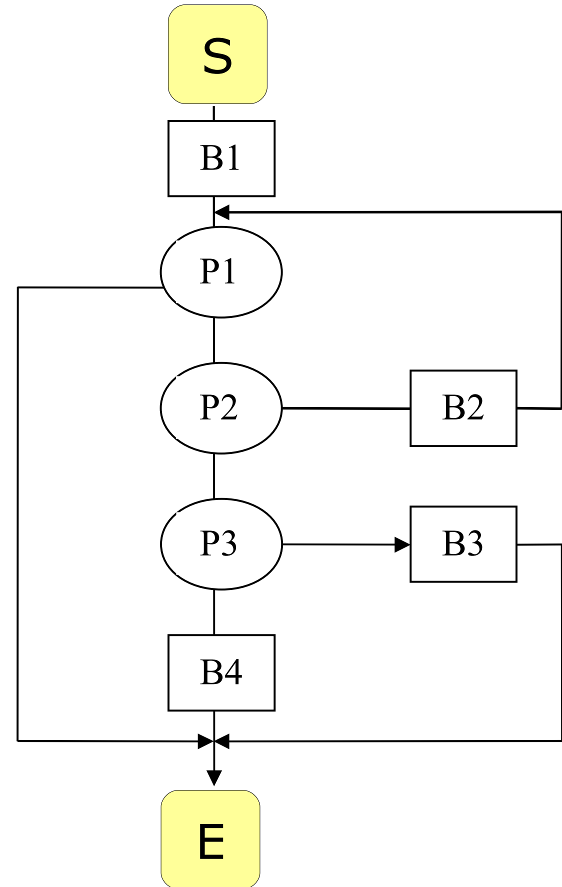
A procedure with loop and return

```
procedure supprime (T: in out Table; p: in out integer;
                    x: in integer) is
 i: integer := 1;
begin
   while     i <> p  loop
      if      T[i].val <> x then  i := i + 1;
      elsif   i = p - 1      then p := p - 1; return;
       else   T[i] := T[p-1]; p := p -1;  return;
      end if;
   end loop;
end supprime;
```

… and its control flow graph

What are the feasible paths ?

How to describe this ?



Paths and Path Conditions

- Let M a procedure to test, and G its control-flow graph. Terminology:
  - sub-path of M = path of G
  - initial path of M = path of G starting at S
  - path of M = path of G starting at S and leading to E

    *i.e. a **complete** execution of the procedure*

  - a given path is associated to predicate (over parameters and state): a condition over the **initial values initiales** of parameters (and global variables) to achieve **exactly** this execution path
  - faisable paths = a path of M pour a set for all parameters and global variables *exists* such that the path is executable.

    i.e. the path condition is satisfiable

Computing Path Conditions by Symbolic Execution

Let P be an initial path in M.

- we give symbolic values for each variable x_0, y_0, z_0, ...
- we set the path condition Φ initially "true"
- We follow the path, block for block, along P:

  If the block is an instruction block B:

  we execute symbolically B by memorizing the new values by expressions (symbolically) dependent on x_0, y_0, z_0, ...

  If the block is a decision block P(x,...,z)

  if we follow the « true » arc we set $\Phi := \Phi \wedge P(\underline{x},...,\underline{z})$,

  if we follow the «false» arc we set $\Phi := \Phi \wedge \neg P(\underline{x},...,\underline{z})$.

  (The $\underline{x},...,\underline{z}$ are the symbolic values for x,...,z.
  This effect is produced by a substitution to be discussed later.)

Execution

- Execution (in imperative languages) is based on the notion of *state.*

  A state is a table (or: function) that maps a variable *V* to some value of a domain *D*.

$$\text{state} = V \rightarrow D$$

  As usual, we denote (finite) functions as follows:

$$\{\ x \mapsto 1,\ y \mapsto 5,\ x \mapsto 12\ \}$$

Symbolic Execution

- In static program analysis, it is in general not possible to infer concrete values of D.

  However, it can be inferred a set of possible values.

