POLYTECH
PARIS-SUD

2026

Cycle Ingénieur – 2ème année
Département Informatique

Verification and Validation

Part IV : White-Box Testing

Burkhart Wolff
Département Informatique
Université Paris-Saclay / LMF

Towards Static Specification-based Unit Test

- How can we test during development (at coding time, even at design-time ?)
- How can we test "systematically"?
  - What could be a test-generation method?
  - What could be an algorithm to generate tests?
  - What could be a coverage criterion ? (or: adequacy criterion, telling that we "tested enough")

Idea:

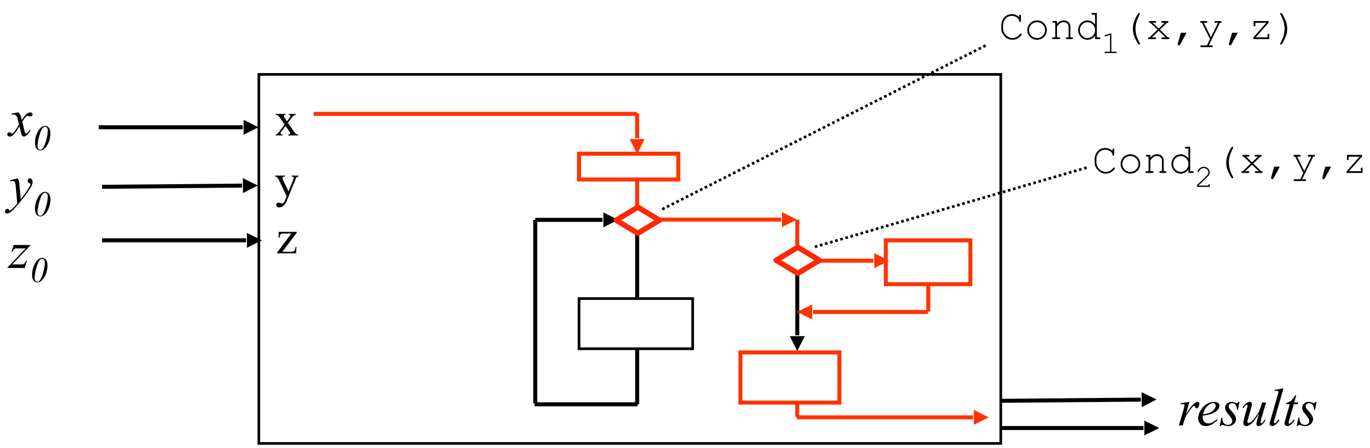
- Let's exploit the structure of the program !!!

  (and not, as before in specification based tests („black box"-tests), depend entirely on the spec).

- Assumption: Programmers make most likely errors in branching points of a program (Condition, While-Loop, ...), but get the program "in principle right". (Competent programmer assumption)

- Lets develop a test method that exploits this !

Static Structural ("white-box") Tests



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

Idea:
a path corresponds to one logical expression over initial values 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 !
  - if your algorithm is incomplete, a test on the spec has at least a chance to find this ! (Example: perm generator with 3 loops)

Functional-test vs. structural test?

Both are complementary and complete each other

- 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) → 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 <u>nodes</u> be either "elementary instruction blocs" or "decision nodes" labelled by a predicate.
- the <u>arcs</u> 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
- eliminate if_then_else's 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

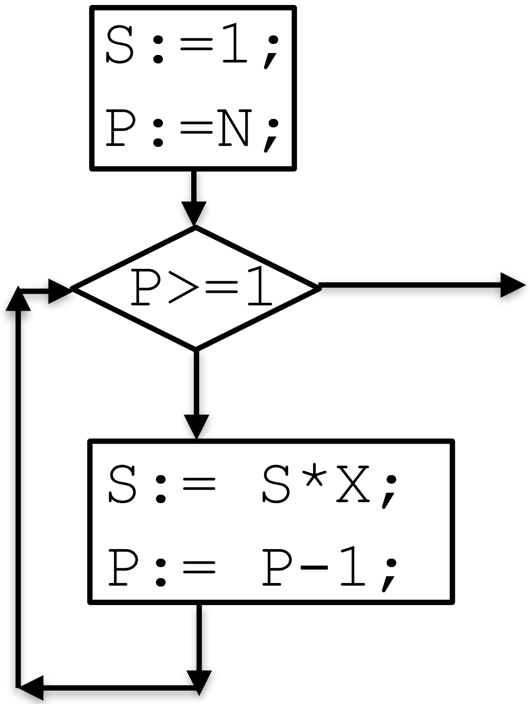
Computing Control Flow Graphs

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

Computing Control Flow Graphs

- Identify longest sequences of assignments

Example:



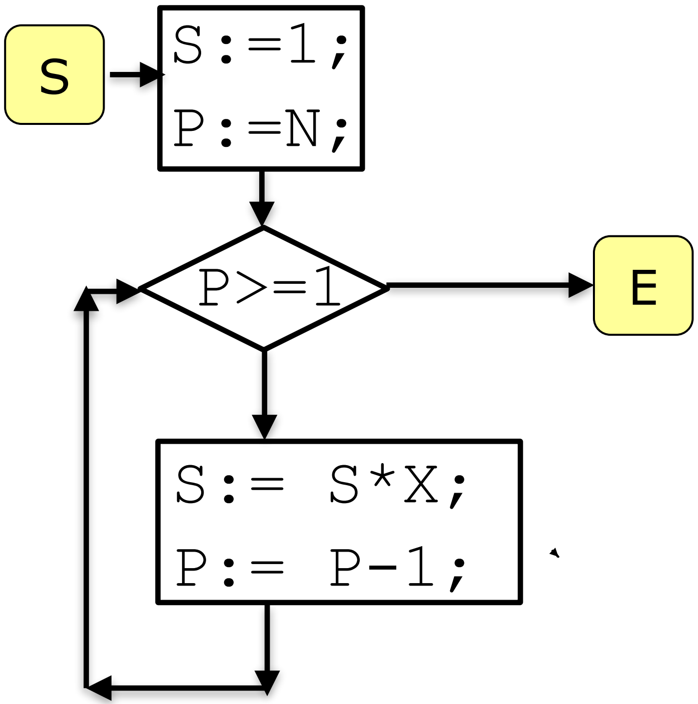
Computing Control Flow Graphs

- 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 ...

