Modélisation & Vérification

Test Structurel

Asma BERRIRI

asma.berriri@universite-paris-saclay.fr

Page web du cours :
https://sites.google.com/view/asmaberriri/home

Outline

Structural testing

- *Functional testing* : (also *black-box* testing).
  Les tests sont générés sur une spécification du composant, le test se concentre sur le comportement d'entrée/sortie.
  Visualise le programme comme une boîte noire

```
          -->              -->
  input   -->  program     -->  output
          -->              -->
```

  Note: Nous pouvons faire des tests "boîte noire" du programme en faisant des tests "boîte blanche" de la spécification
- *Structural testing* : (also *white-box* testing).
  Les tests sont générés sur la base de la structure ou du programme, c'est-à-dire en utilisant des chemins de flux de contrôle, de flux de données ou en utilisant des exécutions symboliques.
  Les tests structurels considèrent un programme comme une boîte blanche
- both: (also: grey-box testing)

Structural testing

- Tests structurels axés sur la logique interne du système
- Les cas de test sont développés avec une connaissance des détails de l'implémentation
- La spécification est toujours nécessaire pour connaître la sortie attendue
- Ne détecte pas les fonctions manquantes (décrites dans la conception/spécification fonctionnelle mais non implémentées)
- Différentes méthodes de test en boîte blanche
  - Dépend de l'objectif du test (logique de couverture)

-> Exploitons la structure du programme !!!

-> Hypothèse : Les programmeurs font très probablement des erreurs dans les points de branchement d'un programme (Condition, While-Loop, ...)

-> Développons une méthode de test pour vérifier cela !

Nous nous concentrerons sur la sélection et la couverture des cas de test

POLYTECH PARIS-SACLAY

A Program for the triangle example

```
proceduretriangle(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;
```

POLYTECH PARIS-SACLAY

Structural Test Criteria

What are tests adapted to this program?

- ? essayer un certain nombre de chemins d'exécution (lesquels? tous?)
- ? trouver des valeurs d'entrée pour stimuler ces chemins
- ? comparer les résultats avec les valeurs attendues (c'est-à-dire la spécification)

Les critères de test structurels aideront à sélectionner les cas de test à partir du code du programme. Critères basés sur:

Flux de contrôle Flux de données Fautes "attendus"

- Défini formellement en termes de graphes de flux
- Metric : pourcentage de couverture atteint
- Adéquation basée sur les exigences métriques pour les critères

Objectif : Couvrir la structure du logiciel

POLYTECH PARIS-SACLAY

Control-flow based Coverage

1. Control-flow based Coverage

Control flow graph

Oriented graph giving a compact and abstract view of the program control structure:

1. built from the program source code
2. a node = a maximal block of consecutive statements i_1, ...i_n
   - i_1 is the unique access point to the block
   - the statements are always executed in the order i_1, ...i_n
   - the block is exited after the execution of i_n
3. edges between nodes = conditional or unconditional branching (if then else, while loops, loops)
4. entry node E, and ending node S: unique, clearly identified

POLYTECH PARIS-SACLAY

Control flow graph properties

Control Flow Graph (CFG) properties

- Connected graph (a path from n_1 to n_2, for each pair of nodes (n_1, n_2)
- All nodes are reachable from E
- Execution path : a path from E to S
- Each path represents (a set of) executions of the program control flow graph

POLYTECH PARIS-SACLAY

Control flow graph

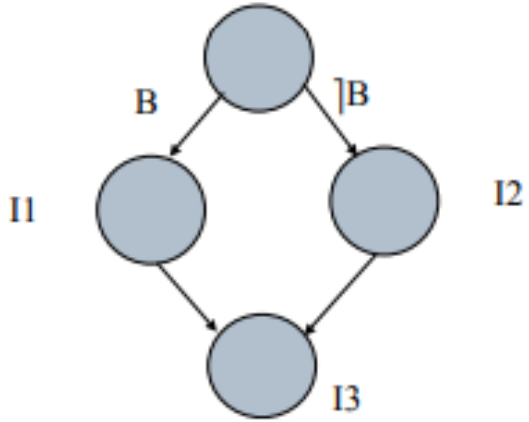
A graph with oriented edges root E and an exit S,

- the nodes be either *elementary instruction blocks* or *decision nodes* labelled by a predicate.
- the arcs indicate the control flow between the elementary instruction blocks and decision nodes (control flow)
- all blocks of predicates are accessible from E and lead to S (otherwise, dead code is to be suppressed !)
- elementary instruction blocks: a sequence of assignments
  - update operations (on arrays, ..., not discussed here!!)
  - procedure calls (not discussed here !!)

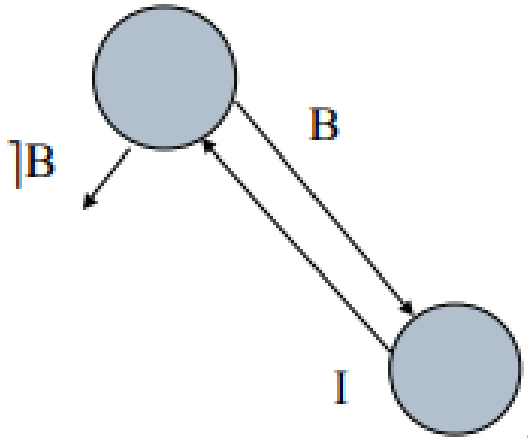
POLYTECH PARIS-SACLAY

Some Patterns for Computing Control Flow Graphs

- Sequence of assignments : a single node
- If B then $I1$ else $I2$ endIf ; $I3$:



- If B then $I1$ endIf; $I3$: ????
- While B loop I endLoop:



Computing Control Flow Graphs

- Identify longest sequences of assignments
- Erase if then else instructions by branching
- Erase while loops by loop arc, entry arc, exit arc

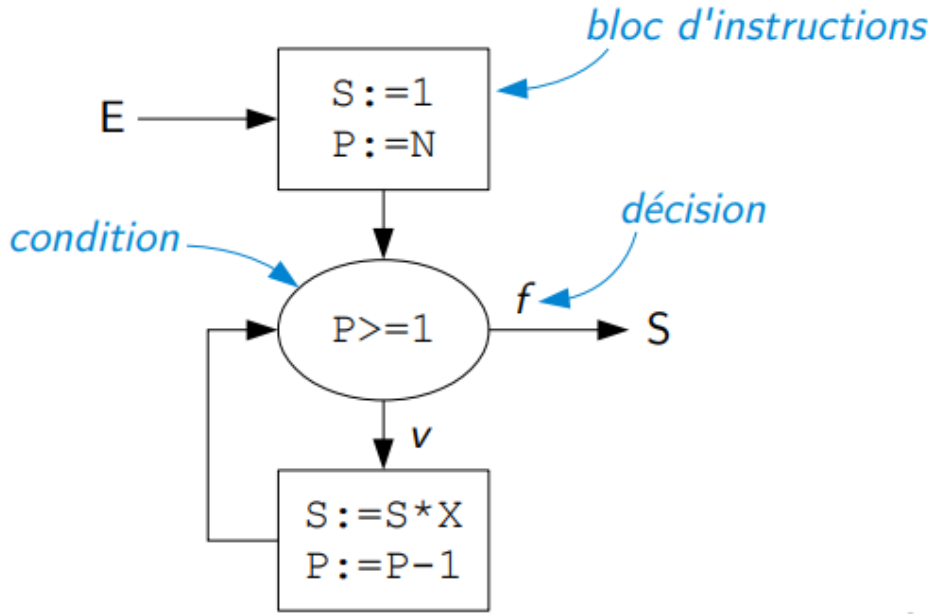
Example:

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

What is the control flow graph ?

Computing Control Flow Graphs

```
S:=1;
P:=N;                    (a)
while P >= 1 loop        (b)
    S:= S*X;
    P:= P-1;             (c)
end loop;
```



- (a) the node associated to the first sequence `S:=1;P:=n`
- (b) the node corresponding to the loop condition
- **(c)** the node associated to the sequence `S:=S*X;P:=P-1` (body of the loop)