  For example, if we know that

  $x \in \{1..10\}$

  and we have an assignment x:= x+2, we know:

  $x \in \{3..12\}$ afterwards.

Symbolic Execution

- This gives rise to the 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.
In our concrete technique, sets will always have the form $\{ x_0 \mid x_0 = E \}$ where E is an arithmetic expression (possibly containing variables of V).

Symbolic States and Substitutions

- Since in our concrete technique, sets have have the form $\{ x_0 \mid x_0 = E \}$, we can abbreviate:

$$\{x \mapsto \{x_0|x_0=E_1\}, y \mapsto \{y_0|y_0=E_2\}, z \mapsto \{z_0|z_0=E_3\}\}$$

to

$$\{x \mapsto E_1, y \mapsto E_2, z \mapsto E_3\}$$

and treat them as substitutions - all variables in an expression were subsequently replaced by their substituands ...

Symbolic States and Substitutions

- Example substitution:

$$(x + 2 * y) \{x \mapsto 1, y \mapsto x_0\}$$

$$= 1 + 2 * x_0$$

- An *initial symbolic state* is a state of the form:

$$\{ x \mapsto x_0, y \mapsto y_0, z \mapsto z_0 \}$$

Basic Blocks *as* Substitutions

Symbolic Pre-State

x	$\mapsto$	x_0
y	$\mapsto$	$y_0 + 3*x_0$
z	$\mapsto$	z_0
i	$\mapsto$	i_0

Block

```
i := x+y+1
z := z+i
```

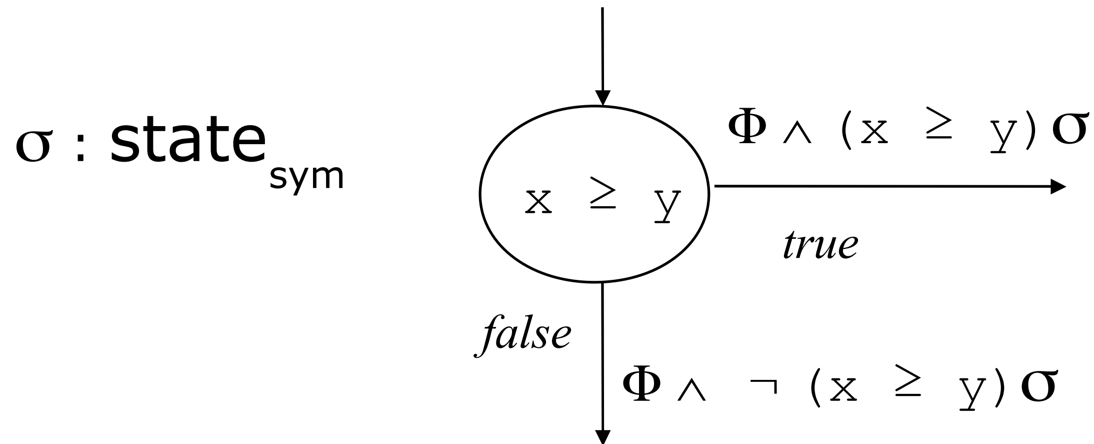
Symbolic Post-State

x	$\mapsto$	x_0
y	$\mapsto$	$y_0 + 3*x_0$
z	$\mapsto$	$z_0 + y_0 + 4*x_0 + 1$
i	$\mapsto$	$y_0 + 4*x_0 + 1$

x_0, y_0 and z_0 represent the initial values of x, y et z.

i is supposed to be a local variable (not initialized at the beginning).

