2008-2009

Cycle Ingénieur – 2ème année

Département Informatique

Verification and Validation

Part IV : System Test III

Burkhart Wolff

Département Informatique

Université Paris-Sud / Orsay

Difficulties with Static Unit Tests so far

- The generation of test-data is left open;
  however, this is the CORE of the problem
- Setup of drivers and stubs necessary

  (but that can be automated ...)
- Random-testdata generators are usually

  very ineffective (why ???) ☹

  writing „simple" versions compliant to

  the post-conditions results in additional programming

  labour ... ☹

Difficulties with Static Unit Tests so far

- When do we have tested enough ?
  When is our test "adequate" ?

  We have to decide on adequacy criteria in advance.
  This can be:
  - criteria on the coverage of the spec of the program
  - criteria on statistical models and an
    error model

- Some empirical observations:
  - No relation between detection order and detection difficulty
  - No relation between detection difficulty and correction
  - The more errors you found, the more you find more…
  - The quality of a test set is independent of its size.

Idea:

- Lets exploit the structure of the program !!!

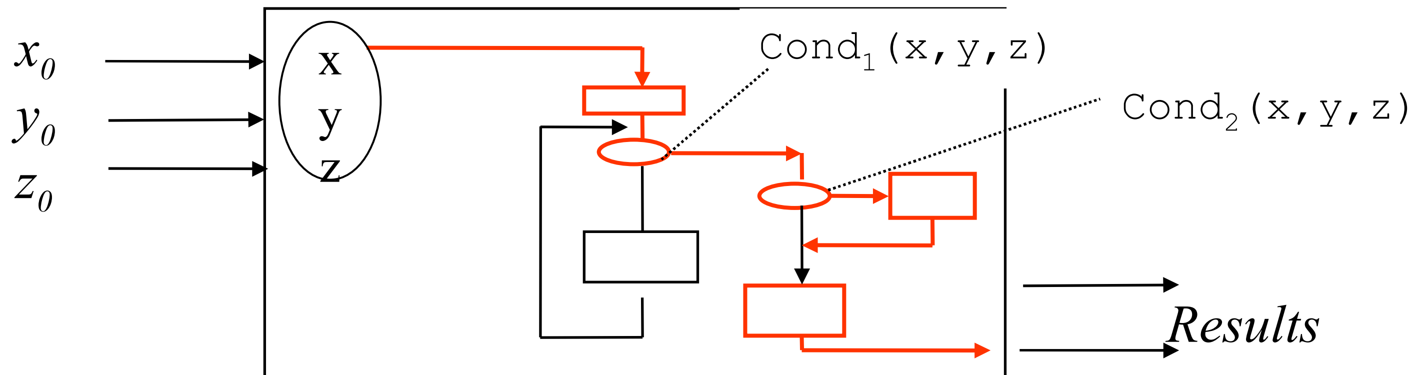
  (and not, as before in specification based tests („black box"-tests), depend entirely on the spec).

  Assumption: Programmers make most likely errors in branching points of a program (Condition, While-Loop, ...).

  Lets develop a test method that tries to check against this !

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 <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
- Erase if_then_elses by branching

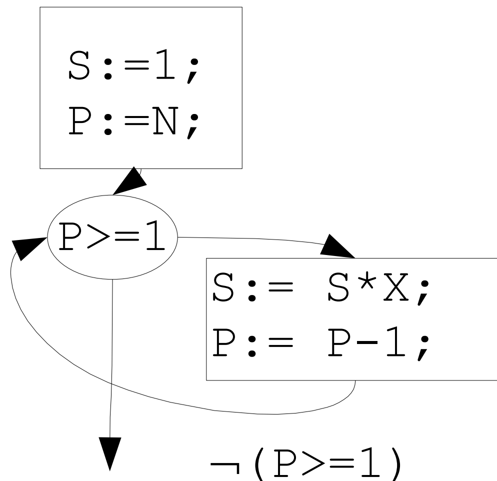
Computing Control Flow Graphs

- 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

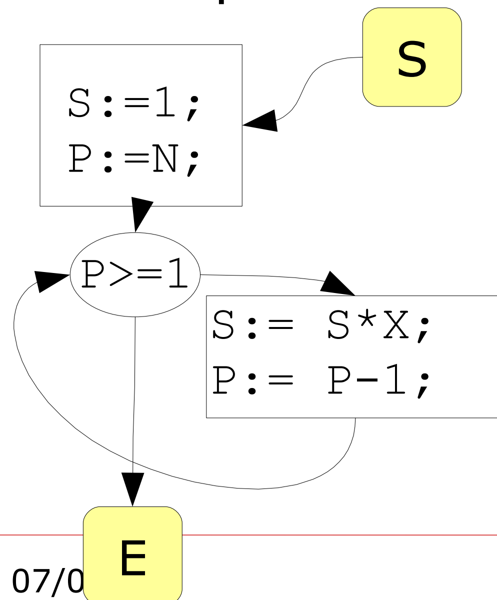
Example:



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

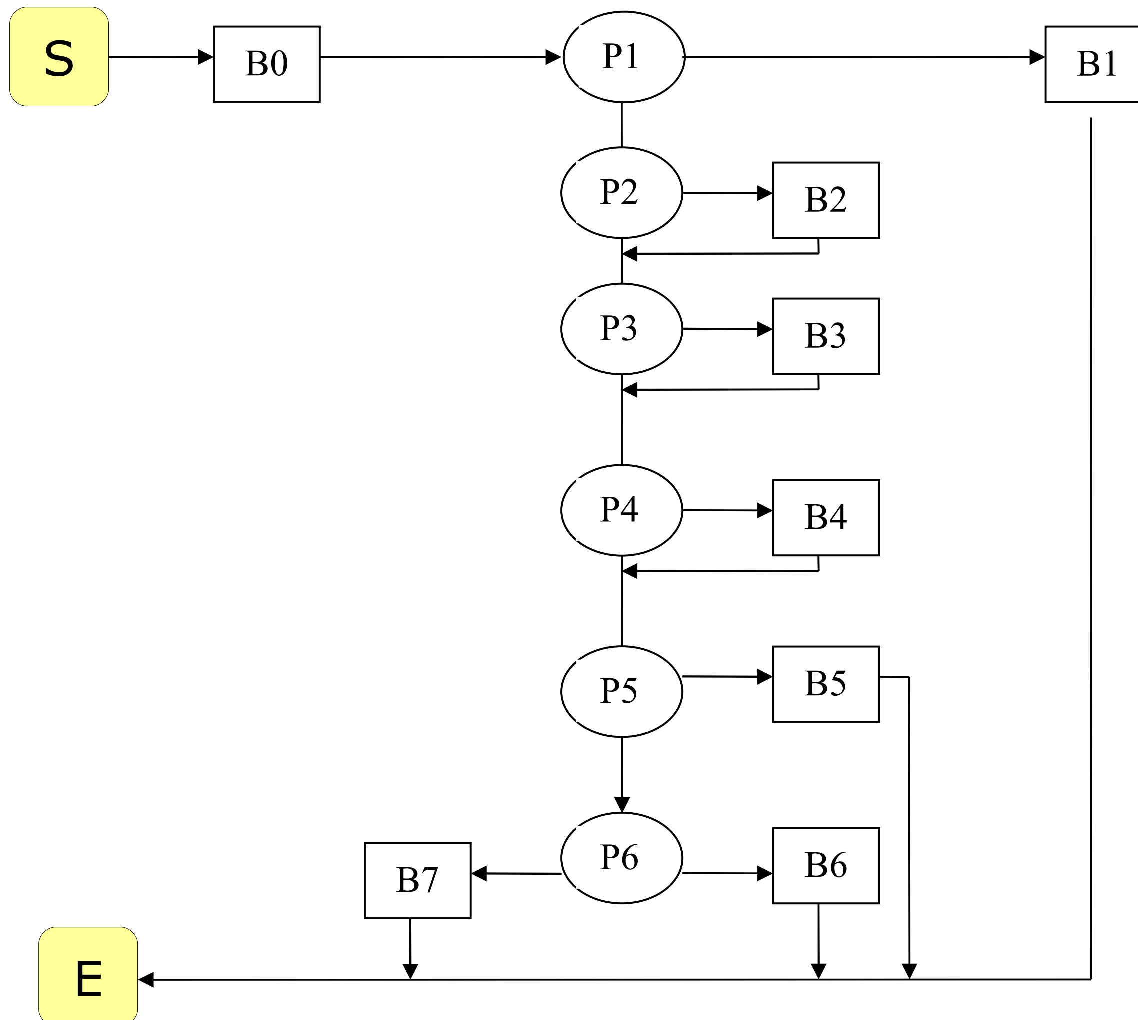
A Control-Flow-Graph (CFG) is usually a by-product of a compiler ...

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



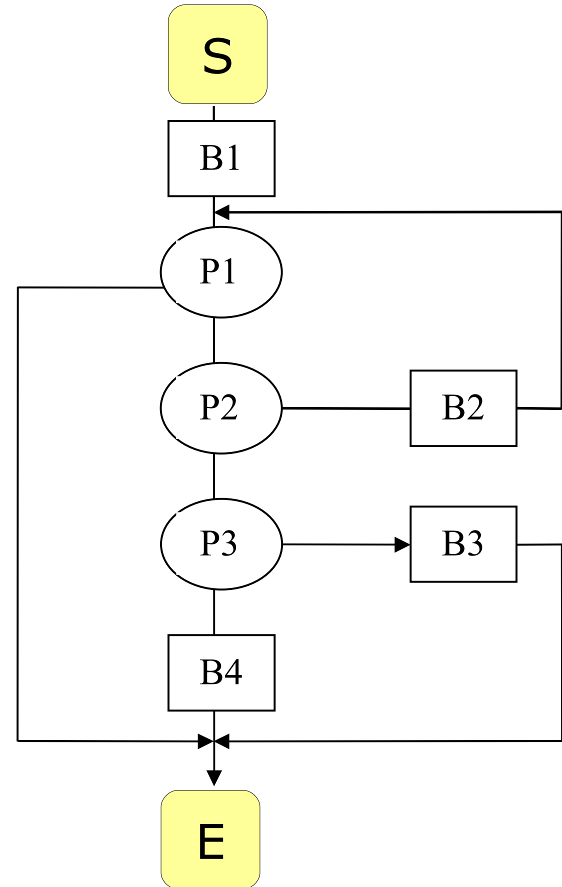
A procedure with loop and return

```
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 <u>predicate</u> (over parameters and state): a condition over the **initial values initiales** of parameters (and global variables) to achieve **exactly** this execution path
  - <u>faisable paths</u> = 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 $\Phi := \Phi \wedge P(\underline{x},\ldots,\underline{z})$,

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

(The $\underline{x},\ldots,\underline{z}$ are the symbolic values for x,...,z).

Symbolic execution

Symbolic Pre-State	
x	x_0
y	y_0+ 3*x_0
z	z_0
i	

B

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

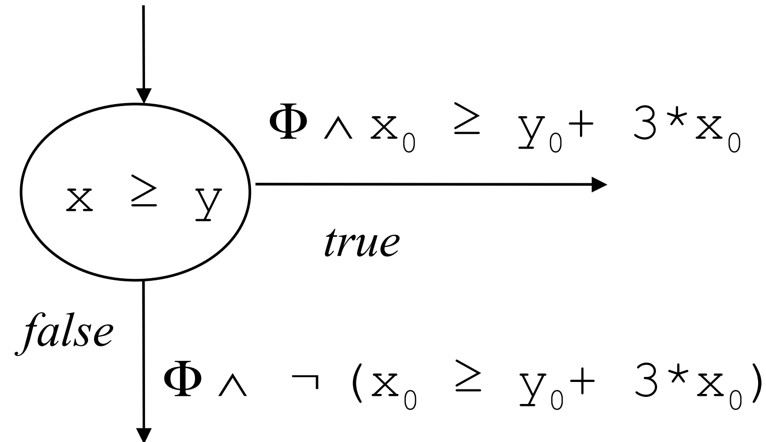
Symbolic Post-State	
x	x_0
y	y_0+ 3*x_0
z	z_0+ y_0+4*x_0+1
i	y_0+ 4*x_0+1

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

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

Symbolic Execution

x	x_0
y	$y_0 + 3*x_0$



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

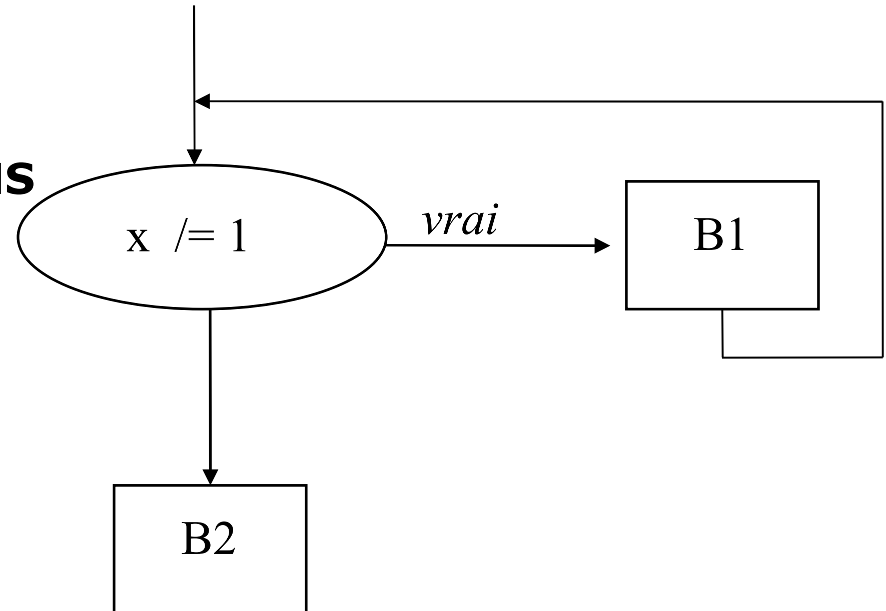
- In general, it is undecidable of a path is feasible ...
- In general, it is undecidable if a program will terminate ...
- In general, equivalence on two programs is undecidable …
- In general, a first-order formula over arithmetic is undecidable ...
- **...**

  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) ...

A Challange-Example (Collatz-Function):

... ALTHOUGH FOR SOME SPECTACULARLY SIMPLE PROGRAMS THESE SYSTEMS FAIL:

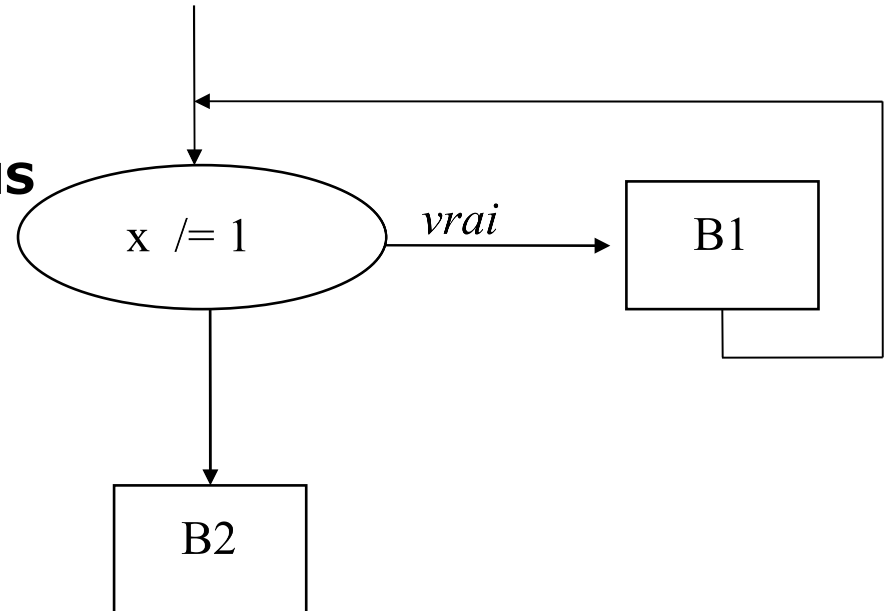