POLYTECH PARIS-SACLAY

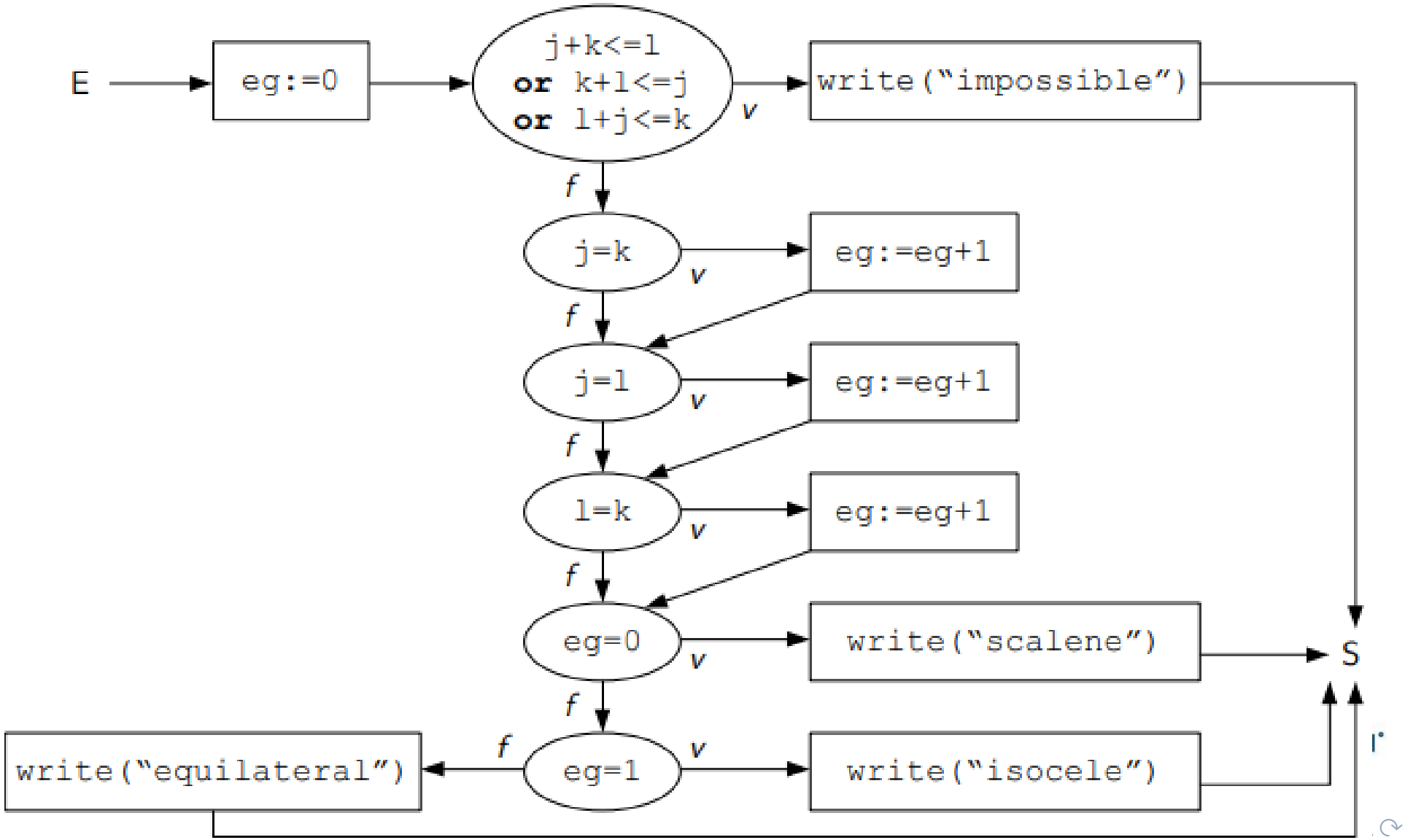
Computing Control Flow Graphs: Exercices

```
triangle(j,k,l : positive):void
    eg := 0;
    if j + k <= l or k + l <= j or l + j <= k
    then write("impossible");
    else if j = k then eg := eg + 1; endif;
        if j = l then eg := eg + 1; endif;
        if l = k then eg := eg + 1; endif;
        if eg = 0 then write("scalene");
        elsif eg = 1 then write("isocele");
        else write("equilateral"); endif;
    endif;
```

- What is the CFG of the body of triangle ?

Computing Control Flow Graphs: Exercices

- What is the CFG of the body of triangle ?



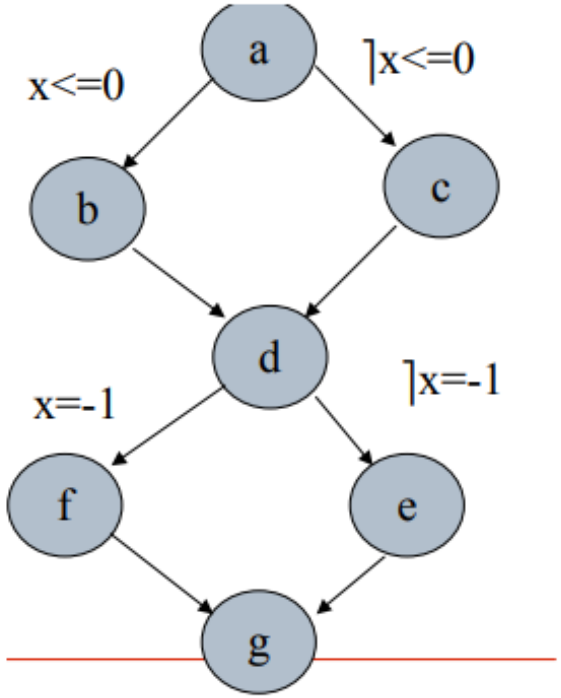
Paths and Path Predicates/Conditions

Let M be a procedure to test, and G (S entry node, E end node) its control-flow graph.

- a path of M: path of G starting at S and leading to E, i.e. a complete execution of the procedure
  - Many paths between the entry and exit points of a typical routine
  - Even a small routine can have a large number of paths
- a program execution: activation of a path in the CFG
  - every path corresponds to a succession of true/false values for the predicates traversed on that path
- predicate (over parameters and states): a condition over the initial values of parameters (and global variables) to achieve exactly this execution path
- path predicate: set of predicates associated to a path
- For a given path *ch*
  - if there exist input data for which the program execution sensitizes *ch* (the associated path predicate is satisfiable), then the path is said to be executable or feasible
  - Otherwise, the path is unfeasible

Execution Paths

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```

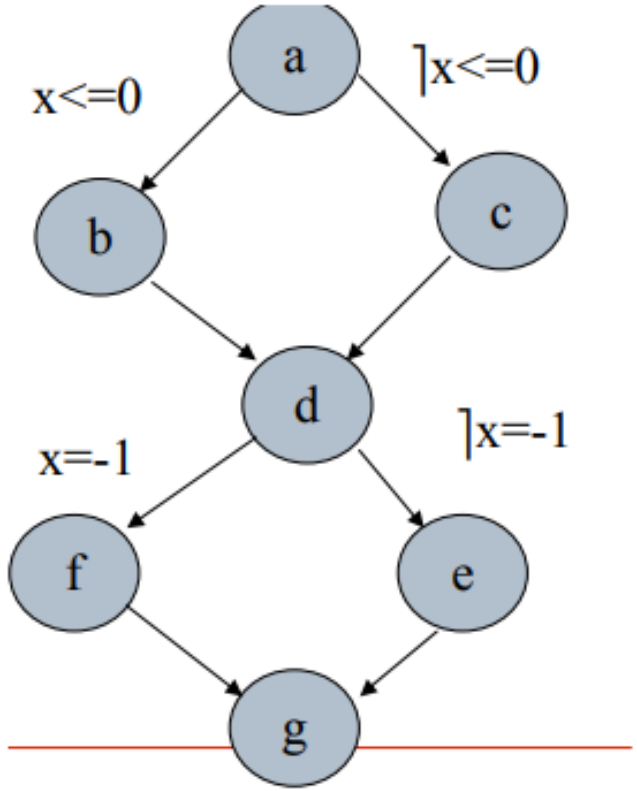


$ch1 = [a, b, d, f, g]$
$ch2 = [a, b, d, e, g]$
$ch3 = [a, c, d, f, g]$
$ch4 = [a, c, d, e, g]$
Paths denoted by regular expressions :
$ch1 + ch2 + ch3 + ch4 = a(b + c)d(f + e)g$

- $+$ for the choice (or)
- . (or nothing) for the concatenation
- ϵ for the empty path
- Exponent $+$ for a finite repetition(at least an element)
- Exponent * for a finite repetition
- For the expression *WHILE b DO I OD*, the path is $chb(chI\ chb)^*$

Execution Paths

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```



- ch1 = abdfg feasible ? no. Following this path requires that a positive number may be equal to -1 !
- ch2 = abdeg feasible ? yes for input $x = 0$, the path is performed
- ch3 = acdfg feasible ? yes for input $x = 2$, the path is performed
- ch4= acdeg feasible ? yes for input $x = 1$, the path is performed

POLYTECH PARIS-SACLAY

Predicate associated to a path

- The difficulty is to find values for inputs that will activate a given path
- The idea is to find some characterization of the path according to the value of the input data

$\longrightarrow$ a path predicate expression / path condition (prédicat de chemin) is a Boolean expression that characterizes the set of input values that will cause a path to be traversed.

- We want to characterize an execution path c by a predicate on the inputs x, y, z : $P_c(x, y, z)$
- A solution of $P_c(x, y, z) = (x = x_0, y = y_0, z = z_0)$ such that $P_c(x_0, y_0, z_0)$ is true
  - When we execute the program with x_0, y_0 and z_0 as input values, the execution follows path c
  - Finding a solution that satisfies a predicate : in all its generality is an undecidable problem
- For each path c, a predicate, called path predicate, and noted Pc, can be defined on input data such that an input data d sensitizes the path c if and only if it satisfies the path predicate associated to c, (i.e. $Pc(d)$ is true)

Symbolic evaluation/execution along a path

Undecidability ...be careful

- In general, it is undecidable if 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 mathematically proven that there is no algorithm

BUT: for many relevant programs, practically good solutions

POLYTECH PARIS-SACLAY

Path predicates : example abdfg

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```

- Let x_0 be the value of x at the beginning
- As we take the (b) branch, we have $x_0 \leq 0$
- The value of x becomes $-x_0$
- As we have to execute (f) it means at (d) the condition is true: so we must have $-x_0 = -1$ ie $x_0 = 1$
- Then executing (f) the value of x becomes 1
- So the path predicate (gathering all the constraints on x_0) is $x_0 \leq 0 \wedge x_0 = 1$
- We rephrase with x : $P_{abdeg}(x) = x \leq 0 \wedge x = 1$, which is always false, there is no solution for the path predicate so the path is not feasible/executable

POLYTECH PARIS-SACLAY

Path predicates : example abdeg

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```

- Let x_0 be the value of x at the beginning
- As we take the (b) branch, we have $x_0 \leq 0$
- The value of x becomes $-x_0$
- As we have to execute (e) it means at (d) the condition is false: so we must have $-x_0 \neq -1$ ie $x_0 \neq 1$
- Then executing (e) the value of x becomes $-x_0 + 1$
- So the path predicate (gathering all the constraints on x_0) is $x_0 \leq 0 \wedge x_0 \neq 1$
- We rephrase with x : $P_{abdeg}(x) = x \leq 0 \wedge x \neq 1$, which can be simplified in $P_{abdeg}(x) = x \leq 0$, So we can find a solution, e.g. $x = -1$ or $x = 0$ etc. The path is feasible.

Path predicates : example acdfg

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```

- Let x_0 be the value of x at the beginning:
- As we take the (c) branch, we have $\neg(x_0 \leq 0)$ i.e, $x_0 > 0$
- The value of x becomes $1 - x_0$
- As we have to execute (f) it means at (d) the condition is true: so we must have $1 - x_0 = -1$ ie $x_0 = 2$
- Then executing (f) the value of x becomes 1
- So the path predicate (gathering all the constraints on x_0) is $x_0 > 0 \wedge x_0 = 2$
- We rephrase with x : $P_{acdfg}(x) = x > 0 \wedge x = 2$, which can be simplified in $P_{acdfg}(x) = (x = 2)$, So we can find a solution, e.g. $x = 2$ (the only one). The path is feasible.