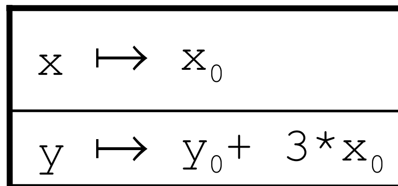
Symbolic Execution

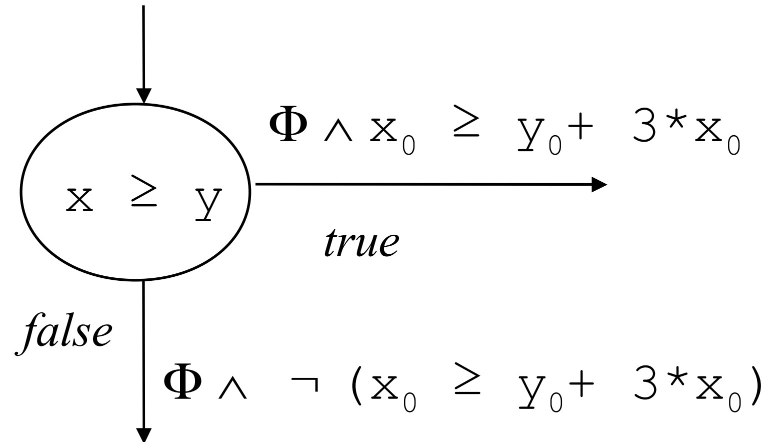


Thus, we execute symbolically and transform the symbolic state in order to obtain an expression depending on the **initial values of the parameters, (accesses to undefined local variables are treated by exception)**

Thus, we can construct for a given path the path-condition. For reasoning GLOBALLY over a loop, we would have to invent an « invariant » (corresponding to an induction scheme).

Symbolic Execution





Thus, we execute symbolically and transform the symbolic state in order to obtain an expression depending on the **initial values of the parameters, (accesses to undefined local variables are treated by exception)**

Thus, we can construct for a given path the path-condition. For reasoning GLOBALLY over a loop, we would have to invent an « invariant » (corresponding to an induction scheme).

Paths and Test Sets

In (this version of) program-based testing
a test case with a (feasable) path

- a test case ≈ an initial path in M

  = a collection of values for variables (params and global)

  (+ the output values described by the spécification)

- a test case set ≈ a finite set of paths of M

  = (by assuming a uniformity hypothesis)

  a finite set of input values and
  a set of expected outputs.

Unfeasible paths and decidability

- In general, it is undecidable of a path is feasible ...
- In general, it is undecidable if a program will terminate ...
- In general, equivalence on two programs is undecidable …
- In general, a first-order formula over arithmetic is undecidable ...
- **...**

  Indecidable = it is known (mathematically proven) that there is no algorithm; the is worse than "we know none" !

BUT: for many relevant programs, practically good solutions exist (Z3 -> Pex, Simplify, JavaPathfinder-SE) ...

Paths and Test Sets

In (this version of) program-based testing
a test case with a (feasable) path

- a test case ≈ an initial path in M

  = a collection of values for variables (params and global)

  (+ the output values described by the spécification)

- a test case set ≈ a finite set of paths of M

  = (by assuming a uniformity hypothesis)

  a finite set of input values and
  a set of expected outputs.

Unfeasible paths and decidability

- In general, it is undecidable of a path is feasible ...
- In general, it is undecidable if a program will terminate ...
- In general, equivalence on two programs is undecidable …
- In general, a first-order formula over arithmetic is undecidable ...
- **...**

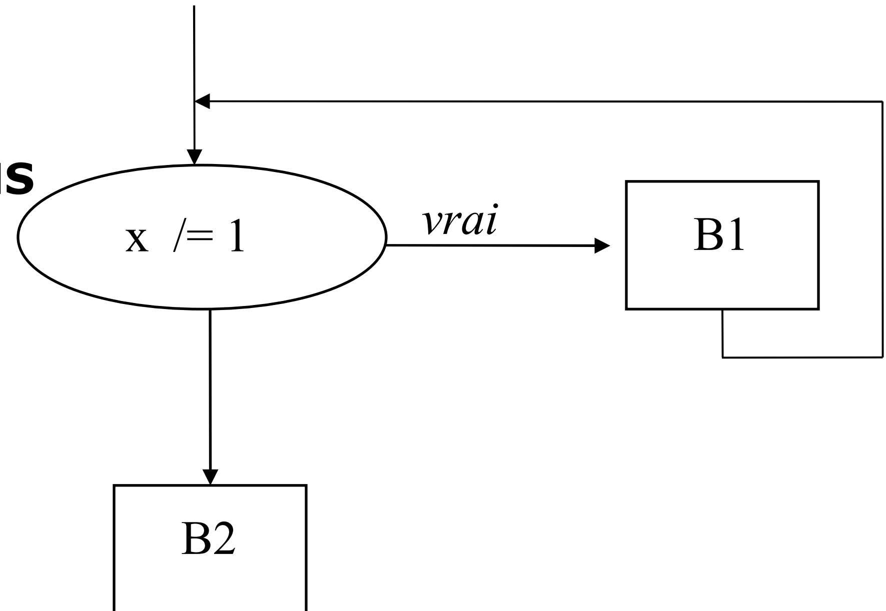
  Indecidable = it is known (mathematically proven) that there is no algorithm; the is worse than "we know none" !