```
while x <> 1 loop
      if pair(x) then x := x / 2;
      else x := 3 * x + 1;
      end if;
 end loop;
```

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

A Challange-Example (Collatz-Function):

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



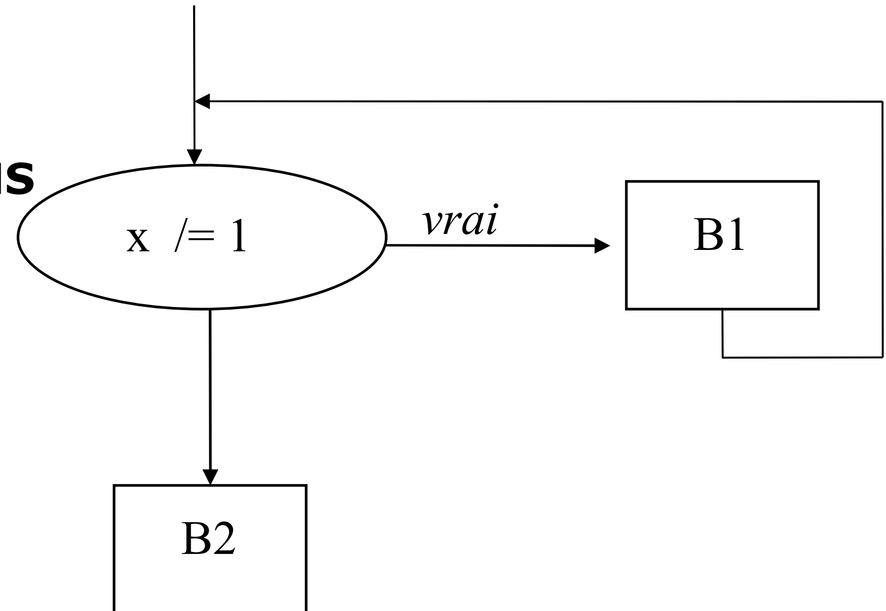

```
while x <> 1 loop
      if pair(x) then x := x / 2;
      else x := 3 * x + 1;
      end if;
 end loop;
```

- 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 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
- 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: $\text{AllPaths}_k(M,T)$.

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

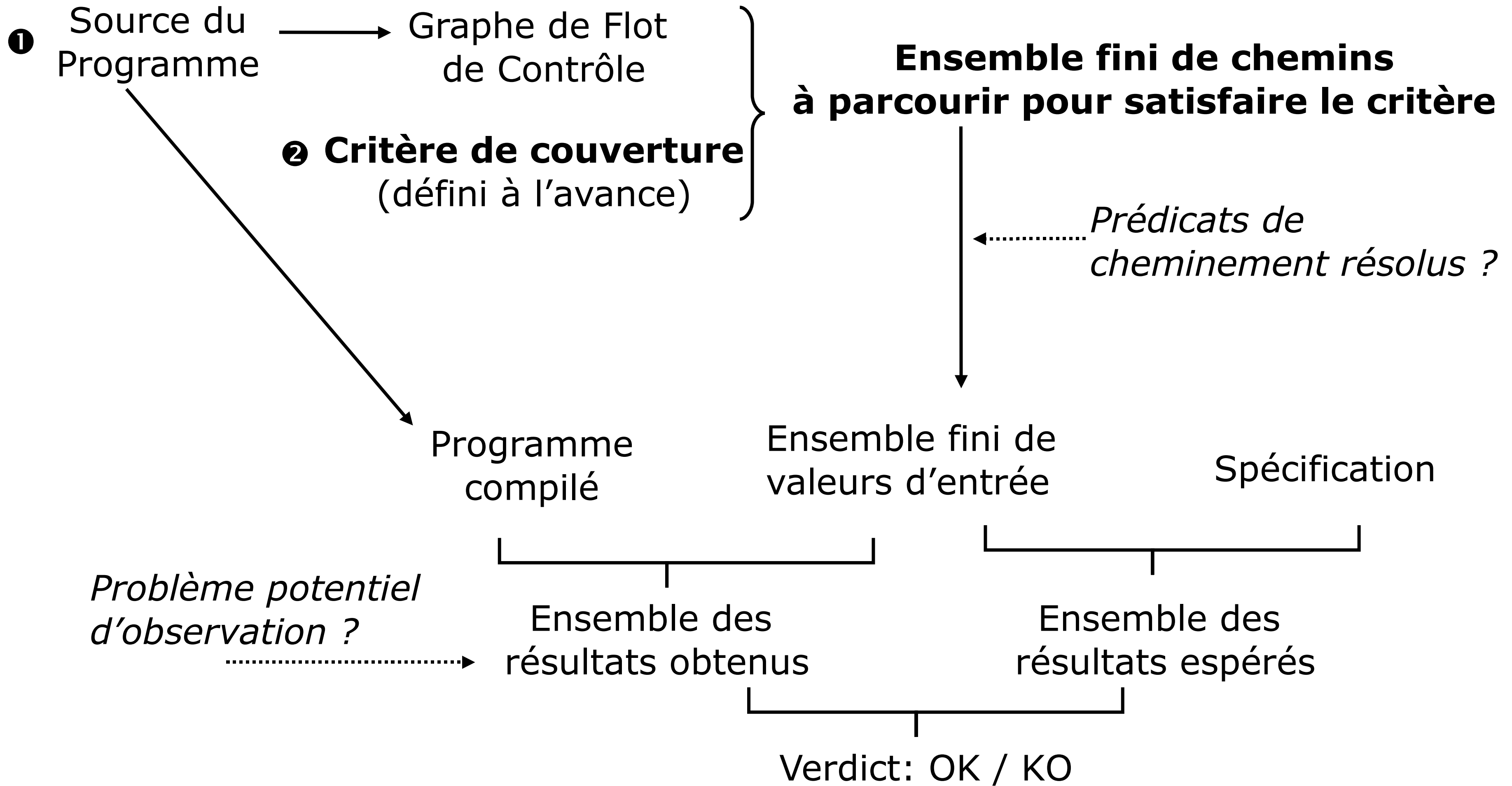
- ☞ we have again a finite number of paths
- ☞ the criterion is less constraining than $\text{AllTransitions}_k(M,T)$

A Hierarchy of Coverage Criteria

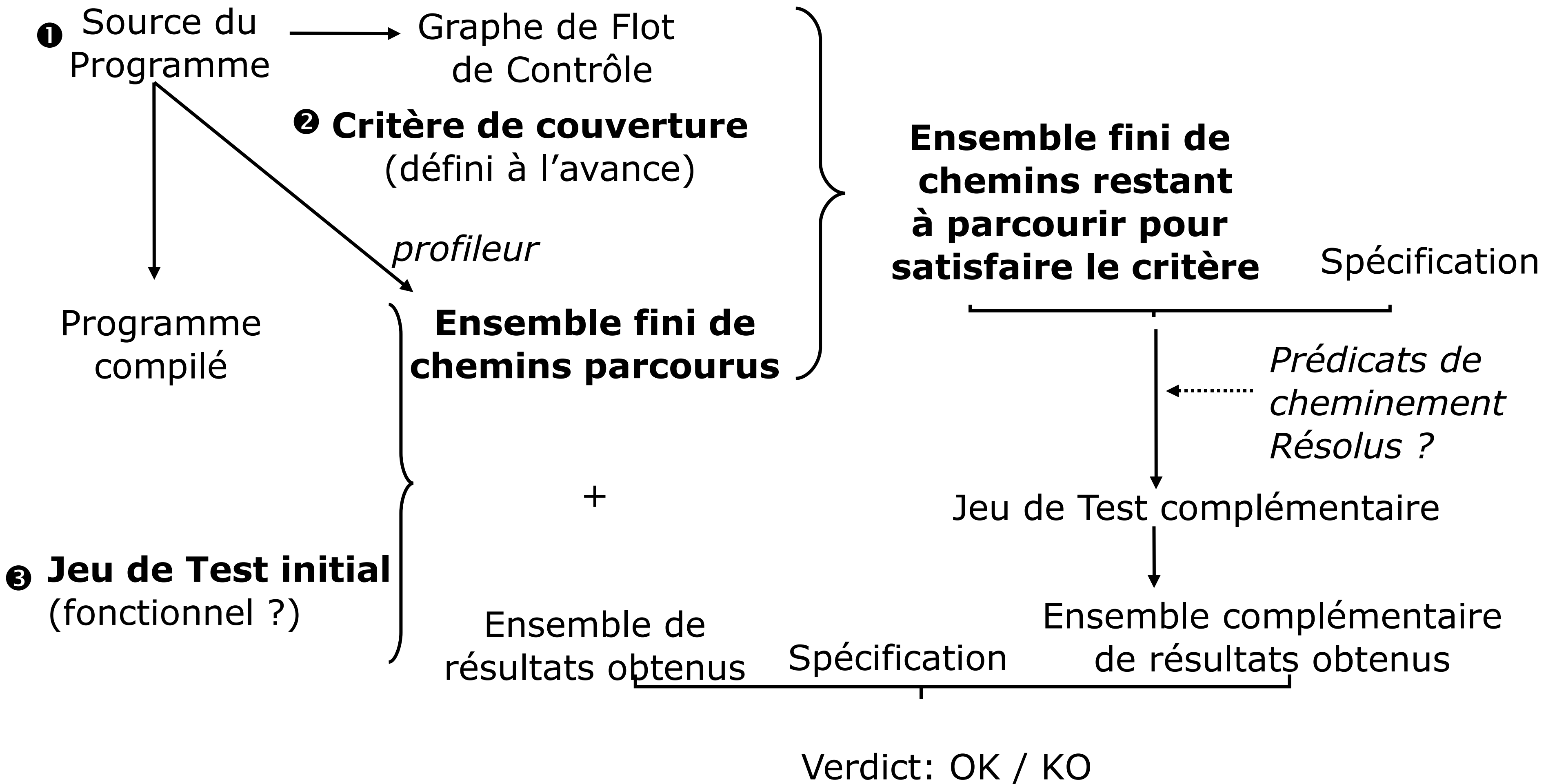
- AllPaths(M,T) ⇒
  AllPaths_k(M,T) ⇒
  AllTransitions(M,T) ⇒
  AllInstructions(M,T)

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

Using Coverage Criteria 1



Using Coverage Criteria 2

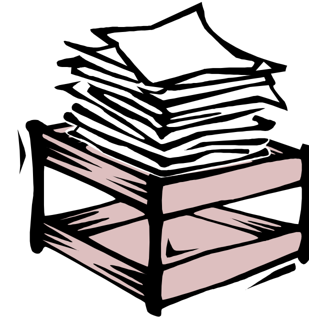


Summary

- 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

Annexe: Junit

Une méthode habituelle de test…



- Principalement des impressions…
  - « Plus il y en a, mieux c'est testé… »
  - « Il ne pourra pas dire que je n'ai pas testé ! »

- Qu'est ce qu'on en fait ???
  - Qui les lit (ou les relit après modification) ?
  - Qu'est-ce qui est réellement testé ?
  - Si le résultat attendu n'est pas précisé…
    … on ne peut pas dire que le résultat obtenu est correct !

- Qui les lit (en détail) ?
  - Vous, (probablement) la première fois
  - Probablement pas, pour chaque version (retestez-vous systématiquement d'ailleurs ?)

Le principe à adopter :

« Pas de nouvelle, bonne nouvelle »

  - Si tout va bien, pas de message
  - Si le test échoue, il doit le faire **explicitement**, **bruyamment** !
  - En lisant le code des tests **on sait facilement ce qui est testé et le résultat attendu**

- Ca ne devrait pas être une corvée de refaire passer les tests

- Les résultats ne sont pas explicitement précisés

```
MaClasse P1 = …, P2 = …;
if (P1.equals(P2))
    System.out.println("P1 est égal à P2" );
else System.out.println("P1 différent de P2");
…
System.out.println("P1.valeur: " + P1.valeur());
```

- Résultat ?

Mille milliards de damned,
P1 et P2 ils valent quoi, ici ?
Fau(drai)t que je recherche
dans le programme !