- Example:

  Add entry node and exit loop-arc, entry-arc, exit-arc



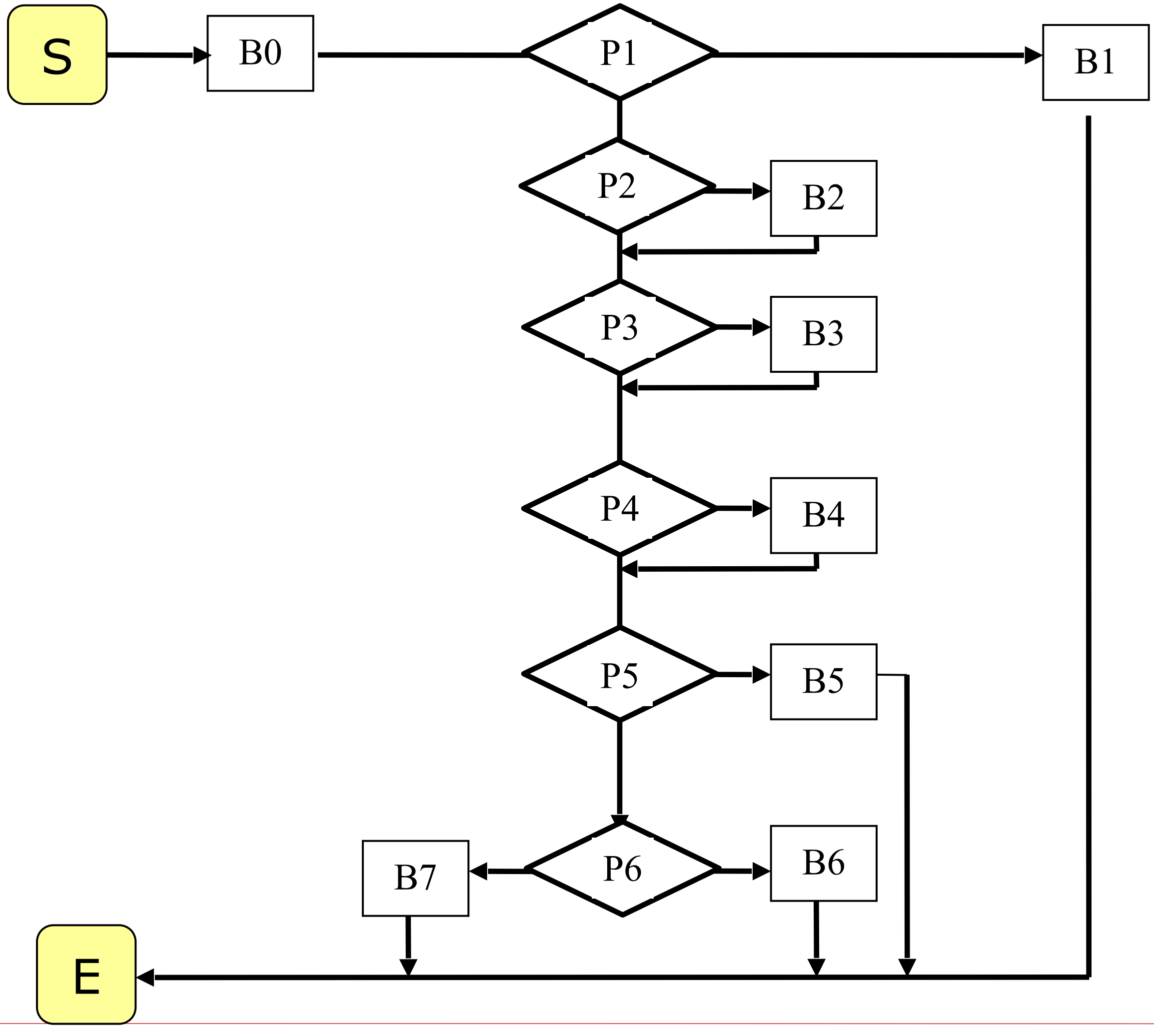
Q: What is the CFG

of the body of triangle ?