BUT: for many relevant programs, practically good solutions exist (Z3 -> Pex, Simplify, JavaPathfinder-SE) ...

A Challange-Example (Collatz-Function):

... ALTHOUGH FOR SOME SPEC-TACULARLY SIMPLE PROGRAMS THESE SYSTEMS FAIL:

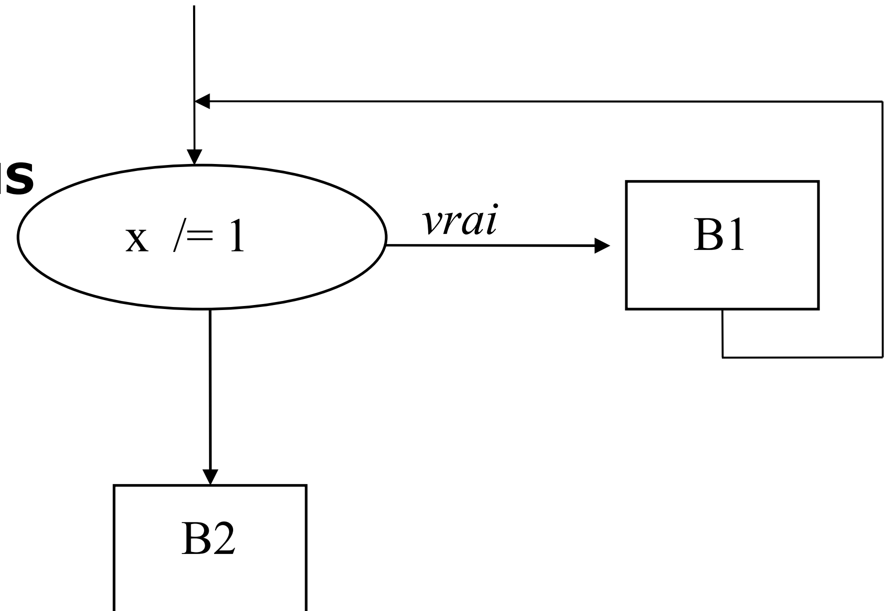
```
while x <> 1 loop
      if pair(x) then x := x / 2;
      else x := 3 * x + 1;
      end if;
 end loop;
```



- does this function terminate for all x ?
- or equivalently: is B2 reached for all x ?

A Challange-Example (Collatz-Function):

… ALTHOUGH FOR SOME SPEC- TACULARLY SIMPLE PROGRAMS THESE SYSTEMS FAIL:

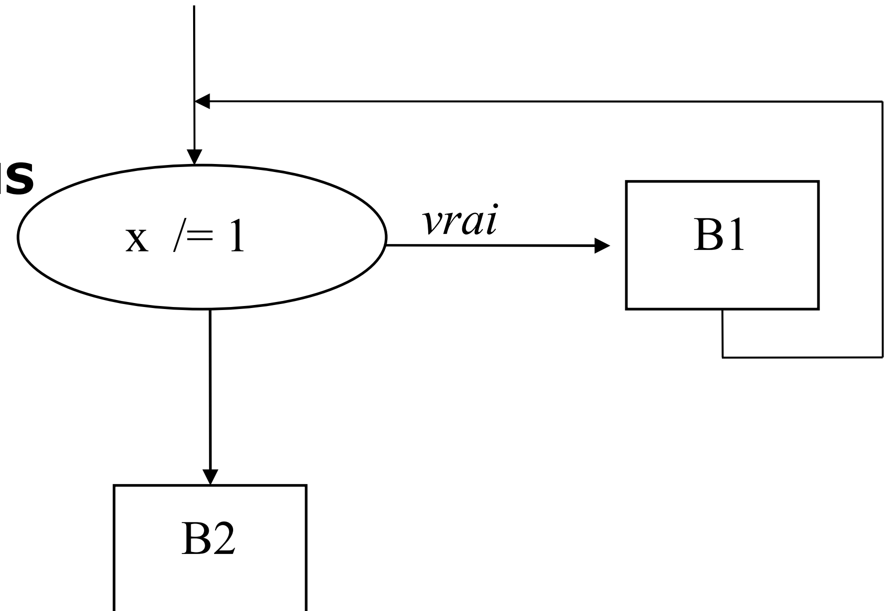



```
while x <> 1 loop
      if pair(x) then x := x / 2;
      else x := 3 * x + 1;
      end if;
 end loop;
```

- does this function terminate for all x ?
- or equivalently: is B2 reached for all x ? ANSWER:unknown

A Challange-Example (Collatz-Function):

… ALTHOUGH FOR SOME SPECTACULARLY SIMPLE PROGRAMS THESE SYSTEMS FAIL:




```
while x <> 1 loop
      if pair(x) then x := x / 2;
      else x := 3 * x + 1;
      end if;
 end loop;
```

- does this function terminate for all x ?
- or equivalently: is B2 reached for all x ? ANSWER:unknown
- this implies that we can not know in advance that there exist infeasible paths !

The Triangle Prog without Unfeasible Paths

```
procedure triangle(j,k,l)
begin
  if j k<=l or k+l<=j or l+j<=k  then  put("impossible");
  elsif j = k and k = l  then put("equilateral");
  elsif j = k or k =l or j = l then put("isocele")
  else   put("quelconque");
end if;
end;
```

☞ If we find a path for which we do not know that it is feasible (maybe for deep mathematical reasons, maybe simply because our prover is too week), however, it is likely in practice that there is an error ...

The notion of a “couverage criteria”

A coverage criterion is a predicate on CFG characterizing a particular subset of its paths ...

M = a procedure (with associated CFG G)

T = a test case set = a finite set of **feasable** paths in M

C = a coverage criterion (= a “set of paths”)

C(M, T) is true iff T satisfies the criterion C

Examples

- all nodes appear at least once in T
- all arcs appear at least once in T
- …

Well-known Coverage Criteria I

Criterion AllInstructions(M,T):

For all nodes N (basic instructions or decisions)
in the CFG of M exists a path in T that contains N

Well-known Coverage Criteria II

Criterion AllTransitions(M,T):

For all arcs A in the CFG of M exists a path in T that uses A

Well-known Coverage Criteria III

Criterion AllPaths(M,T):

All possible paths ...

☹ Whenever there is a loop, T is usually infinite !

Variant: $AllPaths_k(M,T)$.