```
...
P1 différent de P2
P1.valeur: 12
```

« Tester » ou « Animer l'écran ou l'imprimante » ?

Normalement si on « teste P1 et P2» on devrait

En préalable

- Garantir que `P1` et `P2` sont dans le bon état (initialisation)
- Savoir ce que valent `P1` et `P2` à cet instant
- Savoir s'ils doivent être égaux ou différents
- Quelle est la valeur attendue de `P1.valeur()` à cet endroit

A l'exécution des tests

- Garantir qu'on râle si on n'obtient pas le « bon résultat » (valeur attendue **et** système dans l'état attendu)
- Pouvoir exécuter facilement plusieurs tests indépendants quels que soient leurs résultats
- **Dépouiller automatiquement** les tests
- Connaître facilement les tests en échec !

Ca devrait plutôt ressembler à ....

```
MaClasse P1, P2;
SetUp(P1);   // Construire les instances de test
SetUp(P2);   // et les amener dans l'état attendu
…
if (! P1.equals(P2))       // on sait ce qu'on veut !
    throw new ErreurTest(« Indication du problème »)
// else ; rien du tout !
if (P1.valeur() != 12)
    throw new ErreurTest(« Indication du problème »)
// else ; rien du tout !
```

- ***Plus compliqué: vérifier si une fonction lève les exceptions attendues!***
- ***Penser aussi à vérifier « l'état du système » et pas seulement le résultat retourné***

Ce qu'il faudrait faire théoriquement

- Globalement : **définir un plan de test** (et le document associé) !
  - Qu'est-ce qu'on teste ? **Dans quel ordre ? Avec quels incréments ?**
  - Quels « **bouchons** » et « **pilotes** » nécessaires ?
  - Pour chaque classe/méthode : **quels sont les cas à tester, quels résultats ?**

- Pour chaque test :
  - Définir l'environnement de test : classes et **instances** utilisées (et déjà testées), **dans le bon état** !
  - Partir d'un **état connu**, pour lequel la méthode est autorisée
  - Exécuter la méthode avec les bons paramètres
  - Tester la valeur retournée **et** l'état dans lequel on laisse le système

  le tout en gérant les exceptions attendues/inattendues

- Avoir prévu les bons **observateurs** pour tester le résultat et l'état
- Il faut avoir **tester/prouver ces observateurs** (pas de biais !)

Tester/prouver les observateurs ?

- Un exemple en C :

```
enum Couleur { Vert = 1, Orange = 2, … }
int testeCouleur(Couleur C) { return couleur == C; }
int testeCouleur(Couleur C) { return couleur = C; }

Facile de se tromper, moins de faire la différence !
```

- En Java :
  - On ne peut pas garantir qu'une méthode ne modifie pas son receveur
  - Il est facile de se tromper dans un « couper/coller » => on utilise un attribut à la place d'un autre !

JUNIT: la « plomberie » nécessaire !

- Faciliter les différents tests élémentaires (fonctionnels, unitaires)
  - Les différentes égalités…
  - La gestion des exceptions …
- Automatiser le lancement et le dépouillement des tests
  - Ne pas arrêter la campagne de test au premier échec
  - Garantir l’état dans lequel on démarre chaque test
- Faciliter l’ajout de nouveaux tests
- Fournir un rapport de test

En bref, un pilote (driver) de test sinon « intelligent »
du moins « serviable » et « corvéable »…

Junit : les primitives disponibles

- Des fonctions pour exprimer les résultats attendus
  `assertEquals()` pour les types prédéfinis, `Object` et `String`
  // teste `null` et utilise la version courante de `equals()`
  `assertTrue()`, `assertFalse()`
  `assertSame()`, `assertNotSame()` // avec == au lieu de `equals()`
  `fail()` // provoque une exception
  et qui lèvent une exception en cas de résultat inattendu

- Le pilote est « robuste » par rapport aux exceptions levées

- Le pilote repère « syntaxiquement » et dynamiquement les fonctions de test => facile d'en ajouter/supprimer.
- Il garantit l'état de départ des tests quel que soit le résultat du test précédent

le tout de façon simple et concise…

Junit : les principes

- Une classe de test `TestC` qui dérive de la classe `TestCase`
  Attention donc aux problèmes de **visibilité des champs**…

- La classe `TestC` peut définir :
  - Des variables d'instance : utiles pour les différents tests
  - Une méthode `setUp` systématiquement appelée **avant** chaque méthode de test (pour mettre les variables d'instance dans le bon état)
  - Une méthode `tearDown` systématiquement appelée **après** chaque méthode de test (nettoyage, fermeture de fichiers, …)
  - Des méthodes public void `test…()` implicitement considérées comme les fonctions de test
- On peut prévoir une méthode de test par méthode implémentée, ou par état d'une machine à états, ou …

Junit : les principes (suite)

- Les méthodes de test ne doivent pas propager d'exception (sinon le test est considéré comme en échec)

- Junit utilise l'introspection Java pour retrouver l'ensemble des méthodes de test : celles d'en-tete `public void test…()`

- Junit enchaîne les appels aux méthodes de test en les encapsulant avec les appels à `setUp` et `tearDown`, et compte les échecs et les succès

- Variante : en utilisant une classe interne qui dérive de `ExceptionTestCase`, on peut aussi spécifier des méthodes pour tester des méthodes qui doivent lever des exceptions.

Graphe de flot de données

On s'intéresse à l'usage fait des variables !

- On distingue "*définition" et "utilisation"* des variables:
  - x := y + 1 -- une *utilisation* de y et une *définition* de x
  - x := x + 1 -- l'*utilisation* de x précède sa *définition*

- Dans le graphe de flot de contrôle on annote chaque occurrence de x comme étant soit une utilisation u_x, soit une définition d_x

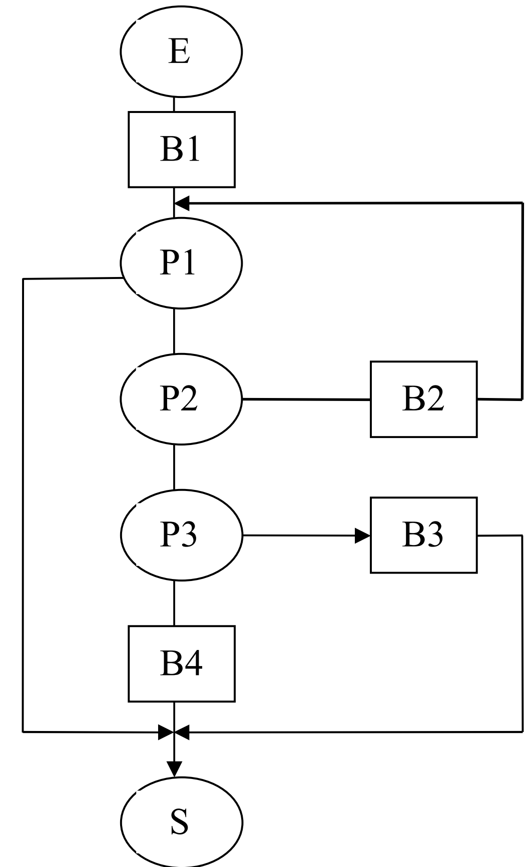
- On découpe les blocs élémentaires de façon à ce qu'aucun bloc ne contienne deux définitions de x ou une définition de x suivie d'une utilisation ultérieure de x dans le bloc

  petite contrainte technique qu'on peut toujours satisfaire…

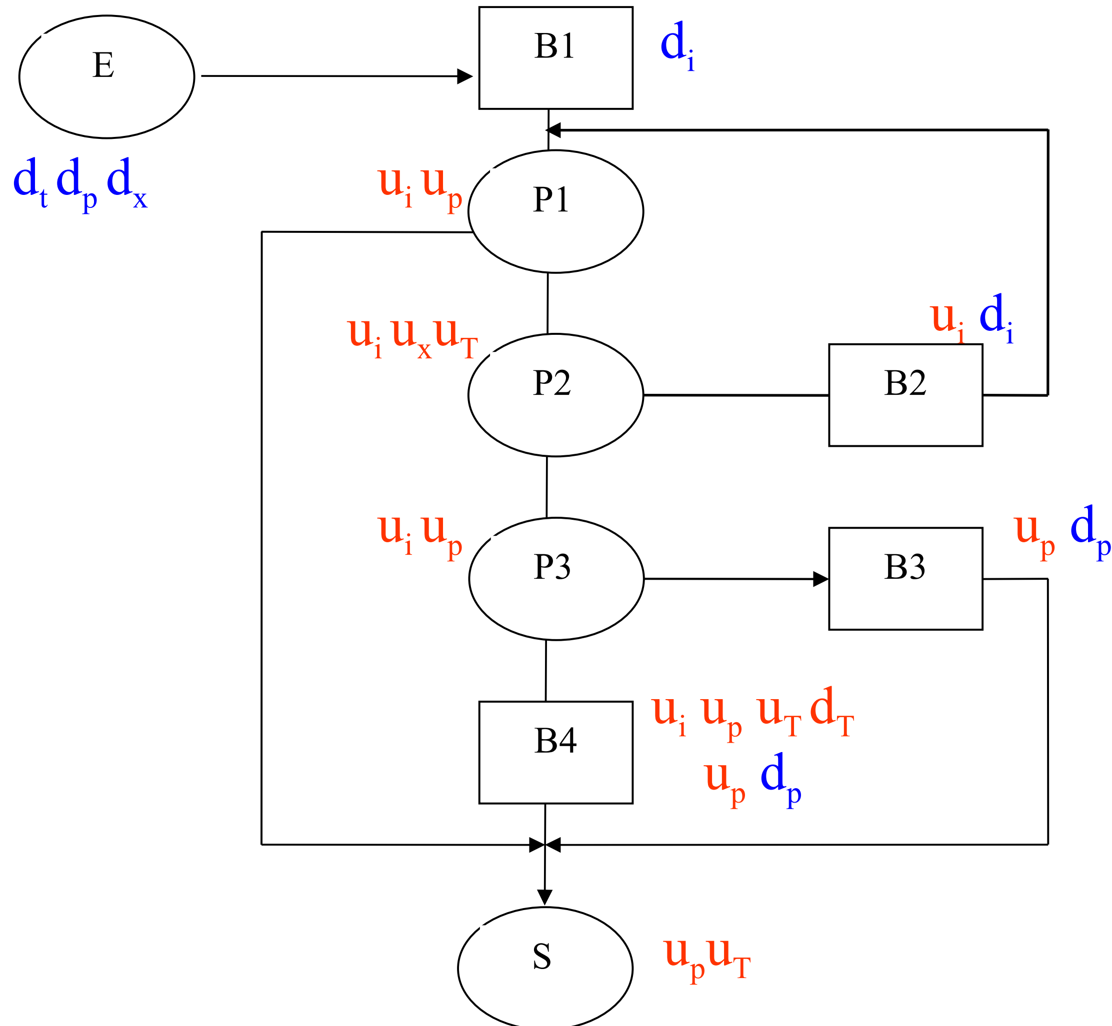
- Les prédicats sont sans « effet de bord » et comportent au

Le graphe de flot de données pour "supprime"

```
procedure supprime (T: in out Table;
                    p: in out integer;
                    x: 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;
```



Le graphe de flot de données annoté…



Soit le fragment de programme suivant:

```
procedure main()is
  monT: Table := …;
  monP: integer := …;
  monX: integer := …;
begin
   …
    supprime(monT, monP, monX);
   …
end main;
```

Donnez l'annotation correspondant à l'appel à `supprime`…

Flot de données (suite)

On va s'intéresser aux couples utilisation/ définitions d'une variable…

- Quelles sont les "définitions" qui peuvent atteindre une "utilisation" ?
- Où cette définition est-elle utilisée par la suite ?

…

- Statiquement: à une occurrence d'une utilisation il peut correspondre un **ensemble** de définitions **possibles**
- Dynamiquement: pour chaque exécution de cette utilisation, il n'y aura qu'**une** définition qui aura fourni la valeur courante …

Anomalies du flot de données

- Une variable x est-elle utilisée/définie raisonnablement ?

  Un chemin du programme présente une « **anomalie** » pour `x` s'il est de l'une des formes suivantes (on oublie les autres variables):
  - u_x…
  - …d_x
  - … $d_x d_x$…

- Il faut raisonner sur l'ensemble des chemins possibles (y compris les alternatives et les boucles)

  `supprime` *présente-t-elle des anomalies du flot de données ?*

- Si un graphe de flot de données a des anomalies, les critères vus ultérieurement peuvent ne plus avoir de solutions **;-(**

Anomalies du flot de données (suite)

- Anomalie ≠ Erreur !
  Mais il pourra être délicat de « tester » certaines définitions : comment savoir si on a mis la « bonne » valeur si on ne s'en sert jamais !

- Que fait-on des chemins infaisables ? ☹

- raisonnement « approximatif » (problèmes indécidables) **;-(**

Quelles approximations ?

Retour sur les définitions...

- Une définition *atteint* une utilisation s'il existe **au moins un** chemin entre la définition et l'utilisation sans redéfinition de la variable :

  « définition » : une instruction qui affecte - *ou peut affecter* - la valeur d'une variable.

- Une définition est *tuée* sur un chemin C s'il existe dans C une définition « non-ambiguë » de la variable considérée.

- Non-ambiguë ?
  - Appel de procédure `P(x, y)`: `x` peut-elle être modifiée par `P` ?
  - Accès à un élément de tableau: `T[i]:=1` tue-t-elle `T[j]` ?
  - Même problème pour une affectation via un pointeur.

Et encore, on ne considère pas la « faisabilité » des chemins !

Anomalie du flot de données ou pas ?

```
    i  := … ;
d T[i]  T[i]  := v;
    …
    i  := i +1;
    …
d T[i]  T[i]  := v;
```