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;
```

The non-structured control-flow graph of a program

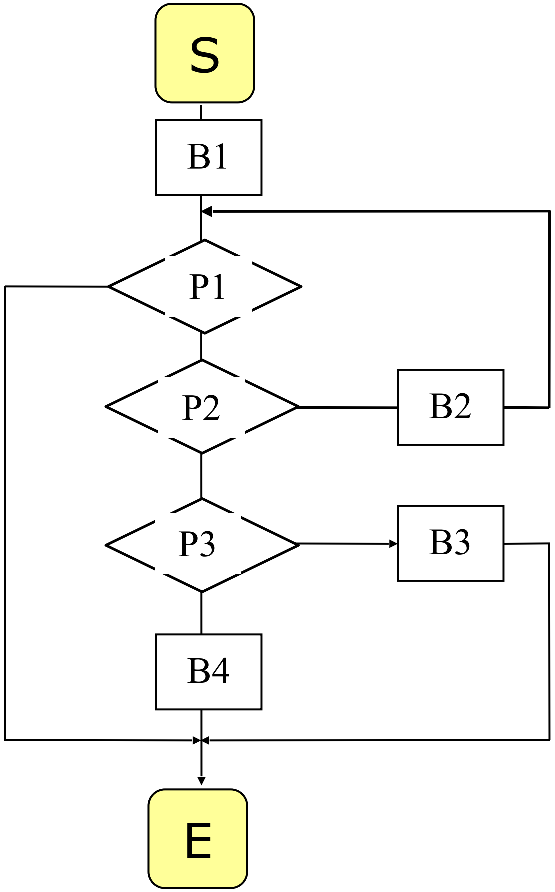


A procedure with loop and return