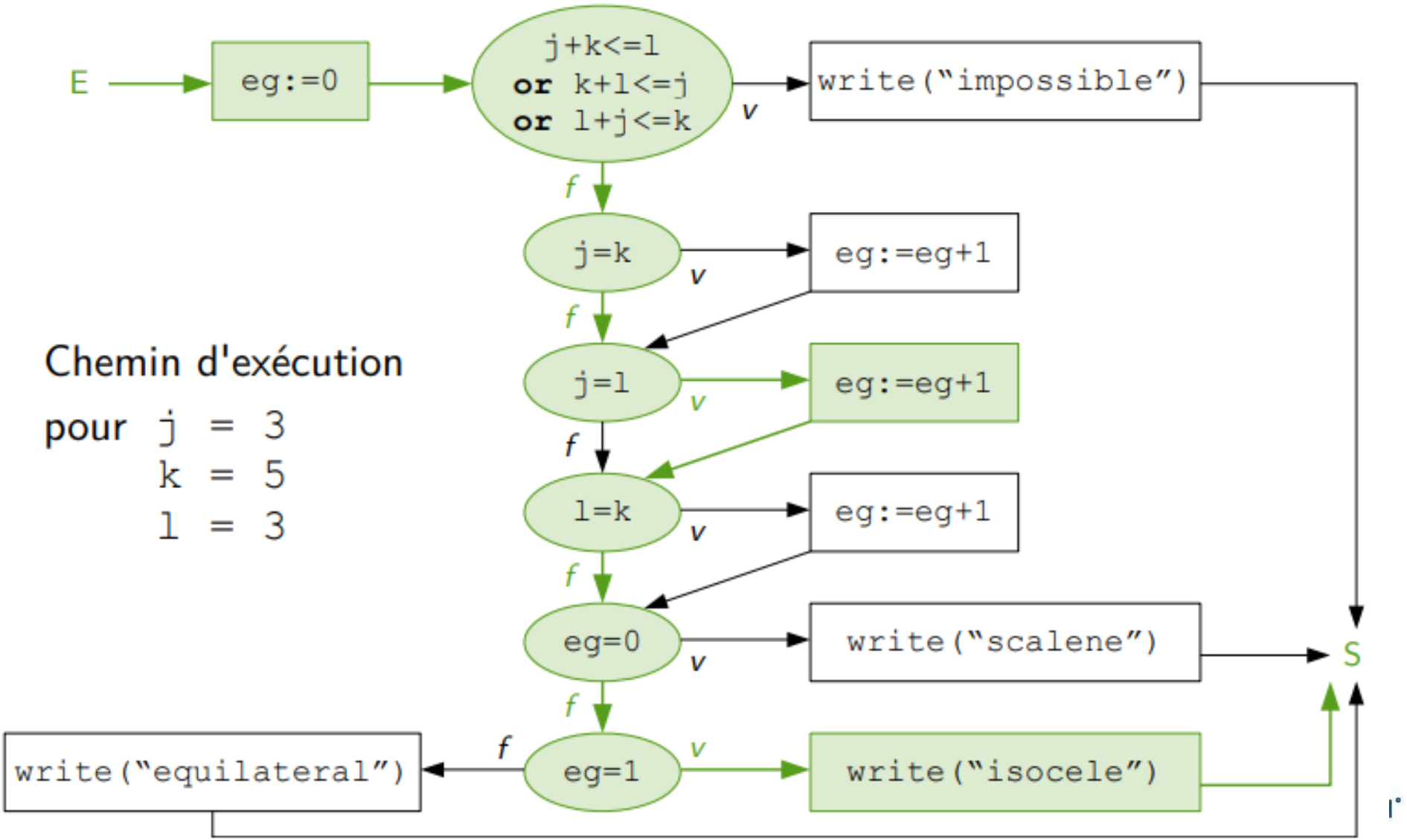
Path predicates : example acdeg

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```

- Let x_0 be the value of x at the beginning:
- As we take the (c) branch, we have $\neg(x_0 \leq 0)$ i.e, $x_0 > 0$
- The value of x becomes $1 - x_0$
- As we have to execute (e) it means at (d) the condition is false : so we must have $1 - x_0 \neq -1$ ie $x_0 \neq 2$
- Then executing (e) the value of x becomes $(1 - x_0) + 1$ ie $2 - x_0$
- So the path predicate (gathering all the constraints on x_0) is $x_0 > 0 \wedge x_0 \neq 2$
- We rephrase with x : $P_{acdeg}(x) = x > 0 \wedge x \neq 2$, So we can find a solution, e.g. $x = 1$. The path is feasible.

POLYTECH PARIS-SACLAY

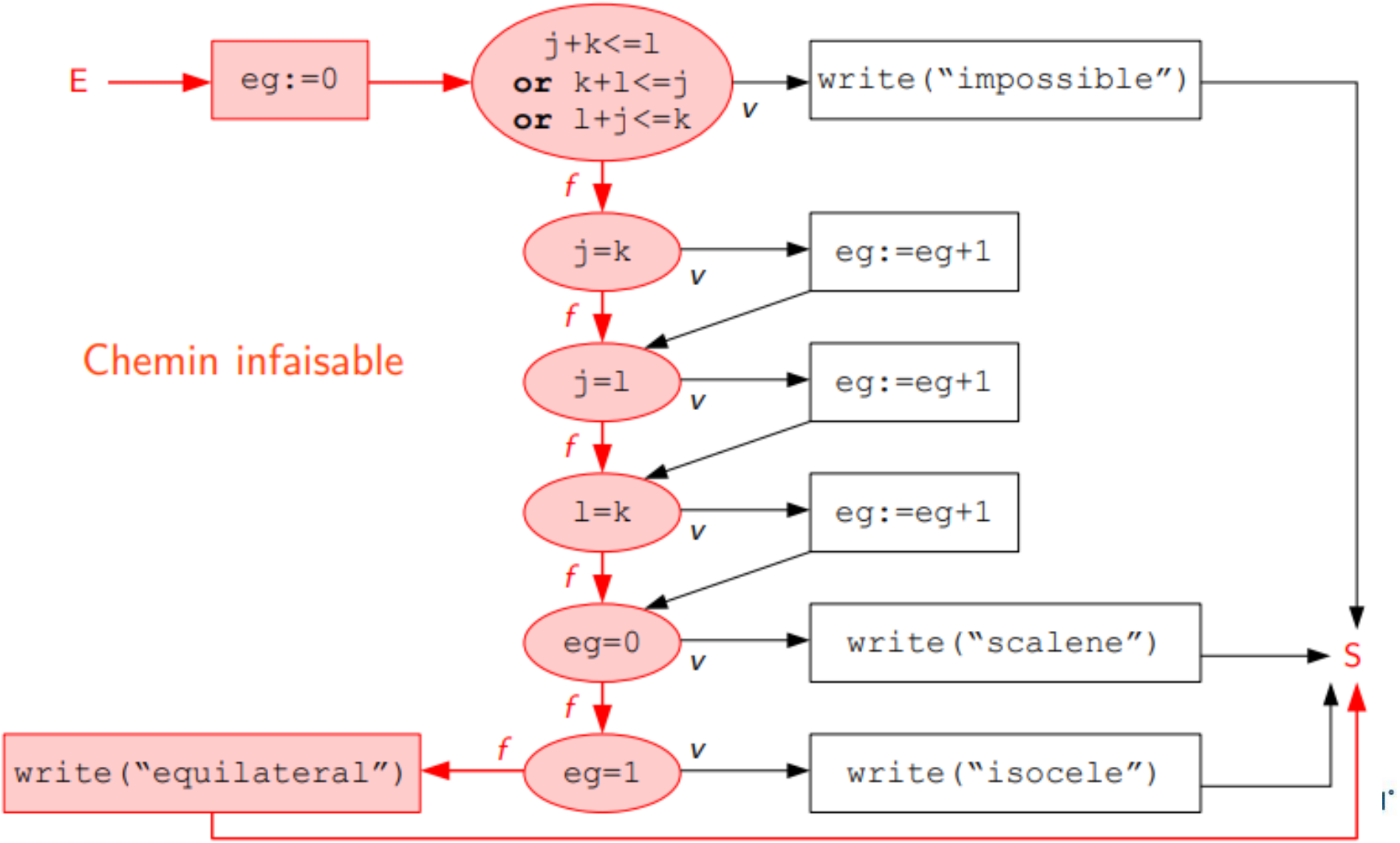
Path predicates : Triangle



Chemin d'exécution

pour j = 3
k = 5
l = 3

Path predicates : Triangle



Path predicates : Triangle

For $j = 3, k = 5$ and $l = 3$, path conditions :

$$\neg(j + k \leq l \vee k + l \leq j \vee l + j \leq k) \wedge j \neq k \wedge j = l \wedge l \neq k$$

Satisfiable conditions for all triplet (j, k, l) forming a triangle such that $j = l$ and $j \neq k$

Path predicates : Triangle

For the second path:

$$\neg(j + k \leq l \vee k + l \leq j \vee l + j \leq k)$$
$$\wedge\ j \neq k \wedge j \neq l \wedge l \neq k$$
$$\wedge\ 0 \neq 0 \wedge 0 \neq 1$$

insatisfiable conditions : There exists no triplet (j, k, l) such that those conditions are satisfiable during exécution

Selection criteria

$$\text{path} \rightarrow \text{path predicate} \rightarrow \text{input test case}$$

$\Rightarrow$ What paths are we interested in ? (there may be an infinite set of paths - program with a loop -).

$\Rightarrow$ Select a set of paths by considering path selection criteria:

- Statement/Node coverage
- Branch coverage
- Condition coverage
- All paths (or $k - \textit{paths}$ / $\textit{paths}$ with at most k loop iterations)

Procedure :

1. Select paths from a CFG based on a path selection criterion C
2. Extract path predicates from each path
3. Solve the path predicate expression to generate test input data.
   Undecidable problem !