```
    i  := … ;
    T[i]  := v;
d T[i]
    …
    j  := i;
    …
    T[j]  := v;
d T[j]
```

On pourrait insérer les instructions en italique dans une conditionnelle afin d'empêcher un raisonnement statique !

On sait traiter certains cas (ex. des accès statiques), sinon on fait des approximations et on raisonne au cas par cas.

Calcul des utilisations “atteintes” par une définition

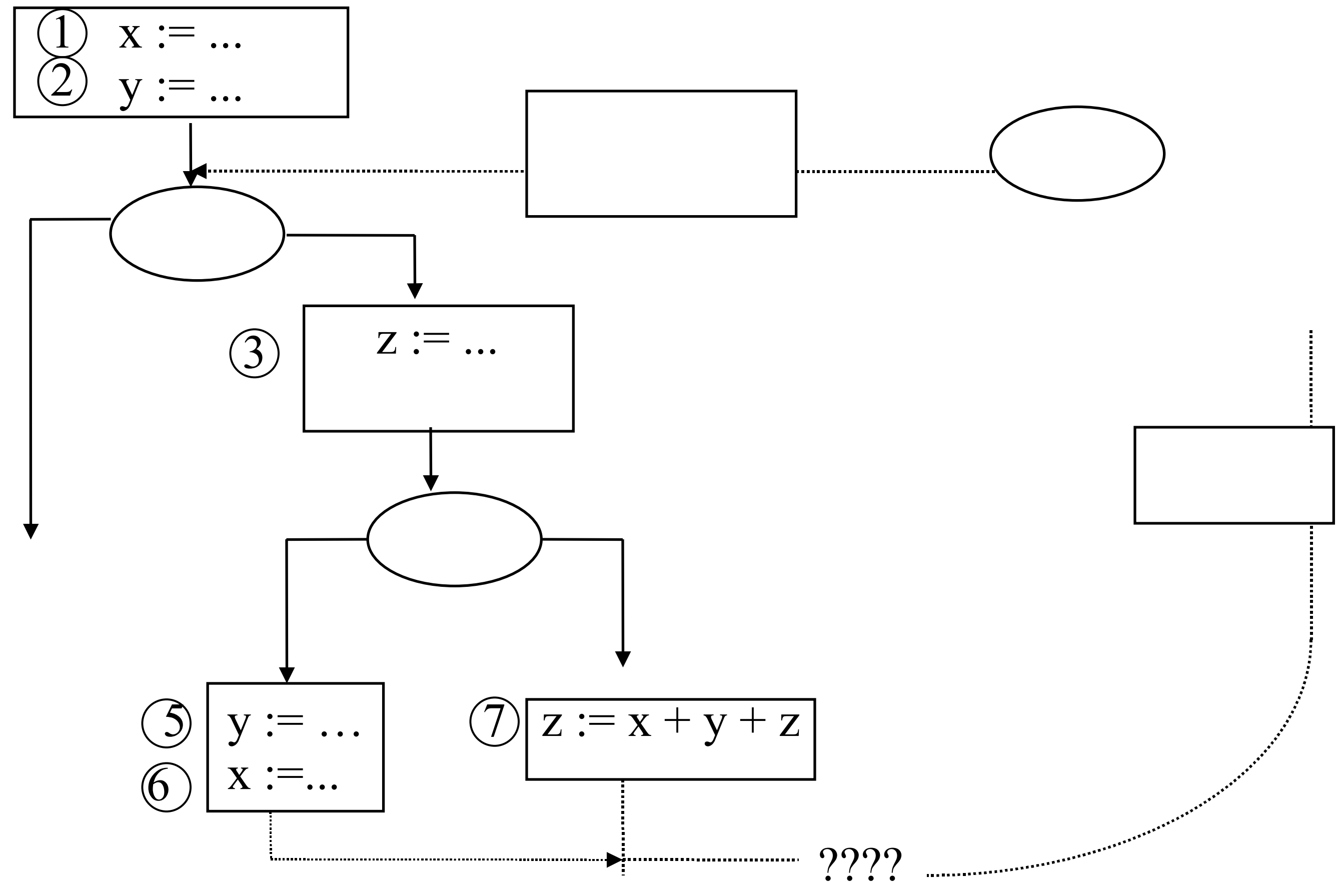
« On voudrait “tester” une définition de x en imposant le passage par une (plusieurs) instruction(s) utilisant la nouvelle valeur de x »

- Comment calculer l’ensemble des utilisations d’une variable dont la valeur peut provenir d’une définition donnée ?
- Comment connaître l’ensemble des définitions qui peuvent fournir une valeur qui va atteindre une utilisation ?
- A l’intérieur d’un bloc:
  par examen des instructions du bloc !

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

- Entre deux blocs distants : quelles sont les utilisations en dehors du bloc que `x` et `z` peuvent influencer ?

Exemple...



Quelles sont les définitions qui influent sur les utilisations dans ⑦

Equations de flot de données

GEN(B) : les définitions dans B et non redéfinies par la suite dans B

KILL(B) : les définitions hors de B dont la cible est redéfinie dans B

IN(B) : les définitions qui peuvent atteindre l'entrée du bloc B

OUT(B) : les définitions qui peuvent atteindre la sortie de B

Relations mutuelles entre ces ensembles ?

Comment les calculer si elles sont en inter-dépendance ?

- Il y a un nombre fini de définitions dans le programme
- *GEN* et *KILL* ne dépendent que de *B*:
  - ☞ **calcul en une passe**
- *IN* et *OUT* dépendent potentiellement de tout le programme:
  - ☞ **calcul par approximations successives** (itérations)

Equations de flot de données (suite)

Pour tout B:

$$OUT(B) = GEN(B) + (IN(B) - KILL(B))$$

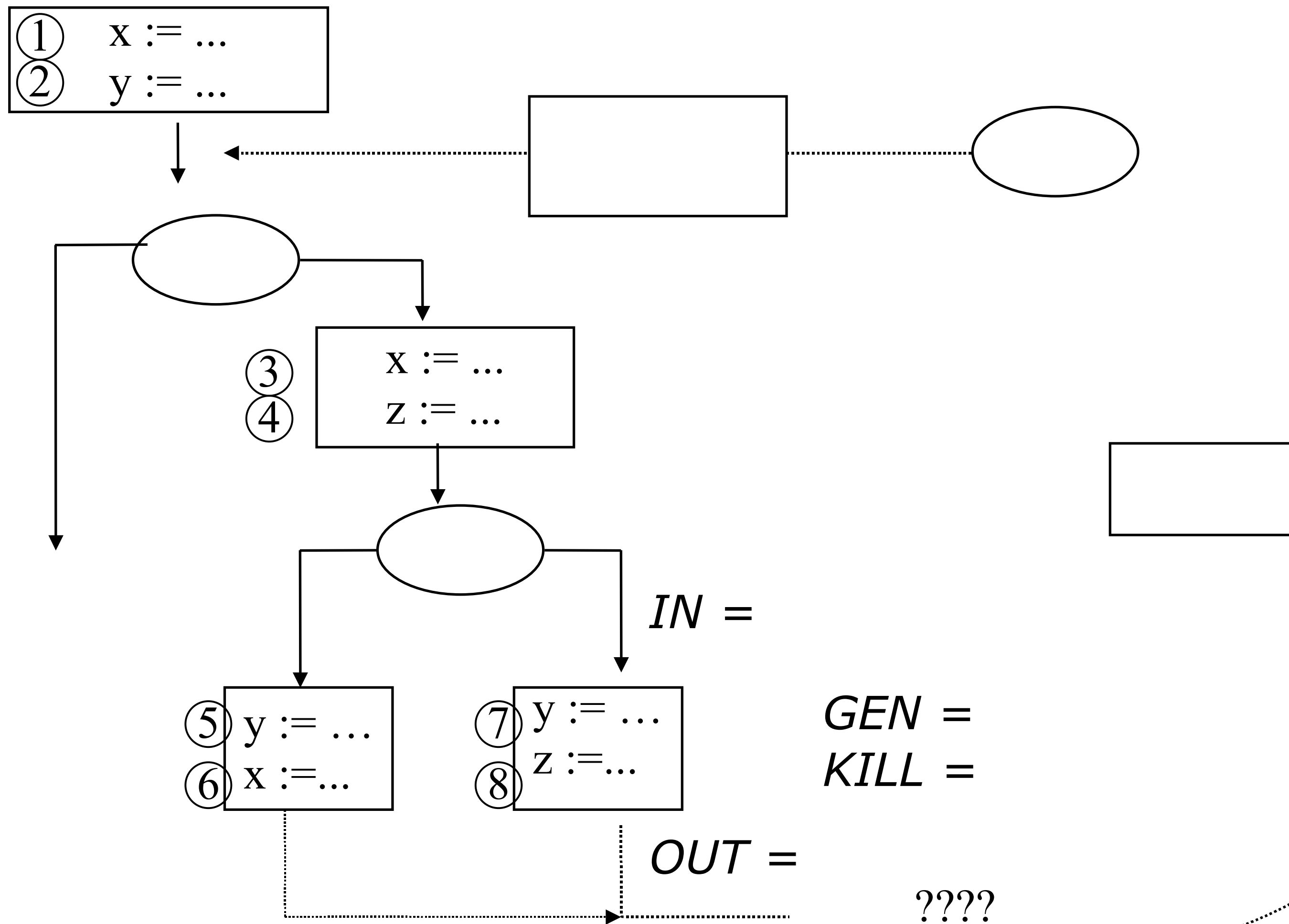
$$IN(B) = \bigcup OUT(B'), \text{ pour tout } B' \text{ prédécesseur direct de } B$$

- On initialise les ensembles IN et OUT à $\emptyset$ et on calcule par approximations successives jusqu'à saturation des ensembles…

  On ne fait que rajouter des définitions et il y en a un nombre fini => la propagation se terminera forcément !

- La propagation suit le flot de contrôle: OUT = *f*(IN)

In, *Out*, *Gen* et *Kill* sur un exemple...



Retour sur les définitions (fin)

- On fait une analyse précautionneuse (« conservatrice ») des dépendances :
  - Le véritable GEN(B), à l'exécution, est un sous-ensemble du GEN(B) qu'on a calculé
  - le véritable KILL(B) est un sur-ensemble du KILL(B) calculé

On se préoccupe donc de dépendances qui peuvent ne pas exister !

D’autres analyses du flot de données

Technique très utilisée en compilation (optimisation)

- Calcul des « variables vivantes » : une variable est « vivante » en un point du programme si sa valeur courante est utilisée dans la suite du programme
- Calcul des sous-expressions communes : deux expressions syntaxiquement identiques sont-elles partageables (i.e. elles travaillent toujours avec les mêmes valeurs pour les variables)

Ces deux problèmes s’expriment aussi sous forme d’équations de flot de données, i.e. de la forme

$IN = f(OUT, GEN, KILL)$ ou bien $OUT=f(IN, GEN, KILL)$

où f est un opérateur ensembliste ($\cup$, $\cap$, ...)

☞ À chaque fois on définit la bonne notion de IN, OUT, GEN, KILL

Critères de couverture liés au flot de données

- Notations:
  - $d_B(x)$: le bloc B contient une (unique) définition de x…
  - $u_B(x)$: le bloc B contient une utilisation de x…

Ici « bloc » veut dire bloc d'instructions ou prédicat !

- Une définition $d_B(x)$ atteint une utilisation $u_{B'}(x)$ ssi
  - soit $B = B'$, $d_B(x)$ précède $u_{B'}(x)$ et il n'y a aucune autre définition de x entre ces deux instructions
  - soit $d_B(x) \in IN(B')$, et il n'y a pas de définition de x entre le début de B' et $u_{B'}(x)$

Critère "toutes les définitions"

toutes-les-définitions(M, T)

ssi

pour tout x,

pour tout $d_B(x)$,

il existe au moins une $u_{B'}(x)$ t.q.

il existe un chemin B C B′ dans T

où C est sans définition de x

Ici et dans la suite, le sous-chemin C peut être vide !

Forme du critère : $\forall x \ \forall d_B(x) \ \exists u_{B'}(x) \ \exists BCB'$

En clair :

« **toutes les *définitions* sont *utilisées*** au moins une fois »

Critère “toutes les utilisations”

toutes-les-utilisations(M, T)   *ssi*

pour tout x, pour tout $d_B(x)$,

pour toute $u_{B'}(x)$ atteinte par $d_B(x)$,

pour tout B’’ successeur de B’ dans G

il existe un chemin de T qui contient B C B’ B’’ t.q.

C est sans définition de x

Forme du critère : $\forall x\ \forall d_B(x)\ \forall\ u_{B'}(x)\ \forall\ B''\ \exists\ BCB'B''$ dans T

« Toutes les utilisations atteignables de chaque définition, en couvrant les deux branches quand B’ est un prédicat. »

La contrainte sur les successeurs de B’ garantit la couverture du critère « tous les enchaînements », lorsque B’ est un prédicat.

Notes*: B’’ est sans contrainte vis-à-vis des utilisations.*
B’’ doit suivre immédiatement B’ dans le chemin.

Critère "tous les DU-chemins"

tous-les-DU-chemins(M, T)

ssi

pour tout $d_B(x)$,

pour toute $u_{B'}(x)$ atteinte par $d_B(x)$,

pour tout B'' successeur de B' dans G

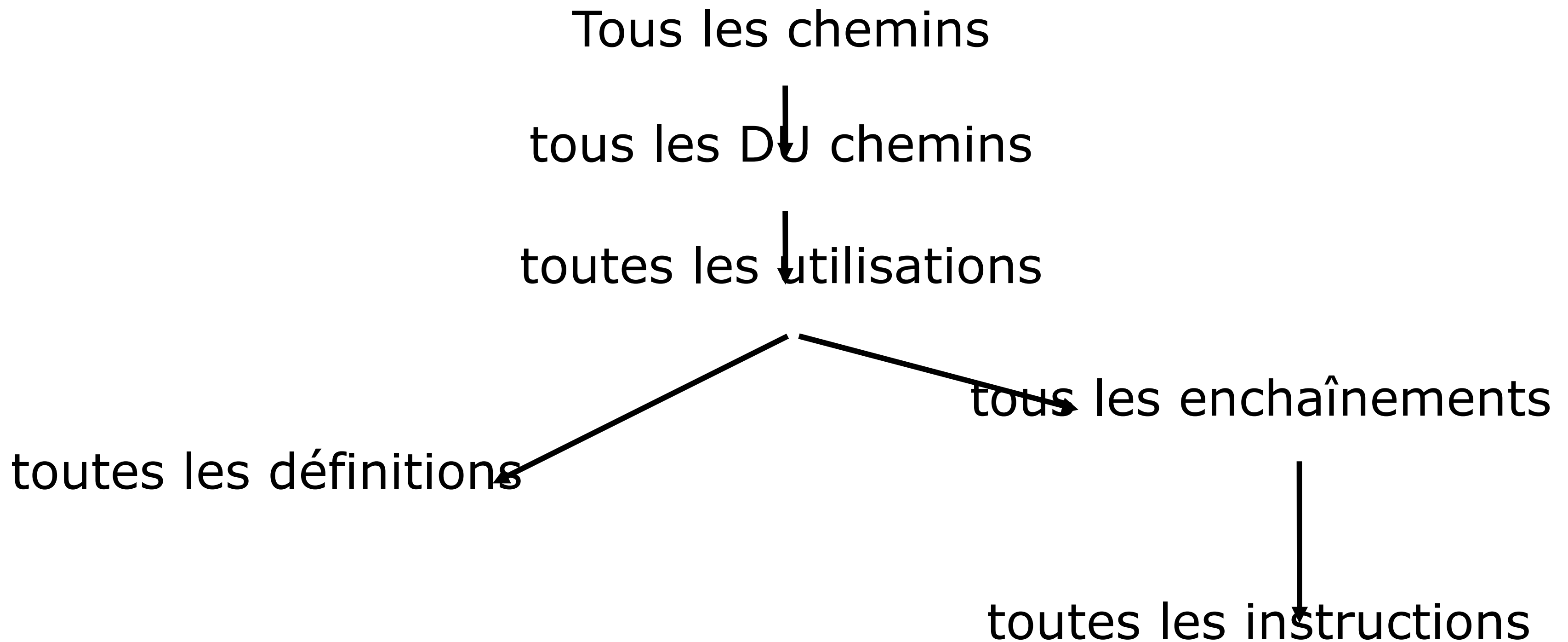
pour tout sous-chemin B C B' B'' t.q.

1. C est sans définition de x
2. B C B' est sans circuit ou est un circuit élémentaire

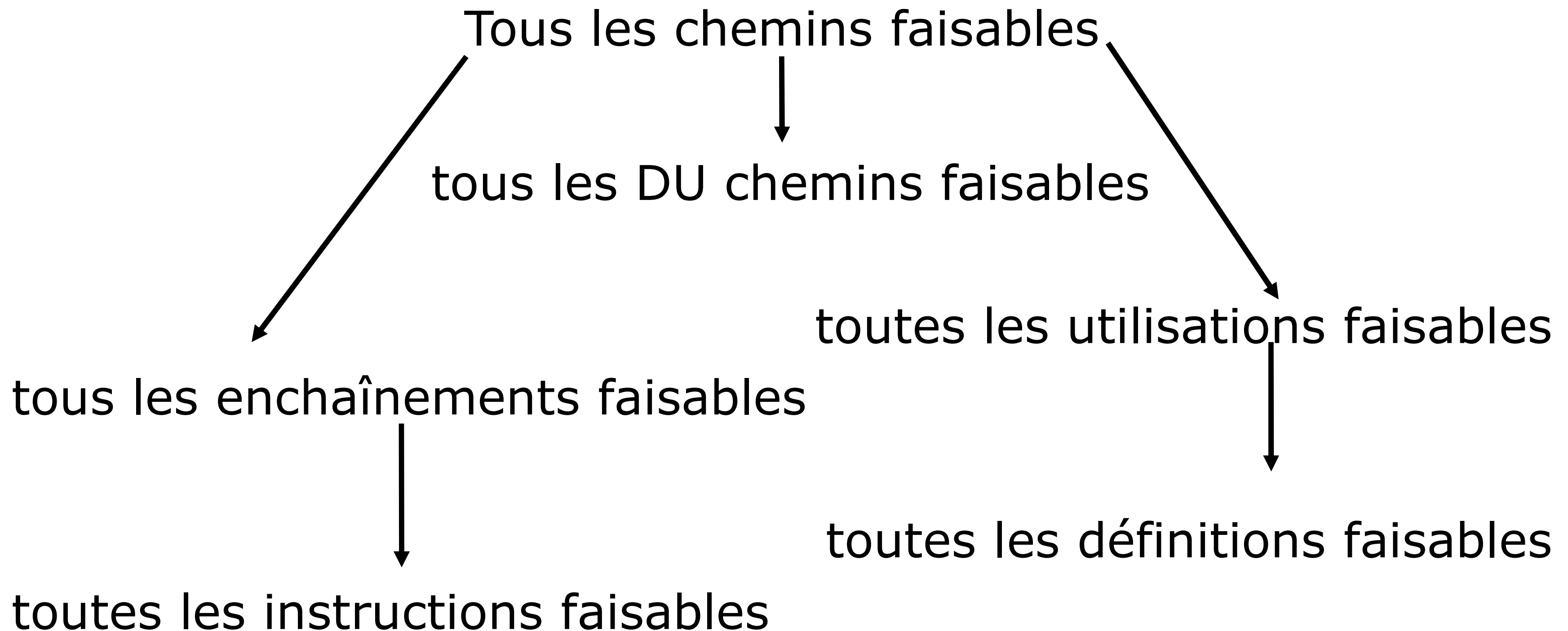
alors B C B' B'' est contenu dans un chemin de T

Forme du critère : $\forall d_B(x)$ $\forall u_{B'}(x)$ $\forall B''$ $\forall BCB'$, *restreint de façon à avoir un nombre fini de chemins.*

Hiérarchies entre (quelques) critères (1)



Hiérarchies entre (quelques) critères (2)



Conclusion sur le test dynamique structurel

- Peut servir à engendrer des tests selon un critère donné
  - peu de test si on se contente de critères simples ☹
  - beaucoup de tests si on affine le critère (passages dans les boucles) ☹ ☺
  - passage ensemble de chemins / valeurs d'entrée ???
- Peut servir à **valider un jeu de test construit indépendamment**:

  Par quels chemins est-on passé ? Quels sont les chemins non testés ?

  <u>en fonction d'un critère énoncé à l'avance!</u>
- Ne peut pas mettre en évidence ce qui n'apparaît pas dans le code
- Remis en cause dès qu'on touche au code ! Ne pas le faire trop tôt...

Test et langages à objets…

Les programmes à objets ajoutent des difficultés au test

- interaction/coopération entre des méthodes « élémentaires »
  quel test unitaire ? Quels critères de couverture ?
  Explosion du nombre de classes, interfaces, méthodes

=> Privilégier les **interactions** entre méthodes

- Le résultat d'une méthode est fonction de l'état de l'objet tel qu'il a été défini par les interactions précédentes !

  « état » du système réparti sur de nombreux objets

  L'encapsulation empêche de tester facilement le résultat !

  Il faut être capable d'amener un objet dans le bon état avant de tester une transition !

Test et langages à objets…

- Coopération entre instances de différentes classes:
  par où commence-t-on ?
  Classes interdépendantes

  } *Stratégies d'intégration ?*
  Stubs + drivers à prévoir !

- Test et héritage (fonctions virtuelles):

  si on définit une nouvelle sous-classe d'une classe déjà testée:

  - que faut-il tester dans la nouvelle classe ?
  - Que faut-il re-tester dans la super-classe ?

- Comment tester des classes abstraites ?
  Tester une « collection » abstraite ?

IFIPS Institut de Formation d'Ingénieurs
UNIVERSITÉ PARIS-SUD 11

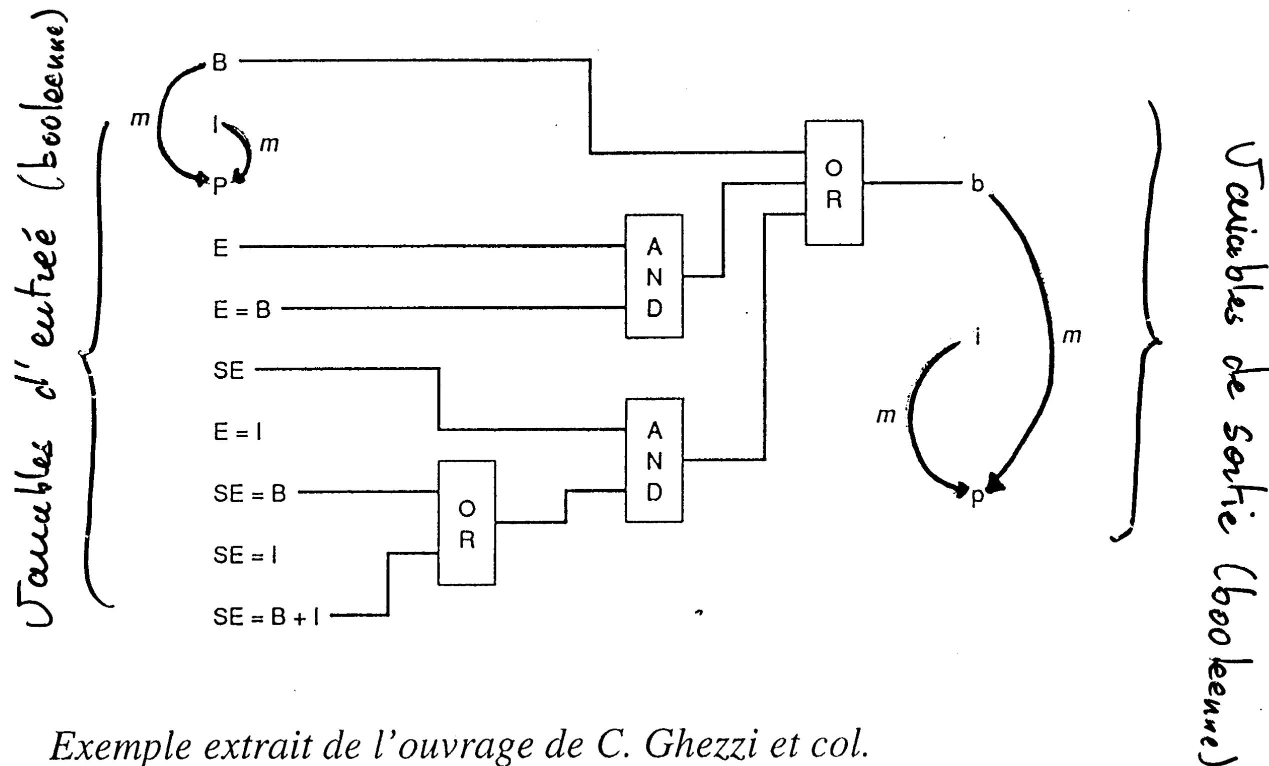
2008-2009

Test Fonctionnel

Test Fonctionnel (1)

- Les spécifications étant rarement formelles, il y a peu d'approches "systématiques":
  - "Tables de décision" ("matrices de test") : une matrice avec les conditions en entrée et les effets possibles, et on coche ce qui va ensemble…
  - "Parcours structurels" de spécifications semi-formelles (diagrammes SADT, scénarios UML, statecharts)
  - Graphes "causes-effet"
- Spécifications fonctionnelles formelles à la UML/OCL (*Object Constraint Language*)

Graphes Causes-Effets



Exemple extrait de l'ouvrage de C. Ghezzi et col.

B: « bold », P: « plain », I: « italic », E: « emphasize » (2 modes possibles)
SE: « super emphasize ».
m: « mask »

Test et langages à objets : travaux en cours

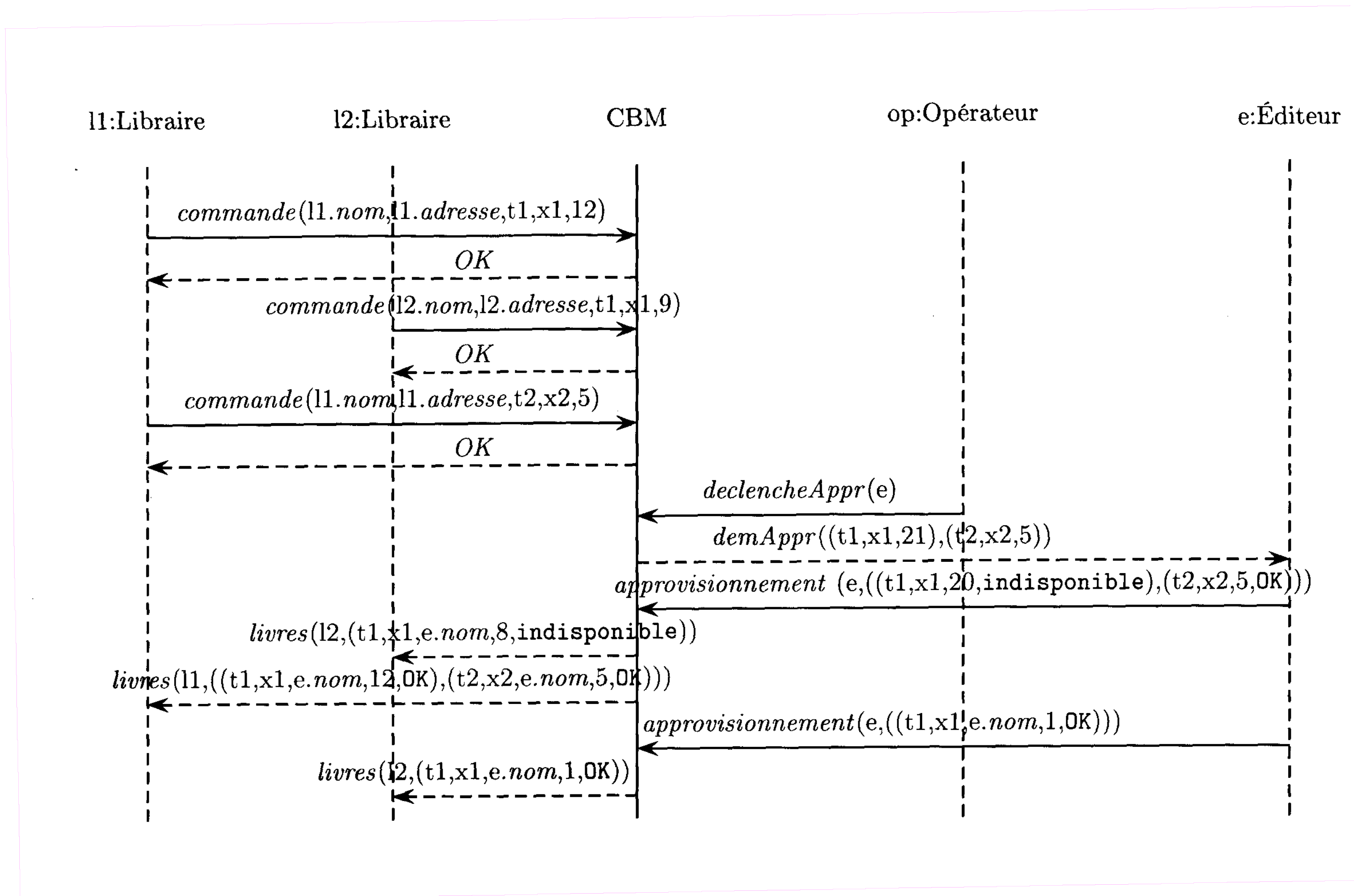
- Graphes d'appels entre méthodes + critères de couverture
  ≈ « Flot de contrôle »

- Critères liés aux définitions/utilisations d'attributs dans les méthodes
  ≈ « Flot de données »

- Test fonctionnels à partir de documents UML…
  - Test à partir de diagramme de séquences
  - Test à partir de « machines à états »
    Cela revient à « parcourir » différents « chemins/transitions »…
  ≈ « Critères de couverture » du graphe !

- « extensions » à UML pour décrire des documents de test et les exporter (« UML Testing Profile » à l'OMG) via les « tag values » et les « stéréotypes ».

Test à partir de diagrammes de séquences



Test à partir de diagrammes de séquences

En pratique:

- Instancier les diagrammes de séquence avec des valeurs précises
  (ex: scénarii, diagrammes associés aux cas d'utilisation)

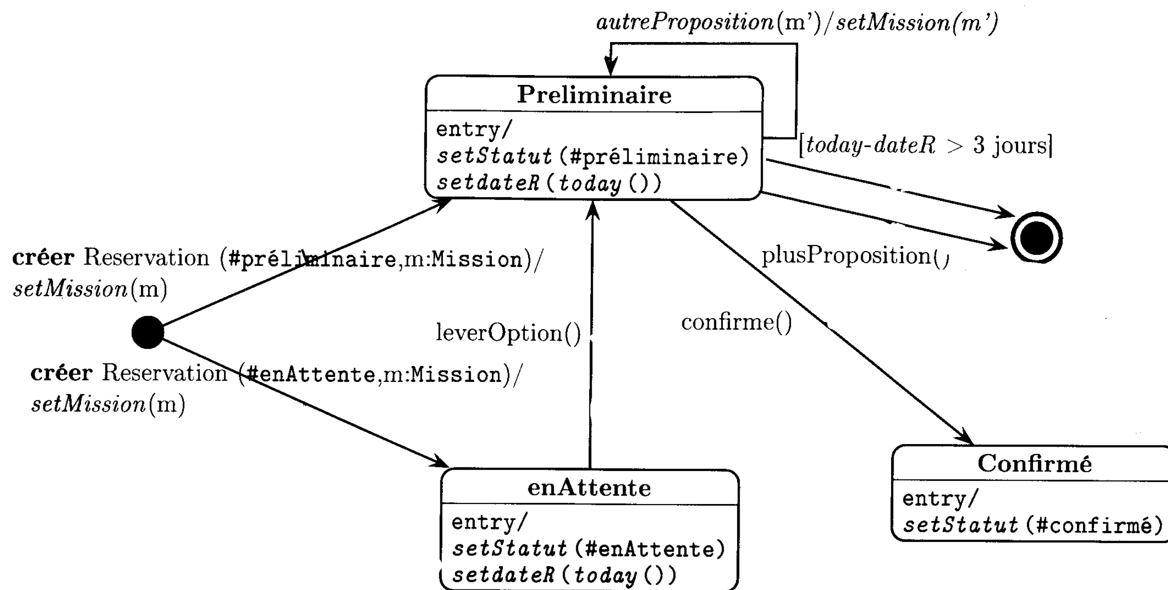
- La séquence doit se terminer par un **effet « observable »** (messages vers les agents extérieurs)

- Il faudrait aussi pouvoir **vérifier l'état du système**
  (=> observateurs à prévoir en plus)

- Construire un ensemble d'instances dans le bon état pour pouvoir jouer ce scénario !

Test à partir des diagrammes d'états

- Soit le diagramme d'état suivant qui décrit le cycle de vie d'une réservation:

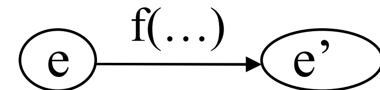


- Quels tests faut-il prévoir ? Comment les spécifier ?

Test à partir des diagrammes d’états

En pratique:

- Tester au moins une fois chaque transition:



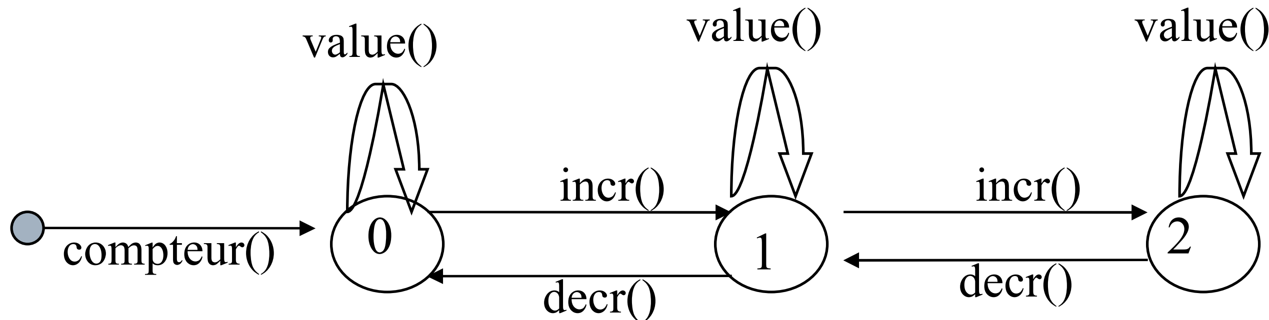
- Pour tester **une** transition ?
  Construire une instance et l’amener dans le bon état
  Vérifier qu’on est dans le bon état
  Vérifier que la garde est satisfaite
  Activer la transition (=> appel de méthode ?)
  Vérifier l’état d’arrivée

  Nécessite à nouveau d’avoir de bons observateurs !

  Plus compliqué si l’événement n’est pas un appel de méthode !

- On peut « factoriser » les préfixes des séquences de test

Exemple : un compteur à 3 états…



Transitions « observations »	**« trace »**	
(-, compteur(), 0)	(new compteur()).value()	0
(0, value(), 0)	A = new Compteur();	
	A.value(); A.value();	0, 0
(0, incr(), 1)	A = new Compteur();	1
	A.incr(); A.value();	
(1, value(), 1)	…	

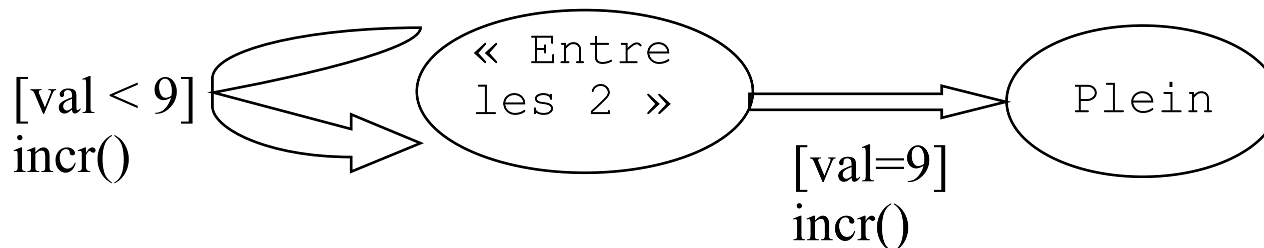
On profite du fait que Compteur() est déjà testé !

Q: Est-ce raisonnable d'utiliser value() ?
(Ex: `int value(int v) { val = v; }` *)*

Exemple (suite)

- Si `value()` garantit qu'il ne modifie pas son receveur, on peut se dispenser de certaines observations
  Il faudrait peut-être mieux « prouver » la correction de `value()` ?

- Question: Si le compteur avait 10 valeurs possibles ?
  - On fait preuve d'« abstraction » (« équivalence » entre états)
  - Quelle « équivalence » ? Un **choix** lié à la stratégie de test ! Exemple: `vide`, `plein`, « `entre les deux` » …
  - Plusieurs transitions d'un état pour la même opération: automate non-déterministe:



Test Fonctionnel à partir de UML-OCL

- Fonctionnel => la spécification sert pour la sélection des tests et comme oracle pour décider du succès des tests…

  - N'a-t-on pas oublié de cas = couvre-t-on tous les cas mentionnés dans les documents d'analyse et de conception ? (cas d'utilisation, diagrammes de séquence, scénario)

- Cas où la spécification est une formule logique :
  - On la met sous **forme normale disjonctive**, en assurant que les cas sont **disjoints**

    $C1 \vee C2 \vee \ldots \vee Cn,$ avec $Ci \neq Cj$ si $i \neq j$
  - on prévoit un test par cas ...

Exemple

$$max \geq x \wedge max \geq y \wedge (max = x \vee max = y)$$

devient :

$$(max \geq x \wedge max \geq y \wedge (max = x \wedge max = y))$$
$$\vee (max \geq x \wedge max \geq y \wedge (max = x \wedge max \neq y))$$
$$\vee (max \geq x \wedge max \geq y \wedge (max \neq x \wedge max = y))$$

soit encore :

$$(max = x \wedge max = y) \vee (max = x \wedge max > y) \vee (max > x \wedge max = y)$$

dont on peut déduire un jeu de test:

{ (10, 10, *10*), (121, -72, *121*), (2, 54, *54*) }

Modèles d'opérations avec pré/post-conditions

Test de conformité : couvrent tous les cas satisfaisant la pré-condition

Test de robustesse : couvrent les cas qui ne satisfont pas la pré-condition (pas toujours utiles, comportement pas toujours défini)

Comme avant: on met la **conjonction** de la pré-condition et de la post-condition sous forme normale disjonctive !

context max(x, y : integer) : integer

pre*:* $x \geq 0$ and $y \geq 0$

post: $max \geq x$ and $max \geq y$ and $(max = x \vee max = y)$

on se ramène à: $(x \geq 0 \wedge y \geq 0 \wedge max = x \wedge max = y)$

$\vee\ (x \geq 0 \wedge y \geq 0 \wedge max = x \wedge max > y)$

$\vee\ (x \geq 0 \wedge y \geq 0 \wedge max > x \wedge max = y)$

Si la clause ***post*** est un ensemble d'implications

- si les prémisses des implications sont disjointes: on construit, pour chaque implication, un cas qui satisfait la prémisse et la pré-condition
- sinon, on réorganise la clause de manière à avoir des prémisses disjointes…

  context max(x, y : integer): integer

  pre*:* $x \geq 0$ and $y \geq 0$

  post: . $x \geq y$ *implies* max = x

  . $y \geq x$ *implies* max = y

donnera