```
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;
```

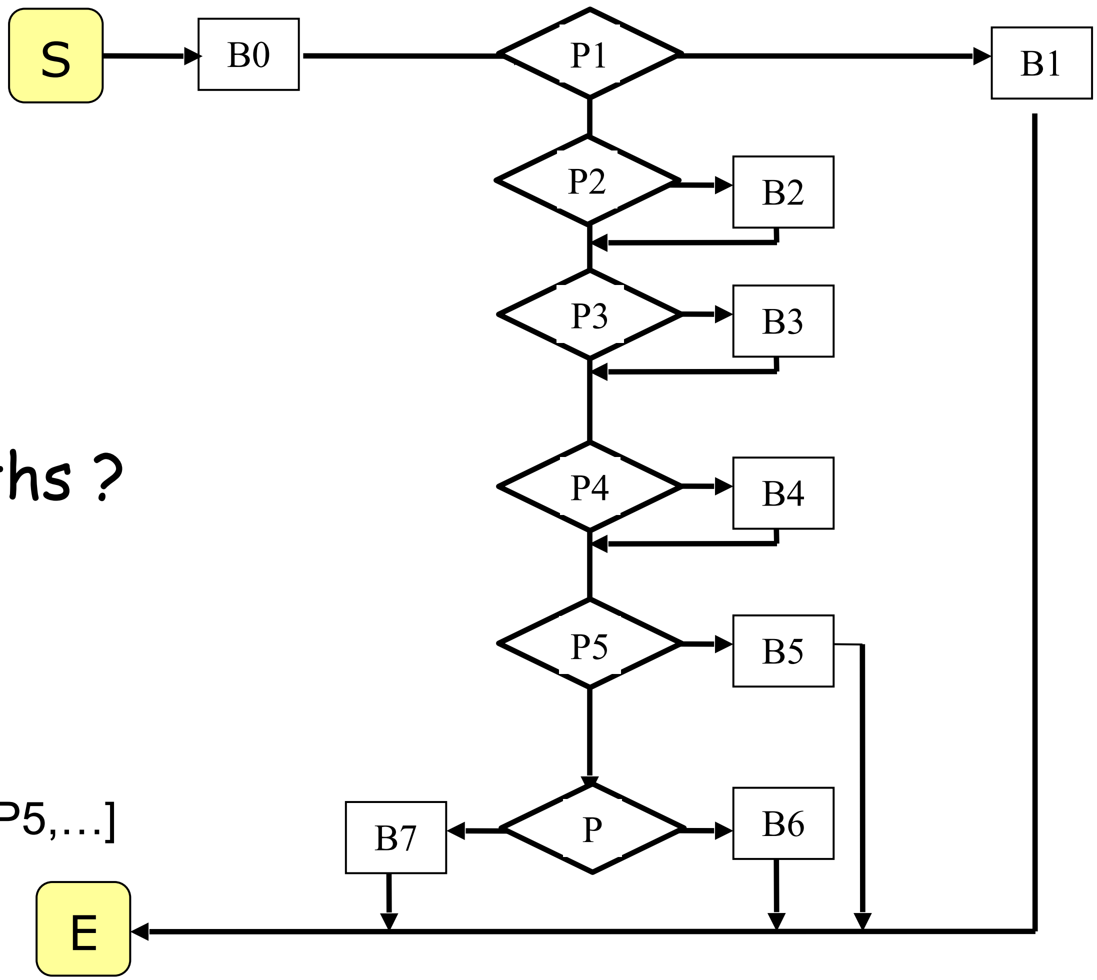
… and its control flow graph

Can we represent this program as control-graph ???



Sure …

… and its control flow graph



Are all paths actually possible executions ?

Are they feasible paths ?

Consider:

[S,B0,P1,P2,B2,P3,B3,P4,P5,…]

Paths and Path Conditions

- Some Terminology:

  - *initial path* M = path of the CFG starting at S
  - *path* of M = path of the CFG starting at S and ending in E

    (a path corresponds to a **complete** execution of the procedure)
  - for an initial path M, a predicate over the parameters and state can be defined: the path-condition Φ_M
  - Φ_M is exactly true over the **initial values initiales** of parameters (and global variables) if the program will run **exactly** M for these parameters
  - faisable paths : M is feasible exactly if a for parameters and global variables concrete values *exist* such that M is executable.

    i.e. the path condition Φ_M *is satisfiable*

Computing Path Conditions by Symbolic Execution

Let M be an initial path in the CFG of our program.

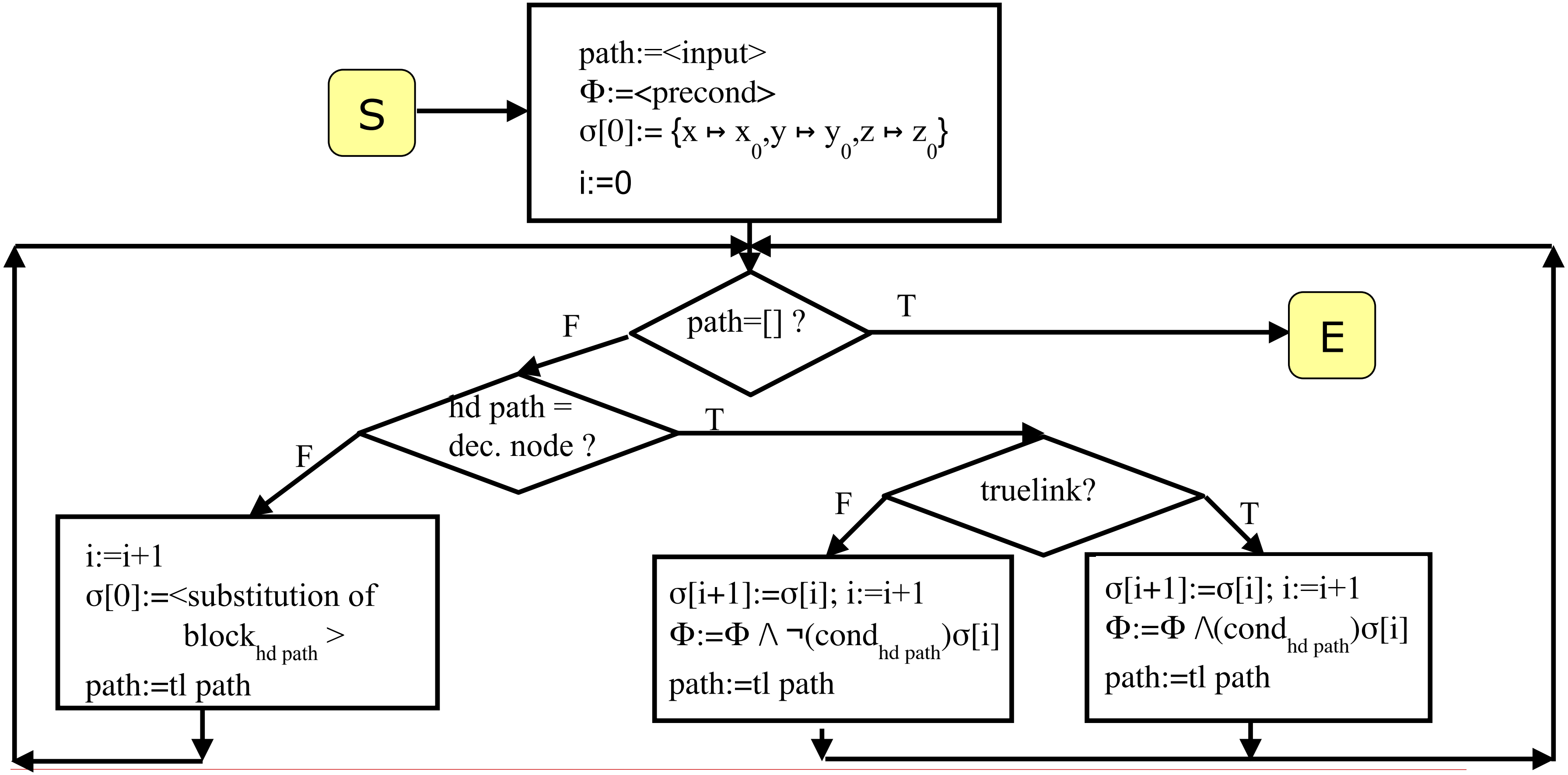
- we give symbolic values for each variable x_0, y_0, z_0, ...
- we set the path condition Φ initially to the pre-condition
- We follow the path M, block for block:
  - If the current block is an instruction block B:

    we execute symbolically B by memorising the new possible values by predicates depending on x_0, y_0, z_0, .. ("symbolically")
  - If the current block is a decision block P(x1,...,xn)
    - if we follow the « true » arc we set $\Phi := \Phi \wedge P(\underline{x1},...,\underline{xn})$,
    - if we follow the «false» arc we set $\Phi := \Phi \wedge \neg P(\underline{x1},...,\underline{xn})$.

The $\underline{x1}$,…,$\underline{xn}$ are the symbolic values for the program variables

Computing Path Conditions by Symbolic Execution

Scheme of an algorithm:



Execution

- Execution 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*.

$$\sigma = V \rightarrow D$$

- *As* usual, we denote finite functions as follows:

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

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.

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

  We denote the set of possible values by a

  predicate over the initial state, so:

  $$x \mapsto (1 \leq x_0 \wedge x_0 \leq 10)$$

- thus, after x:= x+2, we know:

  $$x \mapsto (3 \leq x_0 \wedge x_0 \leq 12)$$

Symbolic States and Substitutions

- An Example substitution:

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

$$= \ 1 + 2 * x_0$$

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

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

Basic Blocks as Substitutions

Symbolic Pre-State σ_{sym}		
x	↦	x_0
y	↦	y_0+3*x_0
z	↦	z_0
i	↦	i_0

Block

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

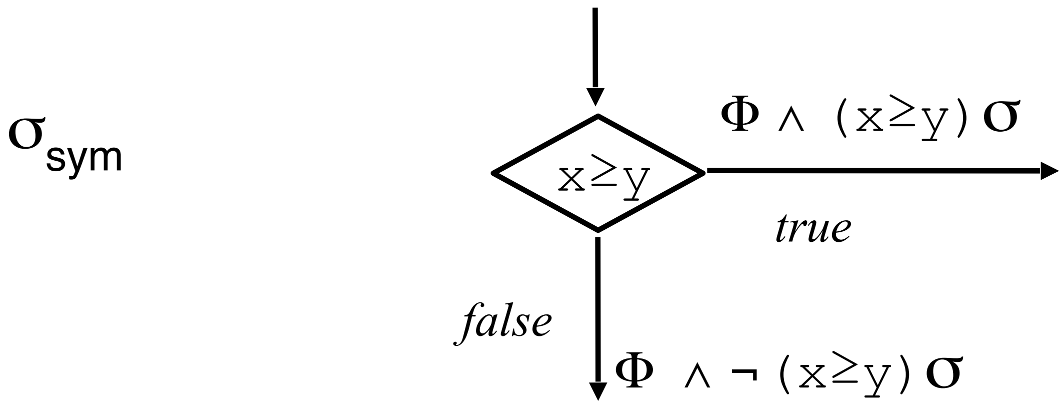
Symbolic Post-State σ'_{sym}		
x	↦	x_0
y	↦	y_0+3*x_0
z	↦	$z_0+y_0+4*x_0+1$
i	↦	$y_0+ 4*x_0+1$

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

i is supposed to be a un-initialized local variable.

Thus, we update the symbolic state whenever we pass a basic block on our path.

Symbolic Execution



Thus, we update the path-condition whenever we pass a decision node on our path.

Example: A Symbolic Path Execution

Recall

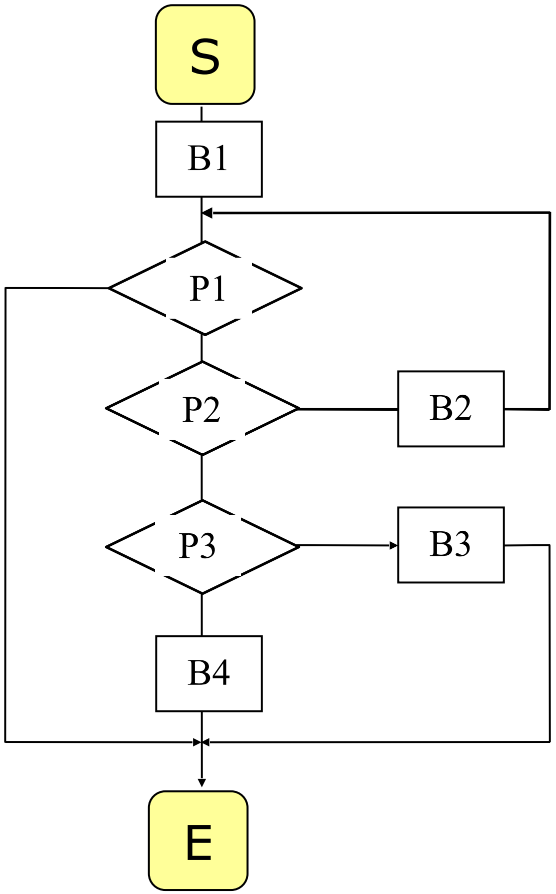
```
procedure supprime (T: in out Table; p: in out integer;
                    x: in integer) is
 i: integer := 1;
begin
   while  i <> p  loop
      if  T[i] <> x then  i := i + 1;
      elsif  i = p - 1 then  p := p - 1; return;
      else   T[i] := T[p-1];     p := p - 1; return;
      end if;
 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