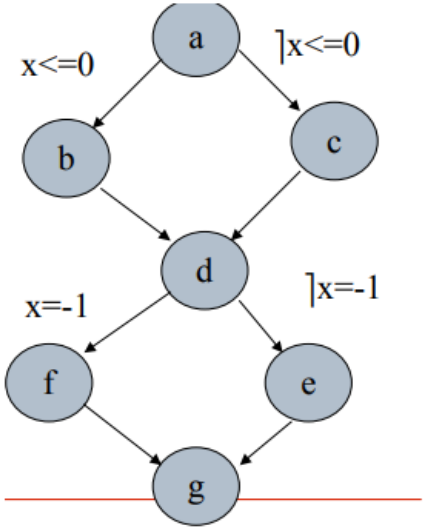
POLYTECH PARIS-SACLAY

Coverage criteria : all nodes/statements

- Elements to be tested : each node of CFG
- Coverage : each node (block of statements) has to be executed
- Limits : presence of dead code (undecidable)
- Also criterion *all-statements* = difference in granularity, not in concept (100% node coverage $\Leftrightarrow$ 100% statement coverage)
- Each statement (or node in the CFG) must be executed at least once
- Coverage rate = nb of executed statements / nb of statements
- Rationale: a fault in a statement can only be revealed by executing the faulty statement
- Coverage rate is not directly related to the size of test case set. Generally, one tries to minimize the size of test sets (in order to minimize the global cost).

POLYTECH PARIS-SACLAY

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```



Give a test set that satisfies the criterion all nodes ?

- The paths *abdeg* and *acdfg* allow the coverage of the criterion. (Remember : *abdfg* is unfeasible: forget it !)
- Path predicate for *abdeg*: $x \leq 0$
- Path predicate for *acdfg*: $x = 2$

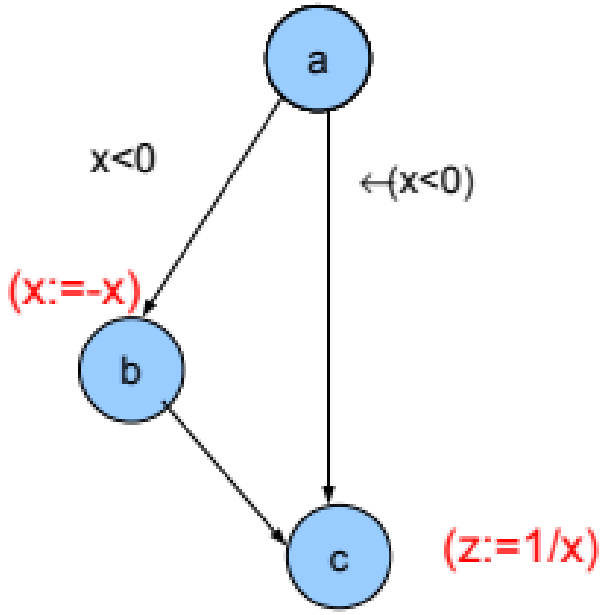
$\rightarrow$ An example of test set: 2

$\rightarrow$ test inputs: $x = -8, x = 2$

POLYTECH PARIS-SACLAY

Weakness of the all nodes/statements criterion

```
if x < 0
then x := -x;
end if;
z := 1/x;
```



- The test case $x = -3$ covers all statements
- It does not exercise the case when x is positive and when the *then* branch is not entered

$\Rightarrow$ *All statements/nodes* can miss some cases

POLYTECH PARIS-SACLAY

All-arcs criterion

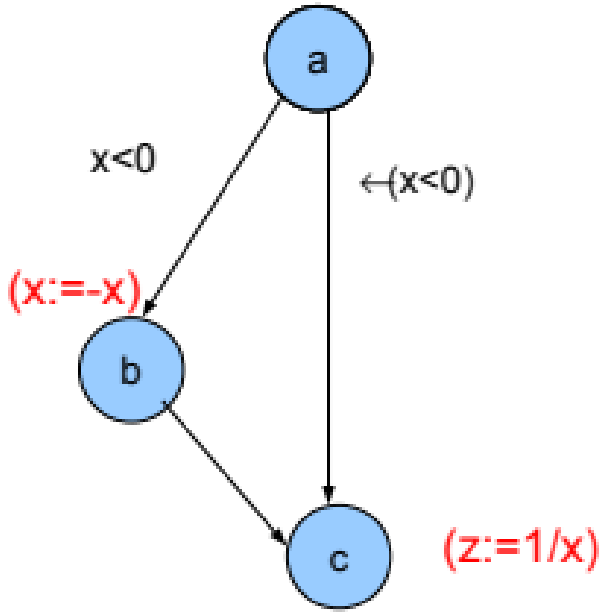
A test set T satisfies the criteria *CB all-arcs* iff for all arcs/edges $n \rightarrow n'$ of the control graph, there exists at least a test data in T that sensitizes a path containing the edge $n \rightarrow n'$

- Adequacy criterion: each edge in the CFG must be executed at least once
- Coverage rate : nb of executed arcs / nb of arcs
- Statements vs arcs
  - Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the arc adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
  - The converse is not true (consider the *if then* statement)

POLYTECH PARIS-SACLAY

Apply all-arcs criterion

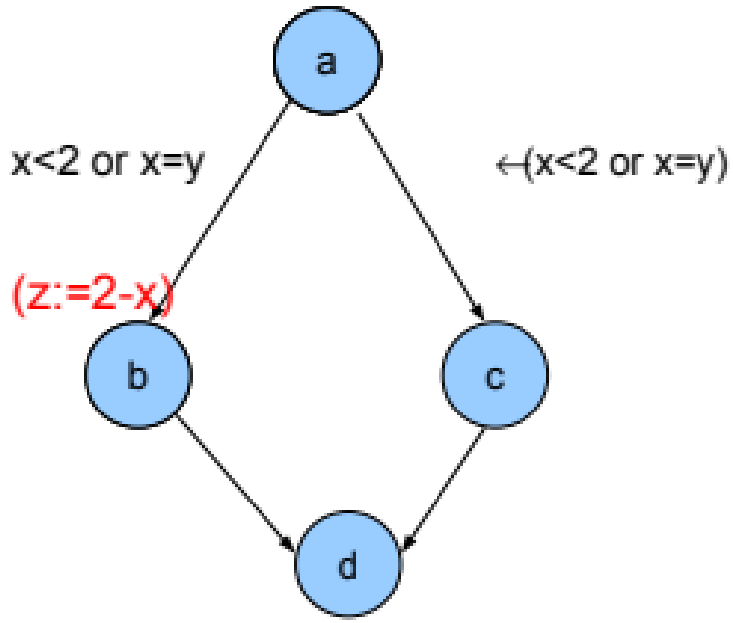
```
if x < 0
then x := -x;
end if;
z := 1/x;
```



- The test case $x = -3$ (path abc) covers the arcs $a - b$ and $b - c$
- We need another test case to cover $a - c$: path $ac-$ Its predicate is $P(x) = x \geq 0$
- Let us take an input x positive. $x = 0$ will reveal the pb (division by zero) but $x = 3$ will not

Apply all-arcs criterion

```
//input variables are x et y
  if ((x<2) and (x=y))
  then z := 2-x;
  else z := x-2;
  end if;
```



- The negation of $(x < 2)$ *and* $(x = y)$ is $(x \geq 2)$ *or* $(x \neq y)$
- $x = 1, y = 1$:condition is true ; $x = 3, y = 3$: condition is false

-> Both test cases satisfy the all-arcs criterion

-> But the decision contains 2 individual conditions (decision): it may be better to cover the 2 possible values of each decision

$\Rightarrow$ *all-decisions* criterion: each decision must be executed at least once

Condition (predicate) coverage criterion

Has each Boolean sub-expression evaluated both to true and false?

```
int foo (int x, int y)
{
  int z = 0;
  if ((x>0) && (y>0))
  {
   z = x;
  }
  return z;
}
```

- $DT_1 : x = 1, y = 1$ ...
  (c_1 and c_2 are both true)
- $DT_2 : x = 1, y = 0$ ...
  (c_1 is true and c_2 is false)
- $DT_3 : x = 0, y = 0$ ...
  (c_1 and c_2 are both false)

MC/DC .. Modified Condition/Decision Coverage

- MC/DC is defined in DO-178B/ED-12B, -"Software Considerations in Airborne Systems and Equipment Certification", 1992
- Each condition in a decision has been shown to independently affect that decision's outcome
- A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions

All-paths criterion

A test set T satisfies the criterion *all-paths* iff there exists at least a test data in T that sensitizes each path of the CFG.

- Adequacy criterion: each path must be executed at least once
- Coverage rate : nb of executed paths / nb of paths
- Path testing consider combinations of decisions in the program

!! But the number of paths in a program with loops is unbounded

!! So it is usually impossible to satisfy the criterion if a program has loops

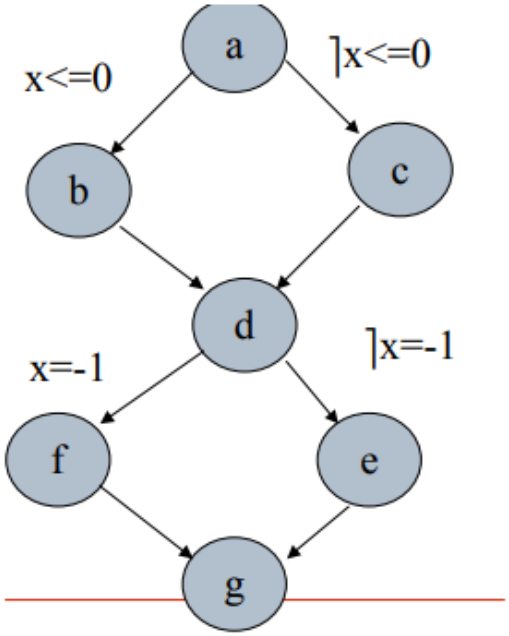
$\Rightarrow$ Limit the number of traversals of loops

$\Rightarrow$ $all - k - paths$ criterion: only paths with at most k crossing/passage in a given path (in loops) are considered

POLYTECH PARIS-SACLAY

Apply All-paths criterion

```
if x<=0          (a)
   then x:=-x    (b)
   else x:=1-x   (c)
endif;
if x=-1          (d)
   then x:=1     (f)
   else x:=x+1   (e)
endif;
write(x)         (g)
```



Give a test set that satisfies the criterion *all paths* ?