. ($x \geq 0$ and $y \geq 0$ and $x > y$) *implies* max = x

. ($x \geq 0$ and $y \geq 0$ and $x = y$) *implies* max = x and max = y

. ($x \geq 0$ and $y \geq 0$ and $y > x$) *implies* max = y

et on procède comme avant !

Quelques techniques de test complémentaires

Test Statistique

" par tirage aléatoire selon une distribution des données d'entrée..."

- Possibilité d'avoir des jeux de test plus volumineux, engendrés automatiquement
- Mise en œuvre des lois de probabilité ? ☹
- Sait-on toujours décider du résultat attendu ? ☹
- Résultats expérimentaux très variables ... ☹
- Un moyen d'évaluer la fiabilité opérationnelle...

Test Statistique (suite)

- test aléatoire **uniforme**

  Quelques résultats expérimentaux (Duran et Ntafos) :

  couverture du graphe de flot de contrôle par des tests aléatoires de taille modérée (de 20 à 120), sur 5 programmes (sinus, coef. binomiaux, triangles, tri, recherche dichotomique):

  - *97 % des instructions*
  - *93 % des enchaînements*
  - *57 % des chemins avec 0, 1 ou 2 passages par boucle*

  A partir d'un seuil, la qualité augmente peu avec la taille…

- test aléatoire "**opérationnel**" : en respectant la distribution opérationnelles des données

  donne une indication de la fiabilité réelle du module

  il faut disposer et simuler la distribution des données

Test statistique structurel

- Choix d'une distribution pour satisfaire avec une probabilité minimale donnée un critère structurel…

M, un module, C un critère structurel → Sc(M): les constructions de M à activer

T : un jeu de test; N = | T |

q_N: « l'objectif de qualité »

T couvre C avec une probabilité q_N si tout élément de Sc(M) a une probabilité au moins q_N d'être activé par un test de T.

$P_c = \min \{ p_k \mid k \in Sc(M) \}$ doit être <u>*maximal*</u>

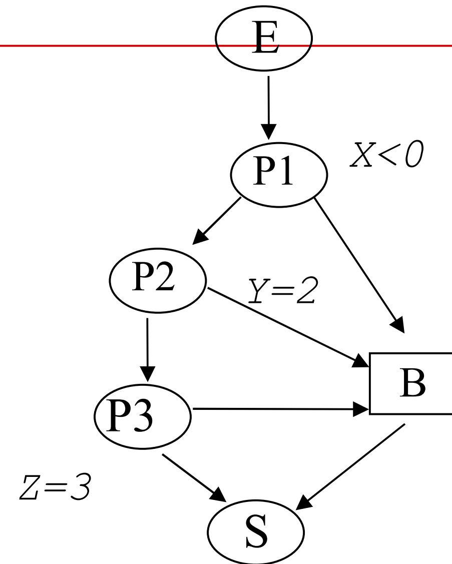
$(1 - P_c)^N = 1 - q_N$

N = nombre de tirages à faire étant donné C et un objectif de qualité Q ?

$N = \log(1 - Q) / \log(1 - P_c)$

N est la borne inférieure théorique du nombre de tests à faire…

Exemple…



- C = « tous les chemins » ;

4 chemins, de probabilités respectives :

$EP_1P_2P_3S$: $p(X \geq 0). p(Y \neq 2).p(Z=3)$

$EP_1P_2P_3BS$: $p(X \geq 0). p(Y \neq 2).p(Z \neq 3)$,

EP_1P_2BS : $p(X \geq 0). p(Y=2)$

EP_1BS : $p(X < 0)$

Pour avoir P_c maximum, on impose que la probabilité de chaque chemin soit 1/4, d'où $p(X < 0) = 1/4$, $p(Y=2) = 1/4 * 4/3 = 1/3$, $p(Z=3) = 1/2$

Si on impose d'avoir, par exemple $q_N = 0.99$, on trouve N = 16 …

☹ En supposant les variables indépendantes

☹ pas toujours aussi simple d'avoir la distribution

☹ N très grand pour arriver à tester les cas aux limites !!

Test statistique structurel (fin)

- test statistique structurel =

  test structurel

  + tirage aléatoire en imposant des probabilités de passage dans les différents chemins

  couvrir les chemins les moins fréquentés avec une probabilité minimale définie à l'avance (=> en déduire le nombre de test !)

  forcer le passage par les autres chemins par de nombreuses données (il ne suffit pas de tester une fois un chemin !)

Test **statique** par inspection

- Se déroule avant toute exécution…
- Réunion en petit comité (3/4) avec modérateur et travail préparatoire (lecture de documents relatifs au module)
- Pas d’analyse/correction des erreurs mises-à-jour
- Compte-rendu des erreurs trouvées pour correction ultérieure
- Méthode:
  - l’auteur expose sa solution
  - check-list des erreurs les plus courantes
- Efficacité:
  - 40% à 70%+ des erreurs de logique et de programmation ?
  - 150/200 instructions à l’heure ?

Un exemple de « Check-list »

Initialisation des variables ?

Constantes nommées ou littérales ?

Validité des prédicats dans les conditionnelles (=, <, ≤, ...)

Prédicats pour la terminaison des boucles (<, ≤, ...)

Bornes inférieures et supérieures des tableaux (0, 1, taille, taille-1)

Délimiteurs des chaînes de caractères (ex: \0 en fin de chaîne de en C)

Correction des allocations dynamiques (allocation/désallocation)

Passage de paramètres dans les sous-programmes (selon les langages)

Gestion des liens dans les listes chaînées

Parenthésage des instructions dans les boucles/conditionnelles

Parenthésage des expressions booléennes

Traitement des exceptions (récupération, propagation)

...

Test statique par évaluation symbolique

- Exécuter **symboliquement** le programme sur des données plutôt qu'en effectuant explicitement les calculs:

  le module + un chemin =>

  le prédicat de cheminement (comme précédemment)

  \+ **les expressions symboliques calculées des variables de sortie**

- Rien n'empêche de faire dynamiquement les calculs en même temps ("exécution symbolique dynamique")

- Permet d'éviter des résultats corrects accidentels!

- Si on veut raisonner sur des boucles (nombre infini de chemins) => oblige à expliciter des "invariants de boucle" (cf. le cours sur la preuve)

Test statique par analyse

Outils d'analyse de programmes…

Exemple: « Logiscope » (Verilog), Clover

- Analyse du graphe de flot de contrôle, de flot de données
- Détection d'anomalies
- Graphe d'appels entre modules
- Mesures de complexité (Halstead)
- Chemins d'appels entre procédures, entre prédicats, avec conditions de parcours à satisfaire
- Analyse par "évaluation partielle": approximation des calculs
- Instrumentation du code

Conclusion: quels types de test ?

- test fonctionnel / test structurel ?
  - le test structurel ne suffit pas: il ne peut pas toujours révéler des parties manquantes
  - le test fonctionnel ne suffit pas: on ne peut pas ignorer comment
    le module réalise ce qu'il a à effectuer.

- test aléatoire / test déterministe ?
  - On ne peut pas ignorer le "profil opératoire" des données
  - On ne peut pas non plus ignorer les cas particuliers (ex. test aux limites) difficiles à atteindre par test aléatoire

Conclusion (suite)

- Enchaîner ?
  - lectures croisées, inspection: <u>économique,</u> <u>rentable</u>
  - test aléatoire
  - test fonctionnel: préparés lors des phases amont du cycle de vie
  - test structurel: à refaire dès qu'on touche au programme !

- Aucune méthode ne suffit à elle seule …

Test d’intégration

- Le moment où on met ensemble un certain nombre de modules et on découvre:
  - le flou dans les interfaces (nombre et type des paramètres)
  - le flou dans les spécifications (qui vérifiait quoi ?)
  - le flou dans les performances de chaque module
  - l’absence de convention de normes de programmation (nommage des objets…)
  - l’existence de « jumeaux » (voire plus) et l’absence de « fantômes »

- On ne contrôle plus d’où viennent les erreurs (localisation), il faut modifier des modules donc les retester unitairement

Test d'intégration (suite)

- Intégration progressive et non pas « big bang »
- Ecriture de « lanceurs » et « bouchons » pour simuler les modules non encore intégrés
- Définir à l'avance les **incréments d'intégration**:
  - quelles sont les classes qui doivent être testées ensemble ?
  - Que rajoute-t-on après ? Dans quel ordre ?

  Définir des « clusters »: groupes de modules à traiter ensemble
- A faire dès l'étape de conception architecturale (cycle de vie…)

IFIPS Institut de Formation d'Ingénieurs
UNIVERSITÉ PARIS-SUD 11

2008-2009

Introduction à la preuve de programmes

Pourquoi prouver?

- tester ne suffit pas toujours…pour des niveaux de qualité très élevés
  On ne peut pas tester exhaustivement !

- le test c'est coûteux de toutes façons, pour quelle certitude ?

=> pourquoi ne pas essayer de prouver ?

- même si on ne prouve pas tout un logiciel mais seulement certaines parties "critiques"

Rôles de la preuve?

- la preuve comme moyen de validation
  - vérifier des propriétés attendues du système à partir de sa spécification
- la preuve comme moyen de vérification
  - vérifier des propriétés “internes” à la spécification
  - vérifier les étapes de conception (construction par raffinement)
  - vérifier le codage par rapport à la conception détaillée

Qui fait de la preuve ?

- Preuve en complément du test

“Cleanroom Software Engineering” (IBM)

  - pas de test unitaire
  - de la preuve “fonctionnelle” avant toute exécution du module
  - du test statistique au niveau du système pour évaluer la fiabilité de l’ensemble

  - Meilleure fiabilité
  - Meilleure productivité

Qui fait de la preuve (suite) ?

- Quelques domaines d'application
  - dans le nucléaire : Merlin-Gérin, CEA, …
  - dans le ferroviaire: RATP (SACEM), GEC-Alsthom, SNCF ?
  - dans l'avionique et l'espace: AIRBUS, CNES, NASA,…
- Soit en conjonction avec des méthodes "maisons", ou des méthodes plus classiques: VDM, Z, B
- Il existe maintenant plusieurs outils avec une applicabilité "industrielle":
  - PVS (Standford)
  - Eve (Canada)
  - B-tools

Les pré-requis de la preuve

- Une spécification formelle de départ (OCL ?)
  - du domaine d'application (propriétés à connaître)
  - des fonctionnalités de l'application: pour chaque fonction il faut expliciter ses pré- et post-conditions
- Une description formelle du langage de programmation
- Des outils de simplification de formules, des procédures de décision
- Des outils de vérification: une preuve peut être fausse…

Nous, on le fera à la main ;-(

La notion de preuve (rappel)

La notion de système formel

- **Langage des formules**: quelles formules peut-on écrire ?
- **Axiomes** (ou schémas d'axiomes): les pré-supposés

sur la logique	*sur le domaine* (ex: les entiers naturels)
$x = x$	$x + 1 \neq 0$
$p \Rightarrow (q \Rightarrow p)$	$x + y = x + z \Rightarrow y = z$
	'+' est associative et commutative

- **Règles d'inférence:** comment combiner des raisonnements

$$\frac{p \Rightarrow q,\ p}{q} \qquad \frac{p \Rightarrow q,\ q \Rightarrow r}{p \Rightarrow r} \qquad \frac{P(0),\quad \forall x: P(x) \Rightarrow P(x+1)}{\forall\, x: P(x)}$$

Un système formel peut être incohérent ;-(

Les preuves n'ont alors plus beaucoup de sens ou d'intérêt...

La notion de preuve (suite)

La notion de dérivation: une formule est un “théorème” ssi

- soit c’est une instance d’un axiome
- soit il existe une séquence de formules t.q.
  - le dernier élément de la séquence est la formule à prouver
  - chaque élément de la séquence est une instance d’un axiome, une des hypothèse ou la conclusion d’une règle d’inférence dont les prémisses apparaissent au préalable dans la séquence

- On part du principe que **l’architecture** de la preuve doit être vérifiable « mécaniquement » : la séquence d’inférences est-elle correcte ? Les transformations de formules sont-elles correctes ?
- **L’intelligence** de la preuve (découverte de lemmes intermédiaires, simplification des formules) reste un problème douloureux…

Preuve de programmes

- Notion de « *correction partielle* » (logique de Hoare)

  preuve sous réserve de terminaison des instructions

- Preuve de « *terminaison* » :
  - associer un “compteur” aux boucles, aux appels récursifs,…
  - montrer que ce compteur est borné inférieurement
  - montrer que la valeur initiale (à l’entrée de la boucle) est supérieure à cette borne
  - montrer que chaque passage dans la boucle ou chaque appel récursif fait strictement décroître la valeur du compteur

On ne traitera pas cette partie par manque de temps !

Preuve complète = correction partielle + terminaison…

Logique de Hoare (1)

- Les instructions sont des “transformateurs de prédicats” :
  une instruction relie la pré-condition supposée vérifiée à la post-condition à assurer en sortie …

Syntaxe : { P } *Instructions* {*Q* }

Sémantique:

Sous réserve que *P* soit vérifié en entrée et que l’exécution de *Instructions* termine, alors *Q* est vérifié en sortie

- A chaque construction du langage de programmation on associe un axiome ou une règle d’inférence.
- La correction partielle d’un programme est prouvée en combinant ces règles et axiomes suivant la structure du programme !
- On annote le programme avec des prédicats intermédiaires (lemmes).

Logique de Hoare: deux triplets particuliers

Deux triplets de Hoare un peu particuliers:

- { True } *P* { Q }

  « Quel que soit l'état initial, si *P* est exécuté et termine, alors Q sera valide dans l'état final »

- { False } *P* { Q }

  « Il n'existe aucun état initial tel que si *P* est exécuté et termine alors Q sera valide dans l'état initial »

Les règles générales

- Axiomes sur les domaines:
  - les règles habituelles sur les entiers, les réels,…
  - les axiomes sur le domaine d'application…
- Axiomes sur la logique:
  - celles vues auparavant…
- Règles d'inférence simples:

Renf-Pré $$\frac{P' \Rightarrow P, \ \{ P \} \ I \ \{ Q \}}{\{ P' \} \ I \ \{ Q \}}$$

Aff-Post $$\frac{\{ P \} \ I \ \{ Q \}, \ Q \Rightarrow Q'}{\{ P \} \ I \ \{ Q' \}}$$

Un langage de programmation simple

Axiome d'affectation simple:

soit *E* une expression simple (pas d'expression indicée, de pointeurs, d'alias,…), et *X* une variable scalaire

$$\{ P[X \backslash E] \} \; X := E \; \{ P(X) \}$$

P[X \ E] ? P dans laquelle on a remplacé syntaxiquement toute occurrence (libre) de X par E.

Ex: { X + Y + 1 + Y = (X + Y + 1) * Z } X := X + Y+ 1 { X + Y= X * Z }

☞ s'applique toujours **de la droite vers la gauche** (à contre-courant)

{ X = Y} Y := 5 { ?? } *De gauche à droite ? Absurde !*

- Passage en séquence:

$$Seq \quad \frac{\{\ P\ \}\ I1\ \{\ Q\ \},\quad \{\ Q\ \}\ I2\ \{\ R\ \}}{\{\ P\ \}\ I1;\ I2\ \{\ R\ \}}$$

- Passage en séquence généralisé :

$$GenSeq \quad \frac{\{\ P_{i-1}\ \}\ I_i\ \{P_i\ \}\ \forall\ i:\ 1 \leq i \leq n}{\{\ P_0\ \}\ I_1;\ I_2;\ \ldots\ I_n\ \{\ P_n\ \}}$$

Exemple de preuve avec AFF et SEQ

$\{ X = b \wedge Y = a \}$