Example: A Symbolic Path Execution

... and the corresponding control flow graph.

We want to execute the path:

[S,B1,P1,E]



Example: A Symbolic Path Execution

We want to execute the path:

[S,	B1,	P1,	E]
$\Phi \mapsto$ True	$\Phi \mapsto$ True	$\Phi \mapsto \neg(i<>p)\,\sigma_{B1}$	$\Phi \mapsto 1 = p_0$
$T \mapsto T_0$	$T \mapsto T_0$	$T \mapsto T_0$	$T \mapsto T_0$
$p \mapsto p_0$	$p \mapsto p_0$	$p \mapsto p_0$	$p \mapsto p_0$
$x \mapsto X_0$	$x \mapsto X_0$	$x \mapsto X_0$	$x \mapsto X_0$
$i \mapsto i_0$	$i \mapsto 1$	$i \mapsto 1$	$i \mapsto 1$

Example: A Symbolic Path Execution

Result:

Test-Case:

For the path M=[S,B1,P1,E]

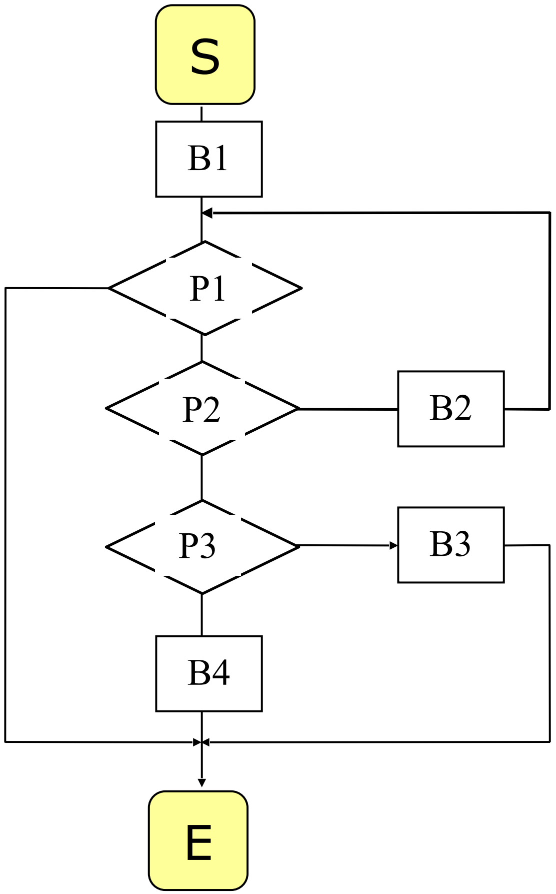
we have the path condition $\Phi \mapsto p_0 = 1$

A concrete Test, satisfying Φ

T	↦	mtTab
p	↦	1
x	↦	17

Example: A Symbolic Path Execution

... and the corresponding control flow graph.