We limit the paths through a loop to maximally k times …

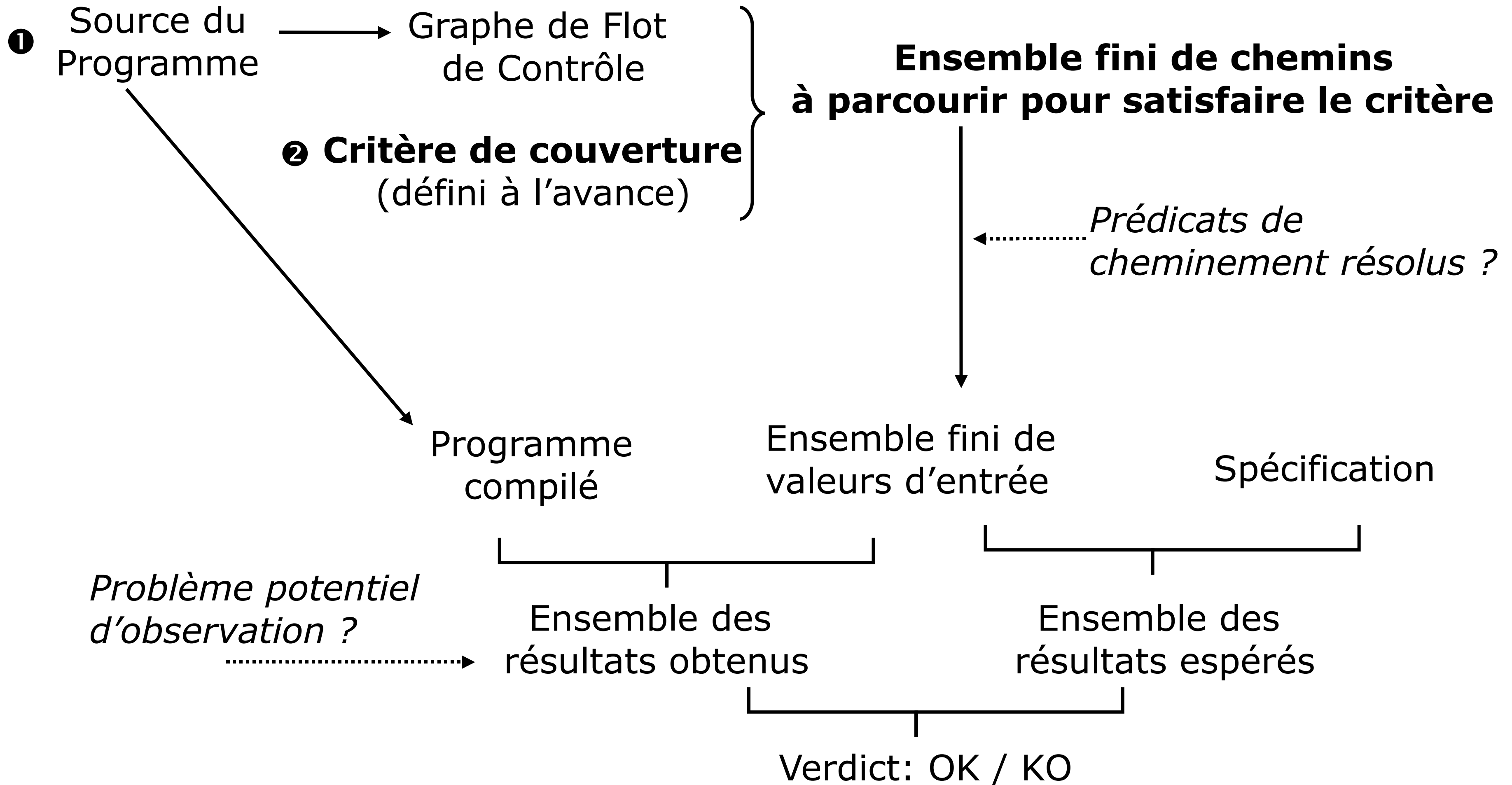
- ☞ we have again a finite number of paths
- ☞ the criterion is less constraining than $AllTransitions_k(M,T)$