```
X := X + Y ;
Y := X - Y
X := X - Y
```

$\{ X = a \wedge Y = b \}$

Un langage simple (suite)

- if_then_else:

$$ITE \quad \frac{\{ P \wedge B \}\ I1\ \{ Q \},\quad \{ P \wedge \neg B \}\ I2\ \{ Q \}}{\{ P \}\ \text{if } B \text{ then } I1 \text{ else } I2 \text{ end if}\ \{ Q \}}$$

- if_then:

$$IT \quad \frac{\{ P \wedge B \}\ I1\ \{ Q \},\quad P \wedge \neg B \Rightarrow Q}{\{ P \}\ \text{if } B \text{ then } I1 \text{ end if}\ \{ Q \}}$$

Les expressions (arithmétiques et logiques) doivent être traduites du langage de programmation vers le langage de spécification !

Ex: `x <= y` devient $x \leq y$

`x / y` devient la division entière entre `x` et `y`

Question: traduction de `B1 && B2` (Java) ?? De `B1 and B2` (Ada) ??

Attention à respecter la sémantique du langage de programmation !

Un exemple simple de preuve

P: { *Vrai* }

max := x;

Quel prédicat intermédiaire P′ ?

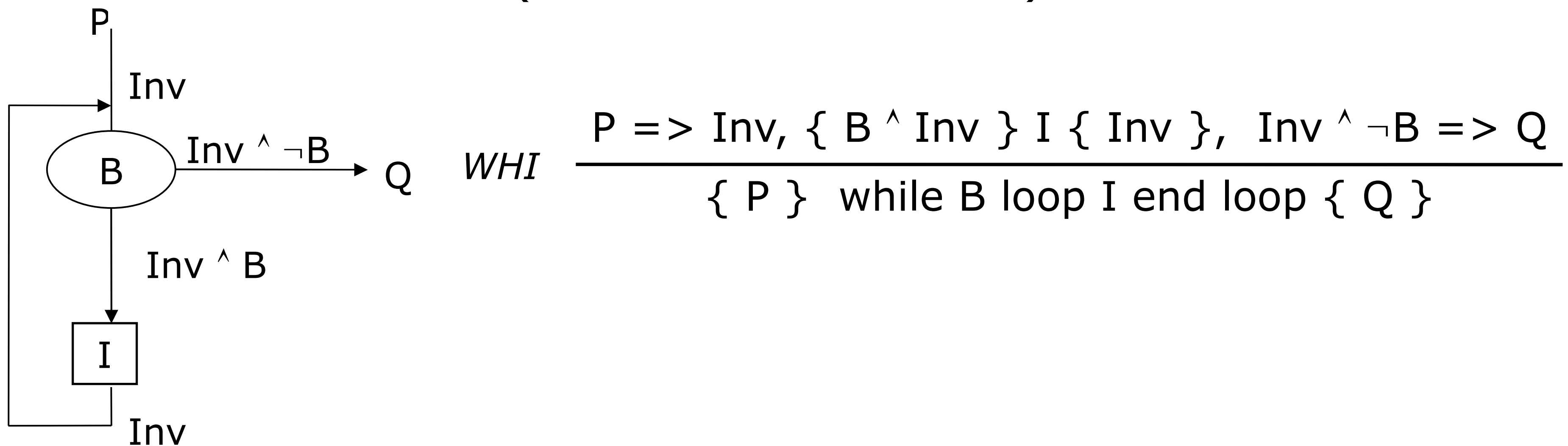
if y > max then max := y; end if;

Q: { $(max \geq x \wedge max \geq y) \wedge (max = x \vee max = y)$ }

- P′ doit être vérifié en cet endroit du programme
- { P } max := x { P′ } doit être prouvable (facilement)
- P′ doit « ressembler » à Q pour faciliter la sous-preuve
  { P′ } if … end if; { Q }

Règle d'inférence pour les boucles

- boucle while (à l'aide d'un invariant):



- boucle while (sans l'aide d'un invariant):

$$WHI\text{-}2 \quad \frac{P \wedge B \{ I \} P}{\{ P \} \text{ while B loop I end loop } \{P \wedge \neg B\}}$$

Un second exemple

P: { $x = i$ }

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

Quel invariant Inv ?

Q: { $S = i\,!$ }

- L'invariant doit être vrai avant l'entrée dans la boucle (et prouvable)
- L'invariant doit être suffisant pour assurer Q (avec $x \leq 1$)
- L'invariant doit être vérifié à **chaque** itération !

- Exprimer les relations qui lient les variables de la boucle à chaque itération !

Un autre exemple avec une boucle

{ X > 0 }