$T1 = \{x = -3; x = 2; x = 1\}$, $T2 = \{x = -5; x = 2; x = 8\}$ are two possible test sets satisfying the all paths criterion (since there are 4 paths, but only 3 executable)

- Path predicate for *abdeg*: $x \leq 0$
- Path predicate for *acdfg*: $x = 2$
- Path predicate for *acdeg*: $x > 0 \wedge x \neq 2$

Comparison of criteria

- $all - paths$ is finer than $all - k - paths$
- $all - k - paths$ is finer than $all - arcs$
- $all - arcs$ is finer than $all - nodes$

Tools

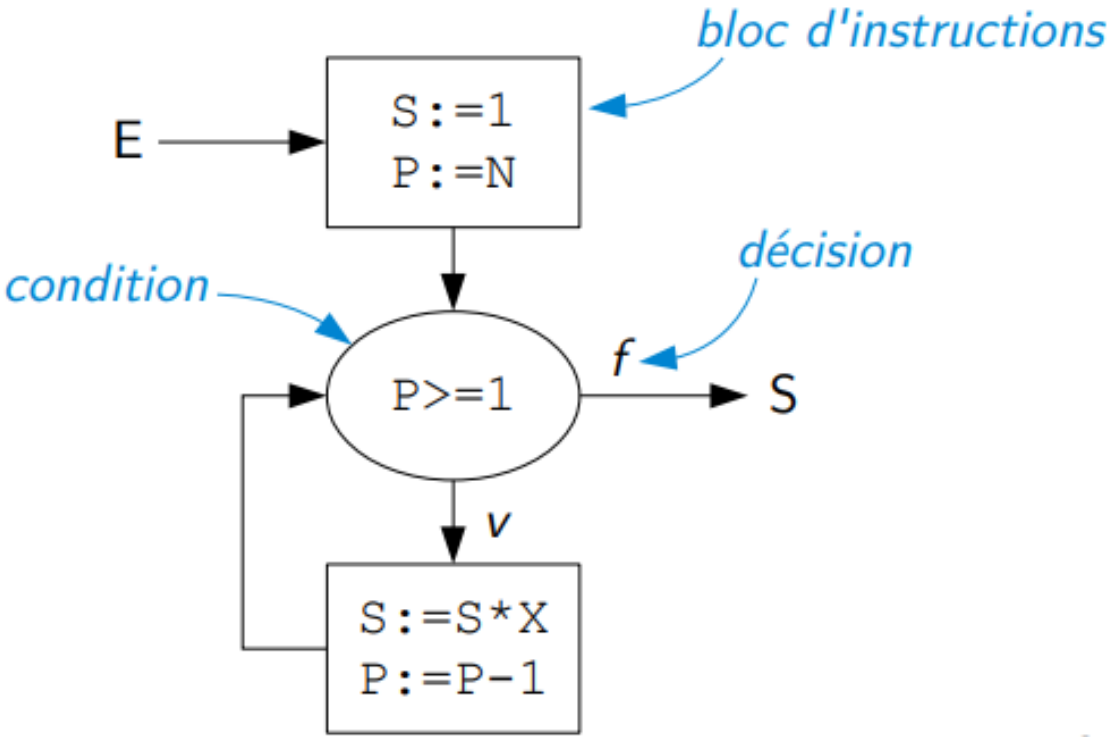
- Evaluation of coverage rate: EclEmma (plugin Eclipse)
- Generation of test cases by using symbolic execution techniques and constraint solving
- Examples : Pathcrawler, euclide, Pex ...
- PathCrawler: tool for the automatic generation of test cases (for C programs) satisfying the rigorous *all – paths* criterion, with a user-defined limit on the number of loop iterations in the covered paths.
  on-line version: `http://pathcrawler-online.com/`

POLYTECH PARIS-SACLAY

2. Data-flow based Coverage

Test structurel: Graphe de flot de contrôle

Graphe orienté faisant apparaître la structure du code en termes de chaînes d'instructions, de branchements conditionnels et de boucles



Graphe de flot de données / data flow graph (DFG)

- Annotation du graphe de flot de contrôle par les définitions et les utilisations de variables
- Base pour l'analyse des dépendances entre les définitions et utilisations d'une même variable : ordre des annotations important
  - Une variable est définie dans une instruction si la valeur de la variable est modifiée (affectation, déclaration, read ...)
  - Une variable est dite utilisée ou référencée si la valeur de la variable est utilisée
  - Si la variable utilisée est utilisée dans le prédicat d'une instruction *if – then – else* ou *while*, il s'agit d'une p-utilisation (p comme prédicat), sinon il s'agit d'une c-utilisation (c comme calcul)
  - A toute définition doit correspondre au moins une utilisation. Sinon anomalie !

```
while (i<n) do    p-utilisations de i et n
  s := s + i;     c-utilisations de s et i et déf. de i
  i := i + 1      c-utilisation de i et déf. de i
end;
write(s)          c-utilisation de s
```

- instruction $i2$ utilisatrice (d'une variable x) par rapport à une autre instruction $i1$: la variable x définie en $i1$ peut être directement utilisée en $i2$
  - directement : aucune autre définition de la variable x entre $i1$ et $i2$
  - on dit aussi : $d_x(i1)$ atteint $u_x(i2)$
- chemin d'utilisation ou chemin *du* : chemin qui relie l'instruction de définition d'une variable à une instruction utilisatrice

```
(1) x := 42
(2) a := x + 2
(3) b := a * a
(4) a:=a+1
(5) z:=x+a
```

$[1, 2, 3, 4, 5]$: chemin d'utilisation pour la variable x mais pas pour la variable a

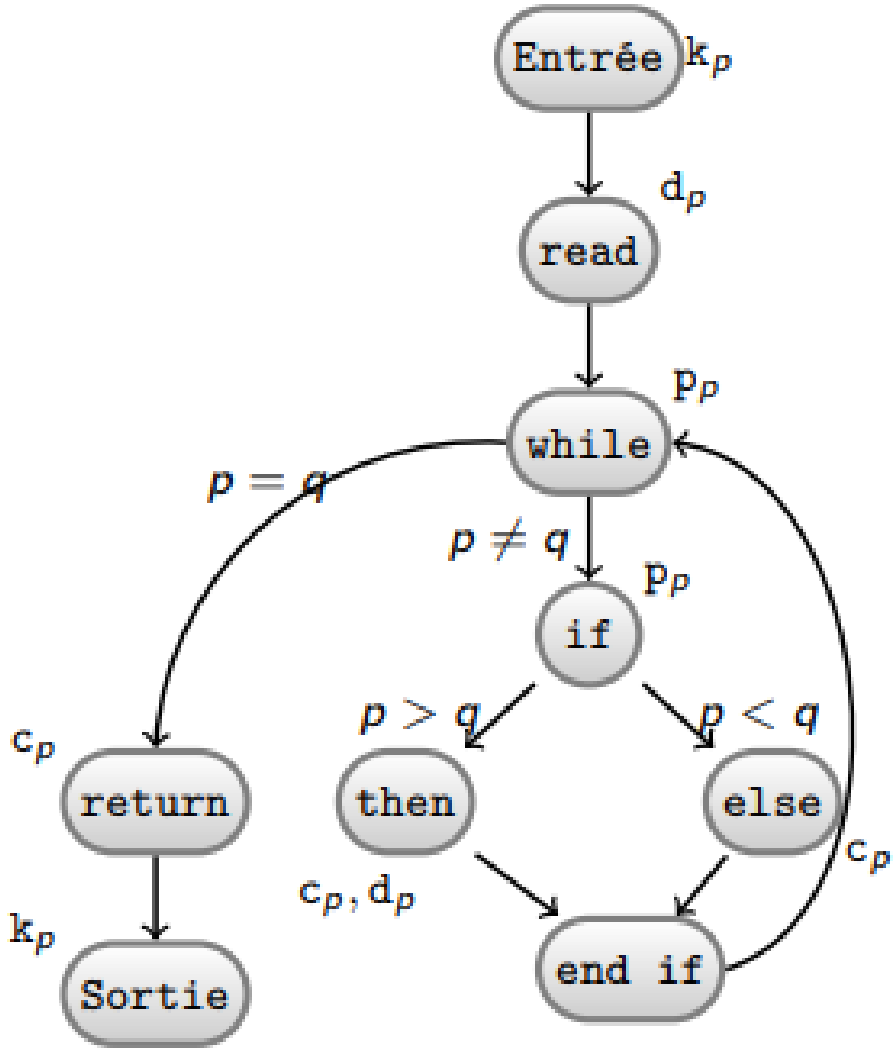
- Graphe de flot de données = on ajoute au graphe de flot de contrôle (CFG) les informations liant définitions et utilisations des variables
  Attention! : un nœud pour chaque instruction, plus un nœud final de sortie
  - d_x : instruction où la variable x est définie dans le nœud
  - u_x (p_x/c_x) : instruction où x est utilisée dans le nœud
  - k_x : la variable x est désallouée dans le nœud
- Analyse des dépendances entre les définitions et utilisations d'une variable
  - Statiquement : Une utilisation peut correspondre à plusieurs définitions
  - Dynamiquement : Chaque utilisation correspond à une seule définition
- Flot de données pour une variable x :
  - Pour une exécution du programme : suite des définitions et utilisations (mot sur l'alphabet $\{d_x, u_x\}$)
  - Pour toutes les exécutions du programme : ensemble des suites de définitions et utilisations (langage sur l'alphabet $\{d_x, u_x\}$)