We want to execute the path:

[S,B1,P1,P2,B2,P1,E]

Example: A Symbolic Path Execution

We want to execute the path:

[S, B1, P1, P2, B2, P1, E]

$\Phi \mapsto$ True	True	$(i<>p)\sigma_{B1} \equiv p_0 \neq 1$	$p_0 \neq 1 \wedge (T[i] \neq x)\sigma_{B1}$	$p_0 \neq 1 \wedge T_0[1] \neq x_0$	$p_0 \neq 1 \wedge T_0[1] \neq x_0 \wedge \neg (i<>p)\sigma_{B2}$	$p_0 \neq 1 \wedge T_0[1] \neq x_0 \wedge 2 = p_0$
$T \mapsto T_0$	T_0	T_0	T_0	T_0	T_0	T_0
$p \mapsto p_0$	p_0	p_0	p_0	p_0	p_0	p_0
$x \mapsto x_0$	x_0	x_0	x_0	x_0	x_0	x_0
$i \mapsto i_0$	1	1	1	$(i+1)\sigma_{B1}$	2	2

Example: A Symbolic Path Execution

Result: Test-Case for Path

M = [S,B1,P1,P2,B2,P1,E]

Path Condition: $\Phi := T_0[1] \neq X_0 \wedge p_0 = 2$

A concrete Test, satisfying Φ

$T \mapsto [3]$	
$p \mapsto 2$	
$x \mapsto 17$	

Paths and Test Sets

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

- a test case ≈ a path M in the CFG
  - = a collection of values for variables (params and global)
    (+ the output values described by the specification)

- a test case set ≈ a finite set of paths of the CFG
  = 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; this is worse than "we know none" !~**

BUT: for many relevant programs, practically good solutions exist (Z3, Simplify, CVC4, AltErgo ...)

A Challenge-Example (The Collatz-Function):

... **A HAIRY EXAMPLE:**

```
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 ***end loop*** reached for all x ?

ANSWER : unknown

- this implies that we can not always know that infeasible paths exist !

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;
```

- ☞ In the contrary, there are programs where all paths are feasible
- ☞ That is rare, however.
- ☞ Worse: in practice the probability for a path to be feasible is smaller the longer the path gets.

The notion of a "coverage criterion"

A coverage criterion is a function mapping a CFG
to a particular subset of its paths ...

- the set of paths covering all basic blocks
- the set of paths covering all instructions
- the set with all loops are traversed
- a particular subset of calls/labels occurring in the CFG has been covered
- ...

Well-known Coverage Criteria I

Criterion C = AllInstructions(CFG):

For all nodes N in CFG (basic instructions or decisions) exists a path in C that contains N

Well-known Coverage Criteria II

Criterion C = AllTransitions(CFG):

For all arcs A in the CFG exists a path in C that uses A

Well-known Coverage Criteria III

Criterion C = AllPaths(CFG):

All possible paths ...

☹ Whenever there is a loop, C is infinite !

☞ weaker variant: AllPaths_k(CFG).