A Hierarchy of Coverage Criteria

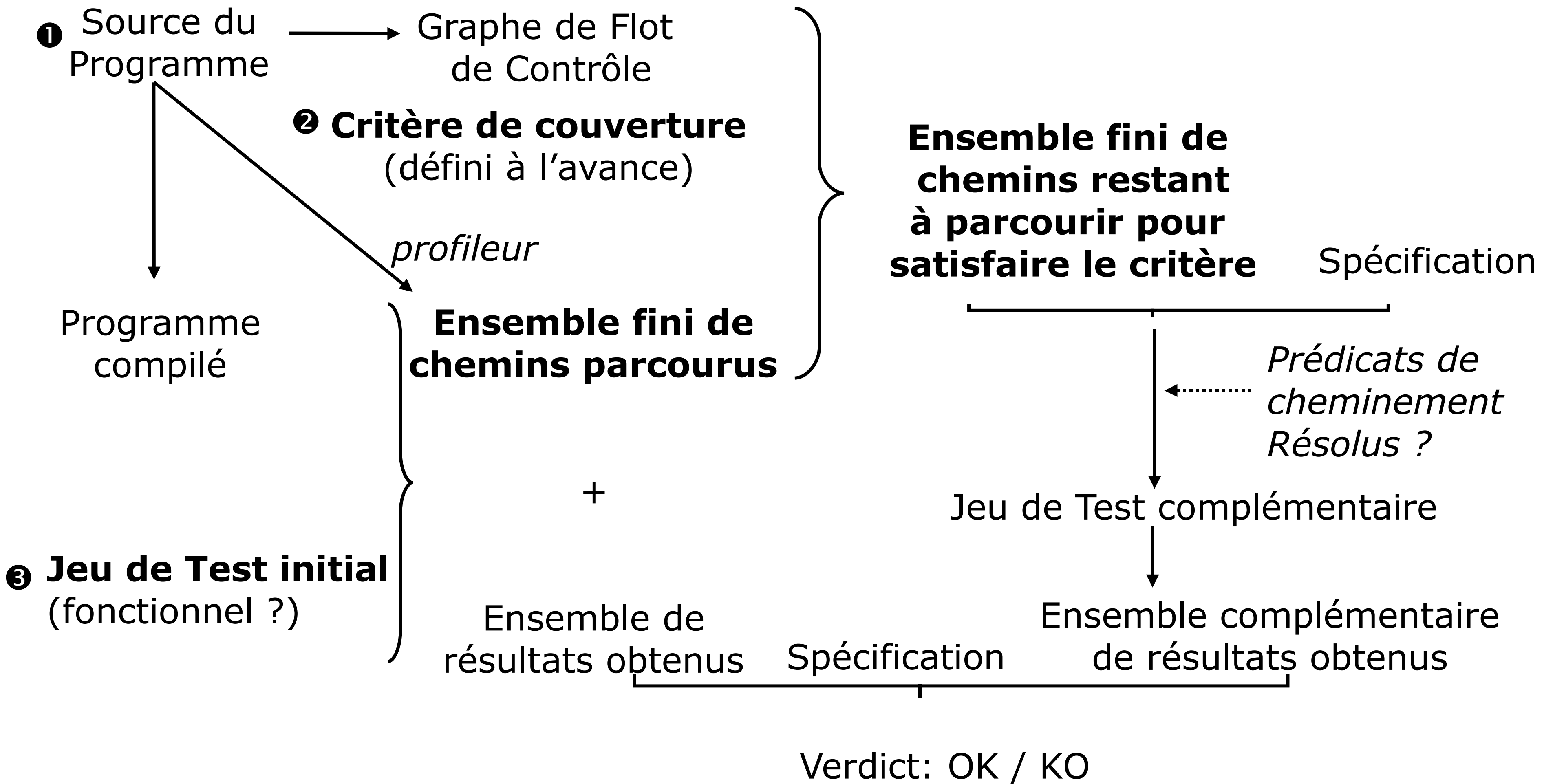
- $AllPaths(M,T) \Rightarrow$
  $AllPaths_k(M,T) \Rightarrow$
  $AllTransitions(M,T) \Rightarrow$
  $AllInstructions(M,T)$

- Each of these implications reflects a proper containement; the other way round is never true.

Using Coverage Criteria 1



Using Coverage Criteria 2



Summary

- We have developed a technique for program-based tests
- ... based on symbolic execution
- ... used in tools like JavaPathFinder-SE or Pex
- Core-Concept:
  Feasible Paths in a Control Flow Graph
- Although many theoretical negative results on key properties, good practical approximations are available
- CFG based Coverage Critieria give rise to a Hierarchy