```
Z :=0; M := 0; S := 0;
while Z < X loop
   S := S + M + 1;
   M := M + 2;
   Z := Z + 1;
end loop;
```

{ $S := X^2$ }

Accès à des éléments de tableaux

- La règle d'affectation simple n'est pas valide pour l'affectation à des éléments de tableaux:
  - { a [3] = 2 } `a[i] := 4` { a[3] = 2 } *Manifestement non !*
  - { ?? } `a[a[2]] := 1` { a[a[2]] = 1 }

- On utilise l'axiome supplémentaire suivant :

$$\{ P[A \setminus mod(A,i,E)] \} \;\; A[i] := E \;\; \{ P(X) \}$$

où *mod*(A, i, E) est une **notation** pour le tableau défini par

$$\begin{cases} mod(A, i, E)[i] = E \\ mod(A, i, E)[j] = A[j] \text{ si } i \neq j \end{cases}$$

Ce qui amènera en général à faire un raisonnement par cas….

Tableaux (suite)

- Il faudrait prendre en compte la validité des indices par rapport aux dimensions du tableau

- Si l'indice est lui-même une expression indexée, on décompose l'instruction en utilisant une variable « fraîche » :

  `a[a[2]] := 1` devient $\left\{ \begin{array}{l} \alpha := a[2]; \\ a[\alpha] := 1 \end{array} \right.$

  Dans la première instruction l'accès indexé est en partie droite d'affectation, donc ne pose pas de problème.

- Problèmes similaires pour les pointeurs (identifier la cible) et le passage de paramètres par référence (alias possibles entre variables)

Test et/ou Preuve (1)

Le test

- Compliqué : chemins infaisables; choix des valeurs significatives; validité du résultat obtenu
- Laborieux: volume des tests; choix des valeurs de test; dépouillement
- Nécessite des outils
- Nécessite d’avoir précisément décrit le résultat attendu
- Quand arrête-t-on de tester : sur quels critères ?

Testing shows the presence of errors, not their absence…

Test et/ou Preuve (2)

La preuve

- Compliqué (choix des invariants; précision de la formulation;
  choix des stratégies de preuve)

- Laborieux (taille des formules; niveau de détail)

- Nécessite des outils (simplification, procédures de décision, stratégies de haut niveau)

- Ce n'est qu'une étape dans le processus de vérification:
  Ce qui compte c'est l'ensemble

  logiciel + matériel + support d'exécution

  Prend mal en compte les performances, l'utilisabilité, …

Test et/ou Preuve (3)

- Est-on sûr que le système formel est consistant ?
- Correction: vis-à-vis d'une spécification formelle de départ
  - ☞ cette dernière peut être inconsistante
  - ☞ elle peut ne pas représenter ce qu'on voulait exprimer
- Est-on sûr d'avoir prouvé les "bons" théorèmes ? Une formule, c'est dur à lire et à écrire…

Correction ≠ Robustesse

Test et/ou Preuve (fin)

Deux techniques complémentaires !

- Un continuum : analyse statique / preuve formelle ?
- Ne pas oublier les techniques de validation
- Concevoir en fonction de la vérification
  - programmes “faciles” à tester
  - programmes “faciles” à prouver
  - documentation explicite et précise de la conception
    - garder les choix de conception
    - spécifier les choix de représentation