POLYTECH PARIS-SACLAY

Calcul du pgcd de deux entiers naturels

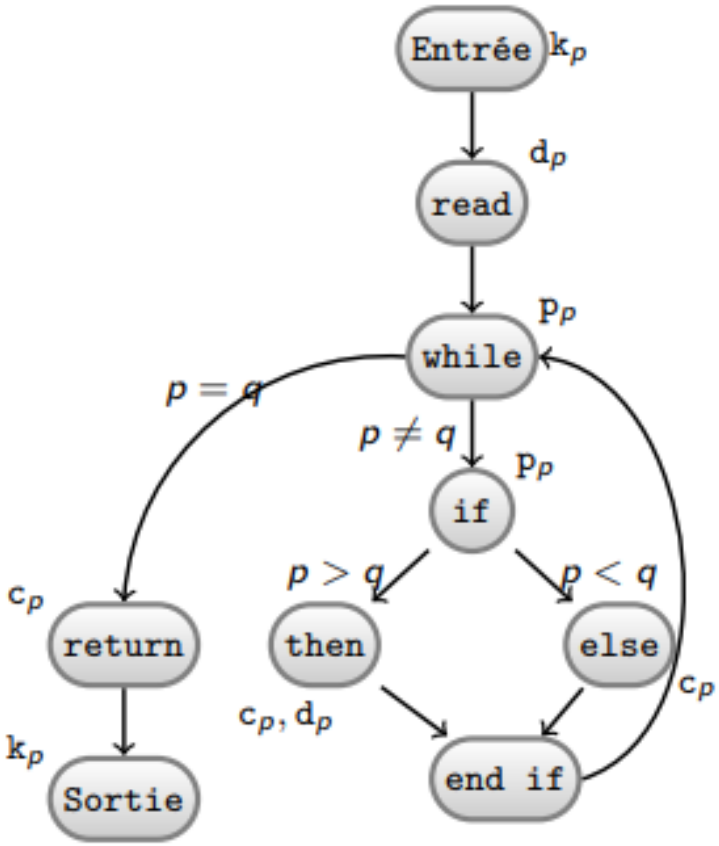
PGCD(p, q) :

1: read(p, q)
2: **while** $p \neq q$ **do**
3: **if** $p > q$ **then**
4: $p = p - q$
5: **else**
6: $q = q - p$
7: **end if**
8: **end while**
9: return p



Utilisation statique

- On peut également utiliser ce graphe de manière statique pour déterminer des anomalies
- On fixe une variable - On calcule le langage engendré par les exécutions possibles vis-à-vis de la variable choisie



Pour la variable p : expression régulière

$$k_p d_p (p_p p_p (c_p d_p | c_p)) * p_p c_p k_p)$$

POLYTECH PARIS-SACLAY

Utilisation statique

Détection des erreurs potentielles :

- Par analyse du langage
- Anomalies du flot de données : Présence d'une anomalie dans un chemin si le flot de données associé pour x est de la forme :
  - $u_x \ldots$ (commence par une utilisation) : variable non initialisée
  - $\ldots d_x$ (finit par une définition) : mise à jour jamais utilisée
  - $\ldots d_x d_x \ldots$ (redéfinition sans utilisation) : mise à jour non utilisée

  anomalie $\neq$ erreur
- Mais présence d'anomalies peut rendre certains critères de couverture impossibles à satisfaire (comment tester une définition si elle n'est jamais utilisée ?)

Dans la suite :

Utilisation du graphe de flot de données pour déterminer des objectifs de couverture de test (dynamique)

Suivre les chemins qui vont de la définition d'une variable à ses utilisations sans repasser par une nouvelle définition de cette variable

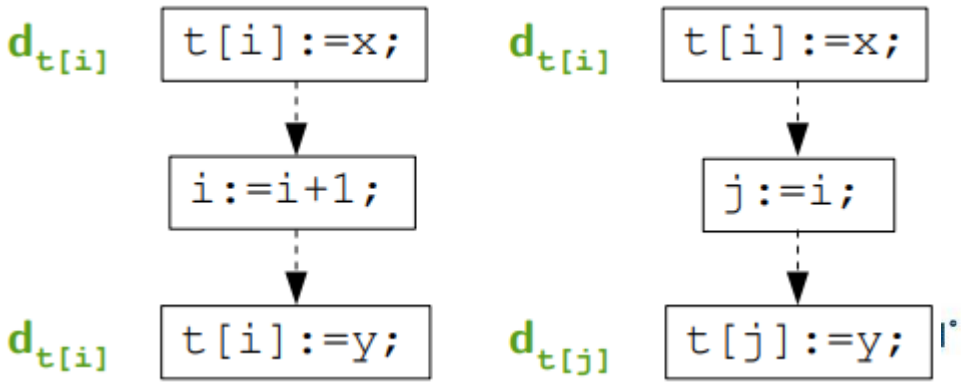
Couverture du flot de données

- Notations (bloc = bloc d'instructions ou condition) :
  - $d_B(x)$: le bloc B contient une définition de x
  - $u_B(x)$: le bloc B contient une utilisation de x
- Une définition $d_B(x)$ atteint une utilisation $u'_B(x)$ si et seulement si x n'est pas redéfini entre les blocs B et B'

Redéfinition d'une variable

Une définition de x est tuée sur un chemin C s'il existe dans C une redéfinition non ambiguë de x. Non ambiguë ?

- Appel de procédure $p(x, y)$: x peut-elle être modifiée par p ?
- Définition d'un élément de tableau $t[i] := exp$: tue-t-elle $t[j]$?
- Définition de la valeur référencée par un pointeur $*a := exp$: tue-t-elle la valeur référencée par un autre pointeur ?



Critère *toutes-les-définitions*

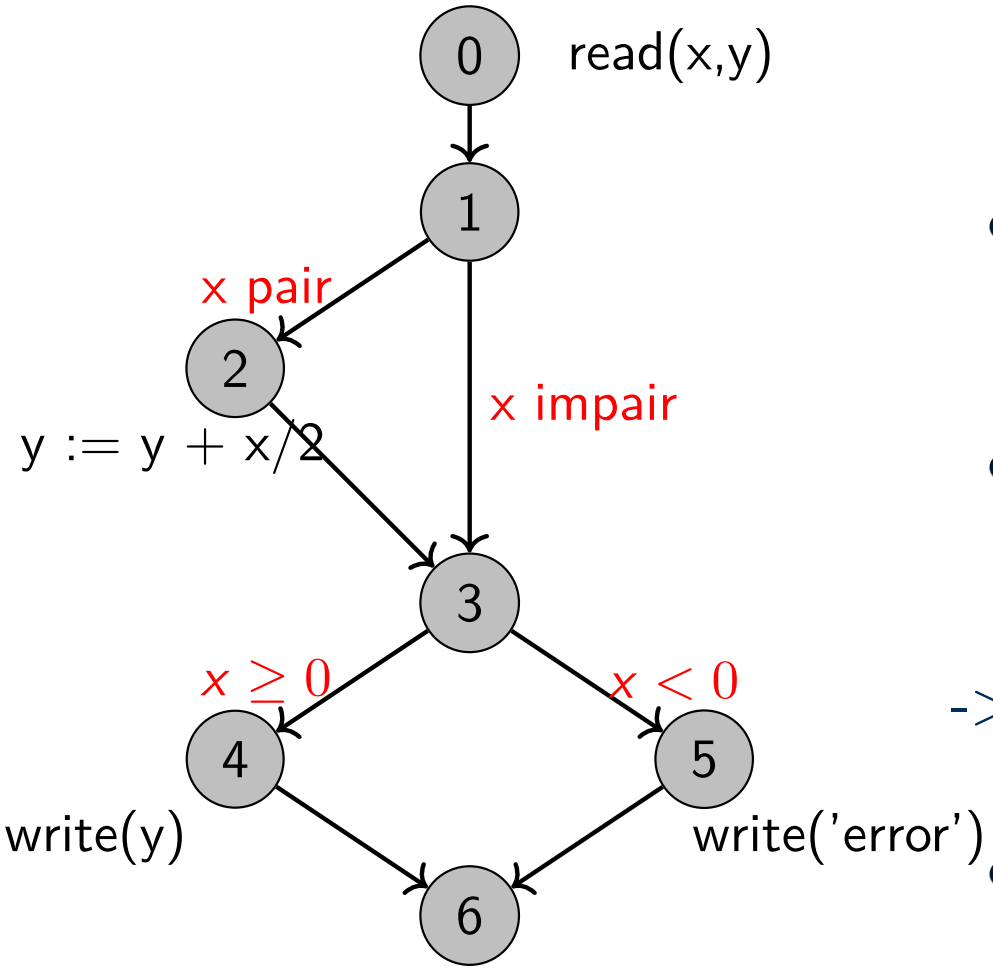
Satisfait par un ensemble de chemins T si :

- pour toute variable x,
- pour toute définition $d_B(x)$ en B,
- il existe au moins une utilisation $u_{B'}(x)$ en B' atteinte par $d_B(x)$ telle qu'il existe un chemin de T qui contient BCB' où C est un chemin sans redéfinition de x

Intuitivement : Toutes les définitions sont utilisées au moins une fois

POLYTECH PARIS-SACLAY

Critère *toutes-les-définitions* : exemple 1



- Pour la variable y : définitions en 0 et en 2. Utilisations en 2 et en 4
- Par exemple, avec le chemin $[0, 1, 2, 3, 4, 6]$ on couvre le critère pour y:

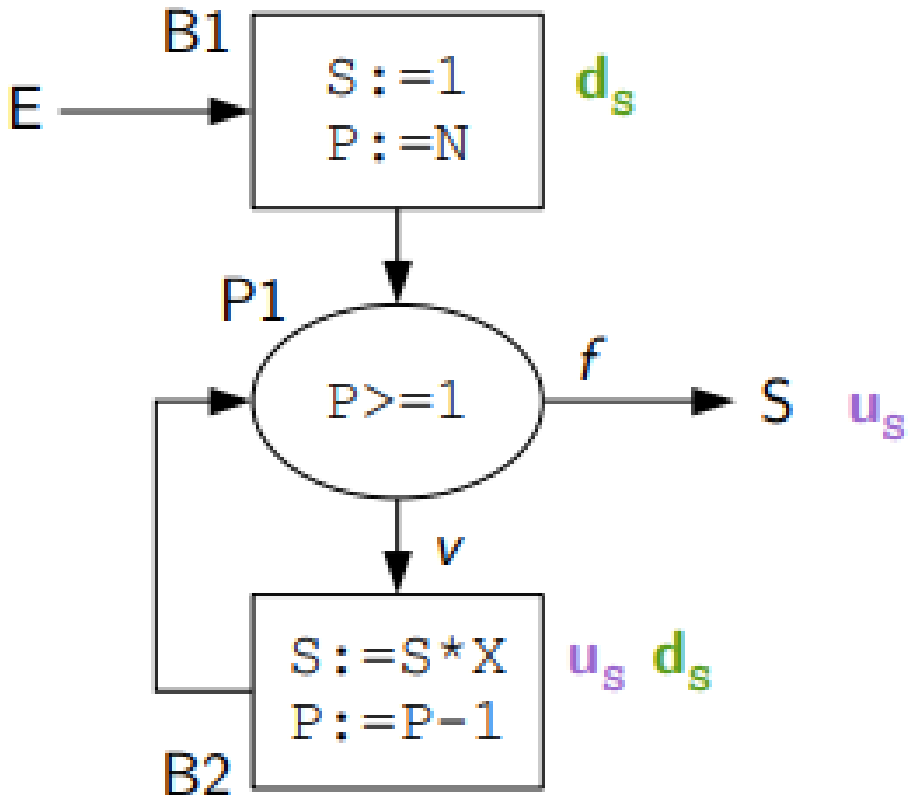
-> Définition en 0 utilisée en 2 - Définition en 2 utilisée en 4

- Pour la variable x : ?

Limite du critère : certains couples définition-utilisation pour y non couverts (utilisation en 4 de la définition en 0)

Critère *toutes-les-définitions* : exemple 2

- Pour la variable S : définitions en $B1$ et en $B2$
- Pour chaque définition, couvrir une utilisation
- Par exemple, avec le chemin $[E, \underline{B1}, P1, \underline{B2}, P1, \underline{S}]$:
  - Définition en $B1$ utilisée en $B2$
  - Définition en $B2$ utilisée en S



Même défaut que précédemment $(d_S(B1), u_S(B2))$ couple *du* non couvert (utilisation en S de la définition en B1 et utilisation en B2 de la définition en B2)

Critère *Toutes-les-utilisations*

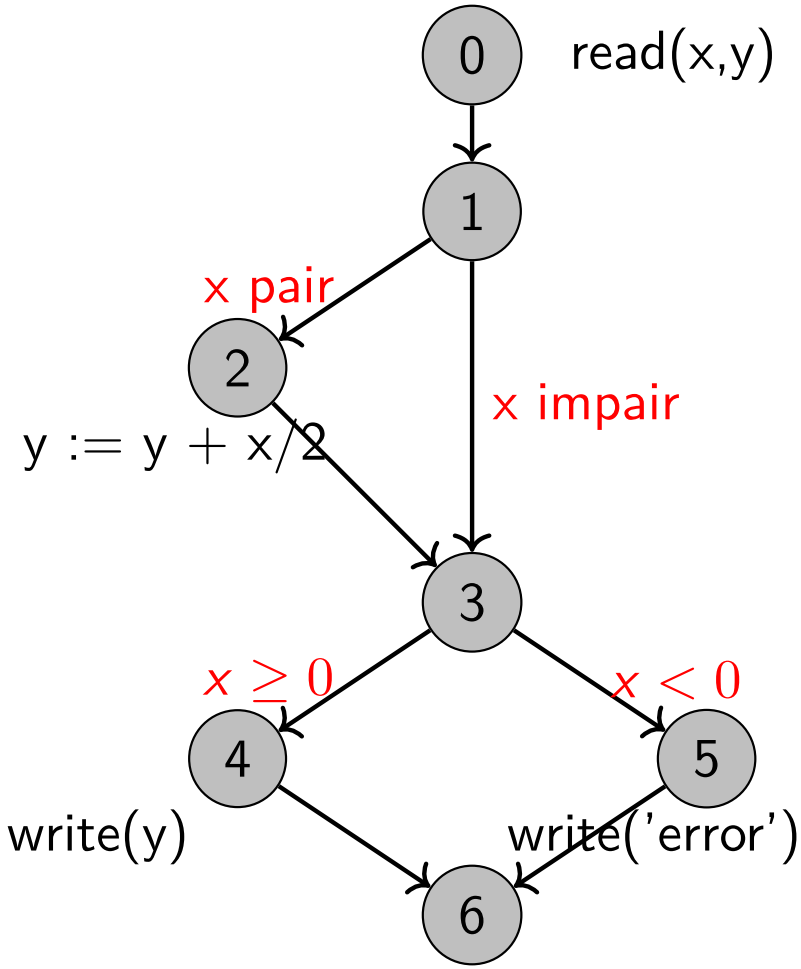
- Vérifier que la valeur donnée à une variable est utilisée correctement
- Il s'agit de couvrir toutes les utilisations, i.e. couvrir au moins un chemin pour chaque paire *du*
- Satisfait par un ensemble de chemins T si :
  - pour toute variable x,
  - pour toute définition $d_B(x)$ en B,
  - pour toute utilisation $u_{B'}(x)$ en B' atteinte par $d_B(x)$ en B, il existe un chemin de T qui contient BCB' où C est un chemin sans redéfinition de x

Intuitivement : Toutes les utilisations accessibles par chaque définition

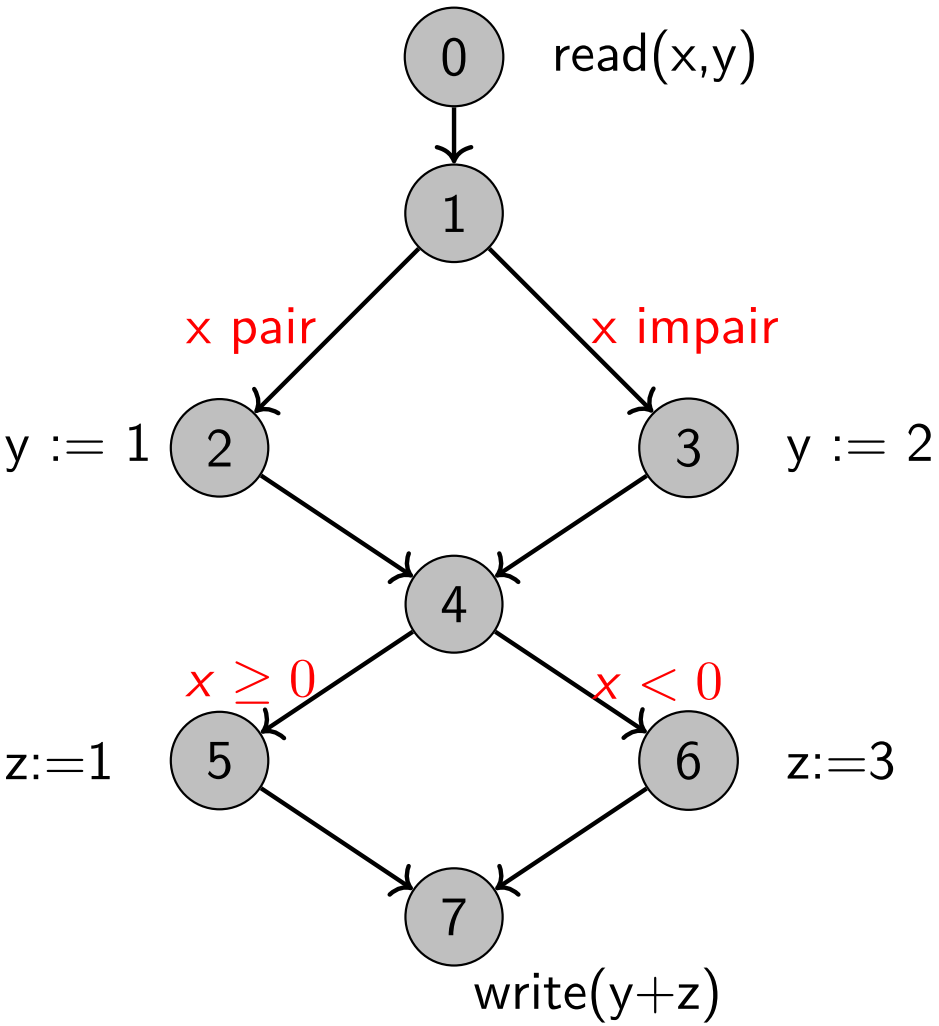
Critère *Toutes-les-utilisations* : Exemple 1 (1)

- Pour la variable y : Définition en 0 et 2, utilisation en 2 et 4
  - Définition en 0, utilisation en 2 : $[0, 1, 2, 3, 4]$
  - Définition en 0, utilisation en 4 (sans passer par une redéfinition de y) : $[0, 1, 3, 4]$
  - Définition en 2, utilisation en 4 : $[0, 1, 2, 3, 4]$

  -> 2 chemins $\rightarrow$ Critère couvert à 100% pour y
- Pour la variable x : Définition en 0, utilisation en 1 et en 3 :
  - $[0, 1, 3, 4]$ couvre les deux paires *du* pour x



Critère *Toutes-les-utilisations* : Exemple 1 (2)



- Les chemins $[0, 1, 2, 4, 5, 7]$ et $[0, 1, 3, 4, 6, 7]$ satisfont le critère *toutes-les-utilisations*
- Mais on ne couvre pas tous les chemins d'utilisation :

-> La définition de y en 2 peut être utilisée par l'instruction au noeud 7 en passant par 5 (couvert par le chemin $[0, 1, 2, 4, 5, 7]$) ou en passant pr 6 (non couvert)

-> Idem pour la définition de y au noeud 3

POLYTECH PARIS-SACLAY

Critère *tous-les-du-chemins*

chemin simple

portion de chemin allant d'une définition à une (p ou c) utilisation sans repasser par une nouvelle définition de la variable. Chemin sans circuit ou contenant un circuit élémentaire. Autrement dit : au plus un noeud apparaît deux fois dans le chemin