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

☞ we have again a finite number of paths

A Hierarchy of Coverage Criteria

- AllPaths(CFG) $\supseteq$

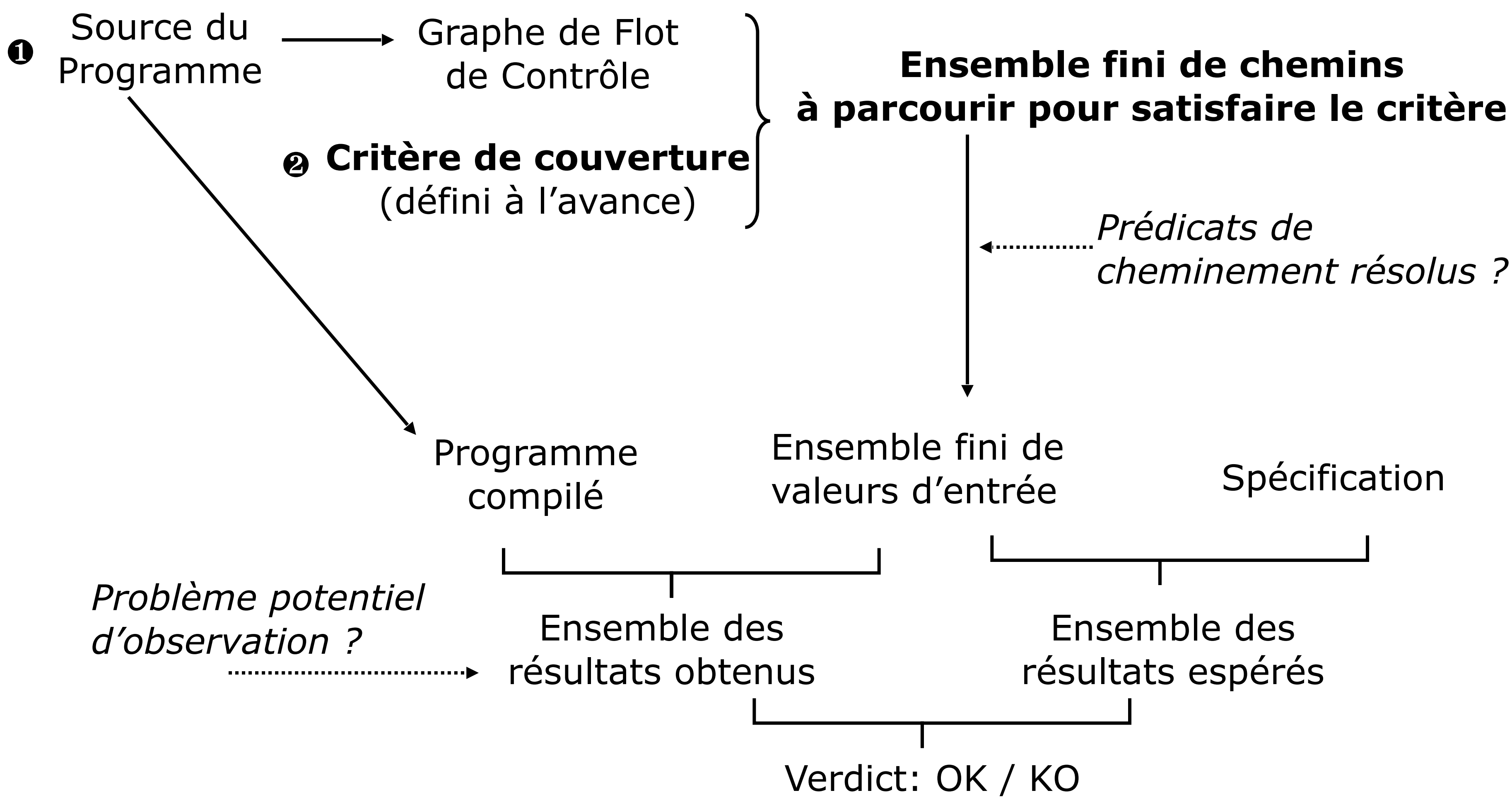
  AllPaths$_k$(CFG) $\supseteq$

  AllTransitions(CFG) $\supseteq$

  AllInstructions(CFG)

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

Using Coverage Criteria 1



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

Schmankerle

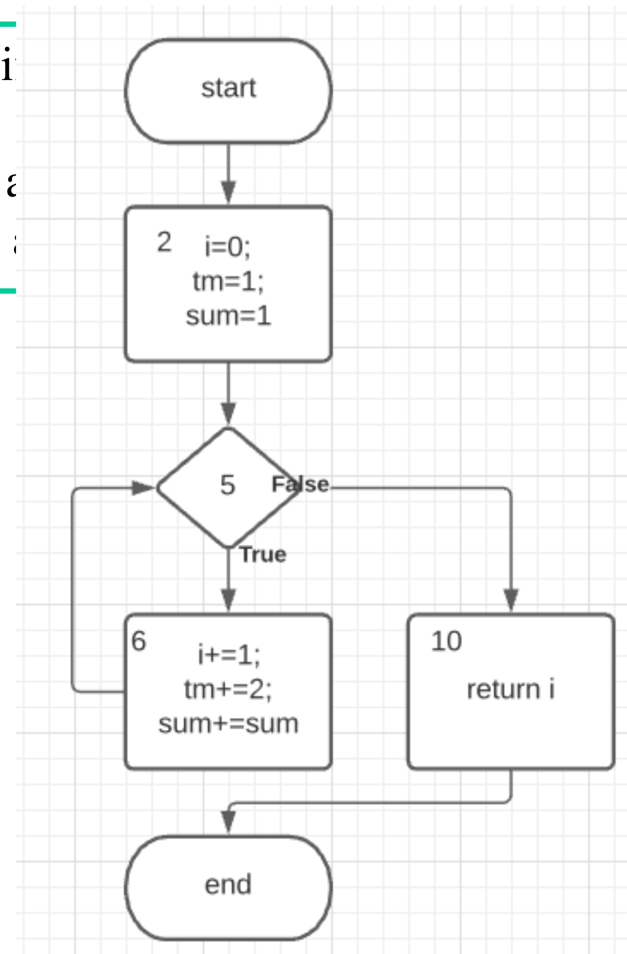
- Program:

```
int f (int a) {
    int i = 0;
    int tm = 1;
    int sum = 1;
    while(sum <= a) {
        i = i+1;
        tm = tm+2;
        sum = tm+sum;
    }
    return i;
}
```

Speci

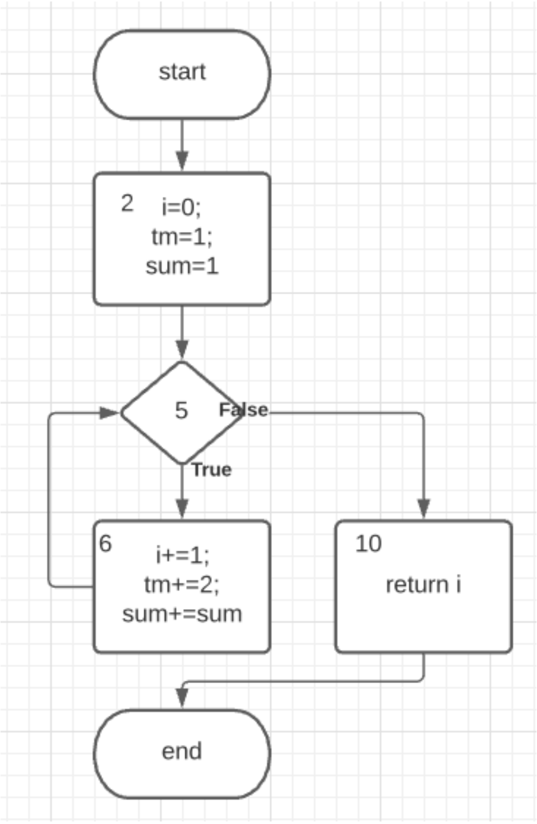
pre : a

post: a



Schmankerle

- CFG de f:



- For example:

AllInstructions(CFG)={[start,2,5,6,5,10,end]}

AllTransitions(CFG)={[start,2,5,6,5,10,end]}

$AllPath_3$(CFG)={[start,2,5,10,end],
[start,2,5,6,5,10,end],
[start,2,5,6,6,5,10,end],
[start,2,5,6,6,6,5,10,end]}

AllPath(CFG)={ $k \in \mathbb{N}$ |
[start,2,5,$(6)^k$,5,10,end]}
(infinite !)

Example: A Symbolic Path Execution

We want to execute the path from AllPath_3:

[S, 2, 5, 6, 5, 10, E]

$\Phi \mapsto a_0 \geq 0$	$a_0 \geq 0$	$(sum \leq a)\sigma_2 \wedge a_0 \geq 0$	$1 \leq a_0 \wedge a_0 \geq 0$	$1 \leq a_0 \wedge \neg(sum \leq a)\sigma_6$	$1 \leq a_0 \wedge 4 > a_0$	$1 \leq a_0 \wedge 4 > a_0 \wedge res = 1$
$a \mapsto a_0$	a_0	a_0	a_0	a_0	a_0	a_0
$i \mapsto i_0$	0	0	1	1	1	1
$tm \mapsto tm_0$	1	1	3	3	3	3
$sum \mapsto sum_0$	1	1	4	4	4	4

Example: A Symbolic Path Execution

Result:

Test-Case:

For the path M=[start,2,5,6,5,10,end]

we have the path condition $\Phi \mapsto 1 \leq a_0 \land 4 > a_0$

A concrete Test, satisfying Φ: $a_0 \mapsto 3$

Execution of program with this test vector 3: f(3) = 1

Verification of the post-condition: post(3, 1) = true

Addendum: Multiple-Condition-Decision-Coverage

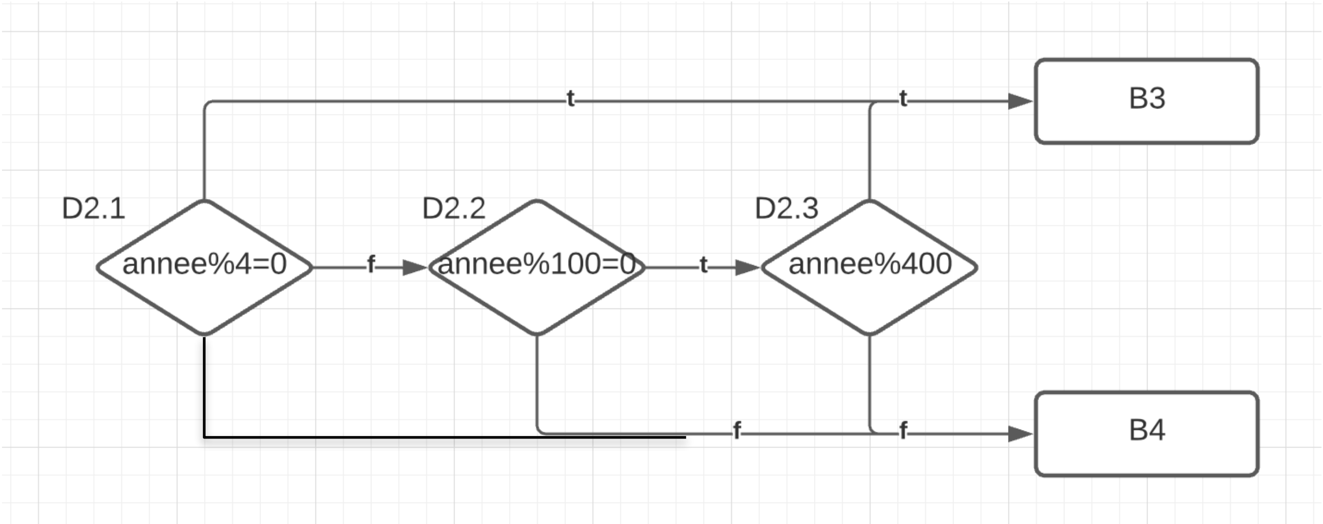
Problem: Consider:

```
if(m1 == m2) {
    res = j2 - j1;
} else {
    if((annee%4 == 0) || (annee%100 == 0 && annee%400 != 0)) {
        daysin[2] = 29;
    } else {
        daysin[2] = 28;
    }
    res = j2 + (daysin[m1] - j1);
    for(int i = m1+1; i < m2; i++) {
        res = res + daysin[i];
    }
}
```

even transition coverage on the Byzantine condition (line 4-5) is very coarse and risks to miss the point !

Addendum: Multiple-Condition-Decision-Coverage

Solution: We use the inherent control flow in C for || and &&
for a refined control flow graph !



now transition coverage (on the refined CFG) checks
each condition individually for true and false.
This kind of coverage is called MC/DC.