Le critère est satisfait par un ensemble de chemins T si :

- pour toute variable x,
- pour toute définition $d_x(B)$,
- pour toute utilisation $u_x(B')$ atteinte par $d_x(B)$,
- pour tout sous-chemin BCB' où C est un chemin sans redéfinition de x et BCB' est simple,
- il existe un chemin de T qui contient BCB'

Intuitivement : Tous les chemins (simples) pour chaque couple définition-utilisation (DU)

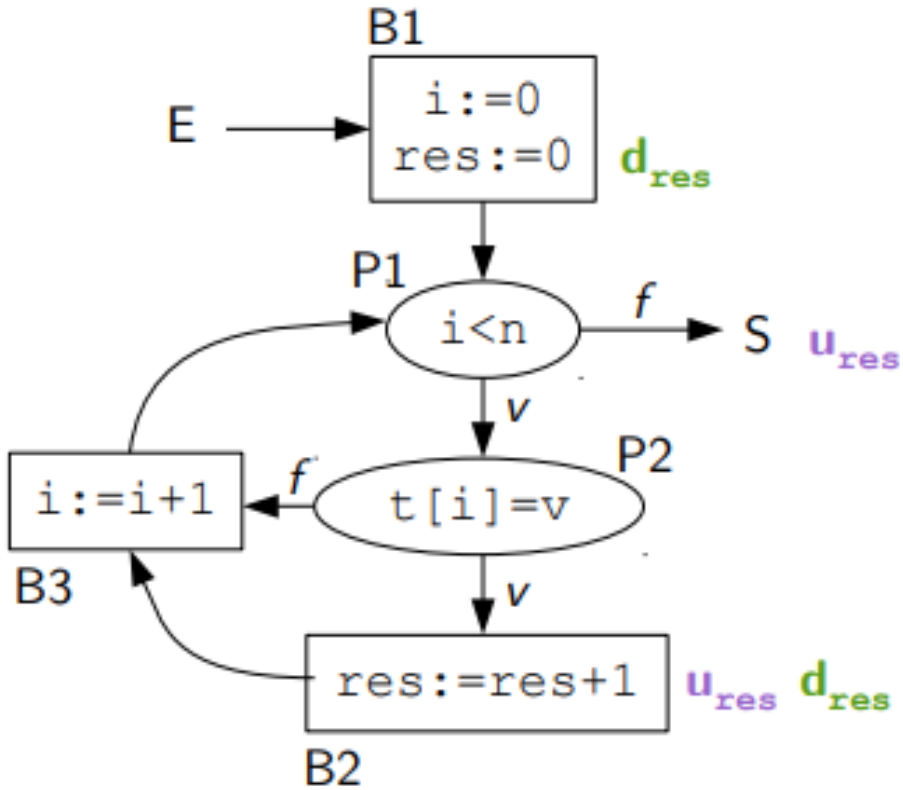
Critère *tous-les-du-chemins*

Pour la variable *res* :

- Définitions en $B1$ et en $B2$
- Utilisations en $B2$ et en S

Couverture du couple $(B1, S)$:

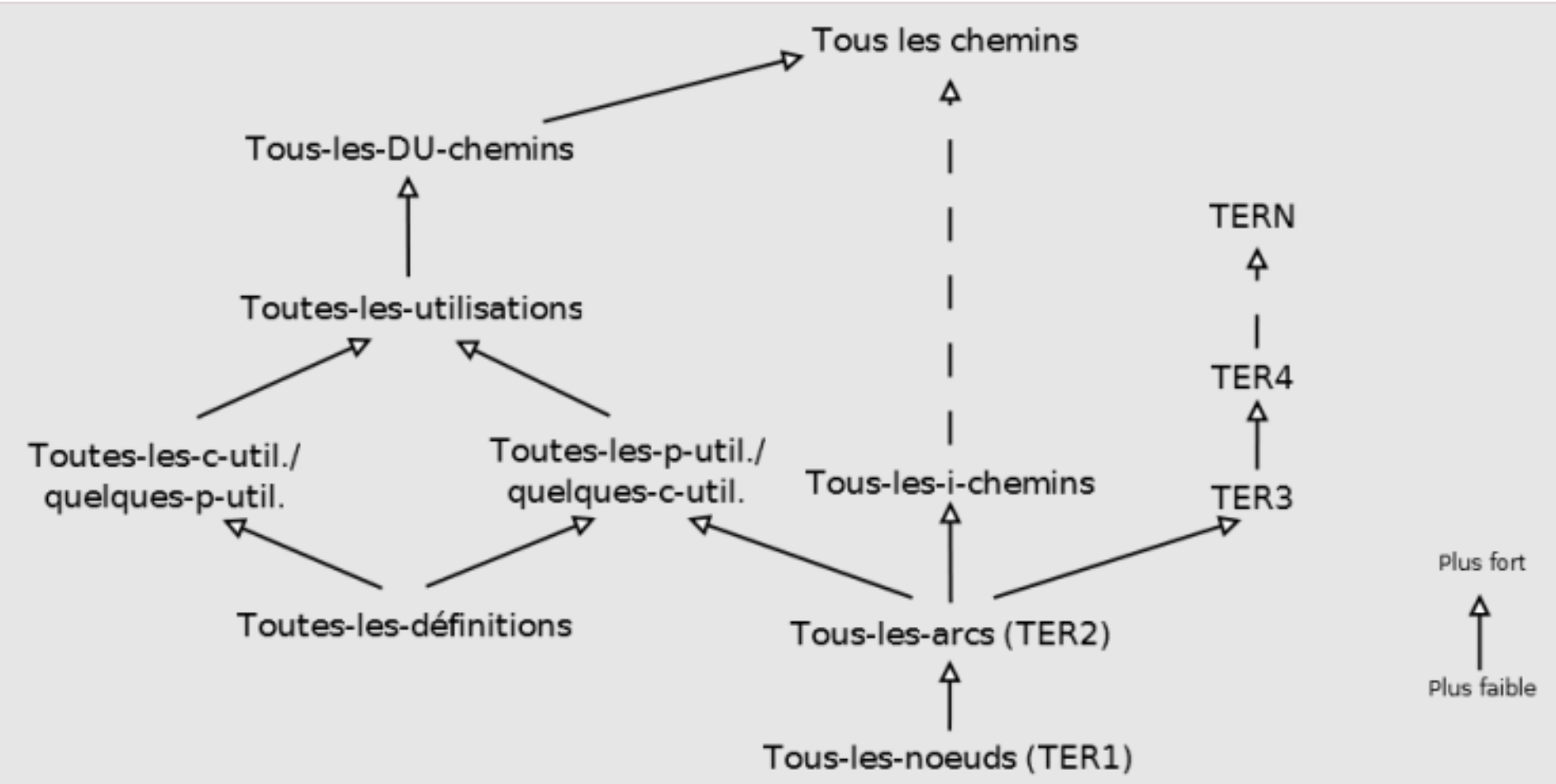
- Chemin sans circuit : $EB1P1S$
- Chemin contenant un circuit élémentaire sans redéfinition de *res*: $EB1P1P2B3P1S$



Couverture de tous les chemins simples menant de la définition à son utilisation

POLYTECH PARIS-SACLAY

Comparaison des critères



Certains critères sont incomparables

Quel est le meilleur critère à adopter ?

Dépend de :

- Niveau de fiabilité à atteindre (point de vue stratégique)
- Types d'erreurs que l'on considère fréquentes et graves
  - tous-les-arcs, toutes-les-décisions : nouvelles combinaisons de chemins sont exercées
  - Erreur logique : on utilise dans une condition la variable x au lieu de la variable y
  - Si p = prob. de présence de ce défaut, réduite à p^2 après avoir satisfait tous-les-arcs, réduite à p^8 après avoir satisfait tous-les-chemins-indépendants (critère non présenté dans ce cours)
- facteur technique : efficacité d'un critère. Plus il est haut dans la hiérarchie, plus il est fort

POLYTECH PARIS-SACLAY

- Ces critères structurels en général pour la couverture de code pour le test unitaire.
- Adaptation possible à d'autres phases de test, par exemple couvrir les arcs du graphe d'appels, couvrir toutes les utilisations de lock, unlock
- Quelques repères :
  code critique, test unitaire : 100% du MC/DC
  code non critique : facile : 50% toutes-les-instructions, bien : $>$ 85% toutes-les-décisions
- Critères structurels sur autre chose que du code :
  couverture des spécifications : couverture du use-case, couverture du modèle (automate étendu)

POLYTECH PARIS-SACLAY

Conclusion

Utilisations des méthodes de test structurel

- Seules, pour générer un jeu de tests pour un programme selon un ou plusieurs critères de couverture
  Outils de génération automatique de tests : Pathcrawler (langage C), Pex (langage C#, Microsoft)...
- Pour évaluer la qualité (en termes de couverture) d'un jeu de tests existant, et le compléter pour satisfaire complètement les critères visés
  Outils de mesure de couverture : CodeCover, Emma, Clover (Java), gcov (C)...

En pratique : test fonctionnel puis couverture de toutes les décisions et de toutes les utilisations

POLYTECH PARIS-SACLAY

Exemple de la norme DO-178C en avionique

5 niveaux de criticité des applications (DAL) : de A à E.
La norme fixe les conditions de sécurité applicables aux logiciels critiques avioniques. Elle fixe des contraintes de développement.

- Niveau A : Un défaut du système ou sous-système étudié peut provoquer un problème catastrophique - Sécurité du vol ou atterrissage compromis - Crash de l'avion
  $\longrightarrow$ requis : couverture du critère MC/DC
- Niveau B : Un défaut du système ou sous-système étudié peut provoquer un problème majeur entrainant des dégâts sérieux voire la mort de quelques occupants
  $\longrightarrow$ couverture du critère toutes-les-décisions
- Niveau C : Un défaut du système ou sous-système étudié peut provoquer un problème sérieux entrainant un dysfonctionnement des équipements vitaux de l'appareil
  $\longrightarrow$ couverture du critère toutes-les-instructions

POLYTECH PARIS-SACLAY