

# Tests & Preuves

**Michel Rueher**

[http://users.polytech.unice.fr/~rueher/Cours/Test/Cours1TestetPreuve\\_SI4.pdf](http://users.polytech.unice.fr/~rueher/Cours/Test/Cours1TestetPreuve_SI4.pdf)

# Plan du cours

- 1. Le Test**
- 2. Aperçu du BMC (Bounded-model checking)**
- 3. Présentation de Event-B**
- 4. Étude de cas :**

*Toyota Unintended Acceleration and the Big Bowl of “Spaghetti”  
Code*

<http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-“spaghetti”-code>

# Test

- Introduction - Motivations
- Définitions & exemples
- Types de test
- Le test structurel,
- Le test fonctionnel
- Méthodes stochastiques

# 1. Motivation du test

- Le Logiciel ne s'use pas mais ... « bug » !
- Bug célèbres : Ariane V, Missile Patriot, Blocage aéroport,...
- Normes AFNOR ( ?), assurance qualité
- La preuve ne dispense pas de tester !

## Motivation du test (suite)

### *Validation & Vérification du logiciel*

- ⇒ Validation : Est-ce que le logiciel réalise les fonctions attendues ?
- ⇒ Vérification : Est-ce que le logiciel fonctionne correctement ?

### Méthodes de V & V

- Test statique : review de code, de spécifications, de documents de design
- Test dynamique : exécuter le code pour s'assurer d'un fonctionnement correct
- Vérification symbolique : Run-time checking, Execution symbolique, ...
- Vérification formelle : Preuve ou model-checking d'un modèle formel, raffinement et génération de code

Actuellement, le test dynamique est la méthode la plus diffusée et représente jusqu'à 60 % de l'effort complet de développement d'un produit logiciel

## Motivation du test (suite)

### Le logiciel est un objet virtuel difficile à valider :

- Le test et la validation représentent **30 % du coût** en bureautique (**60 %** en ingénierie)
- Il est impossible de réaliser des tests exhaustifs de toutes les fonctions et méthodes écrites pour une application

### Remarques :

Le test a pour objectif la mise en évidence de certaines erreurs, **il n'a pas pour objectif de corriger les fautes**

Il est impossible de créer un test qui puisse effectivement vérifier le bon fonctionnement de l'ensemble d'une application

# Le Test

« *Processus manuel ou automatique qui, par échantillonnage basé sur différents critères de couverture, vise à détecter des différences entre les résultats engendrés par le système et ceux attendus par sa spécification ou encore à établir l'absence de certaines erreurs particulières* »

**Erreur** —→ **Défaut du logiciel** —→ **Anomalie du comportement**  
(développement  
du programme)

**Test** —→ **Détecter les anomalies ou défauts  
(pas une preuve de la correction)**

## Le test est difficile :

- Processus d'introduction des défauts est complexe
- Mal perçu par les programmeurs
- Limites formelles

## Le Test – Définitions

- ⇒ Tester c'est *exécuter* le programme
- ⇒ **Oracle** : prédiction des résultats attendus lors de l'exécution du programme

Deux grandes familles de tests

- Test fonctionnel (ou test **boîte noire**)
  - Test structurel (ou test **boîte de verre**)



## Un petit exemple « Trityp »

Soit la spécification suivante :

*Un programme prend en entrée trois entiers. Ces trois entiers sont interprétés comme représentant les longueurs des cotés d'un triangle. Le programme rend un résultat précisant s'il s'agit d'un triangle scalène, isocèle ou équilatéral.*

## Un petit exemple « Trityp » (suite)

### 14 cas de test – G.J. Myers – «The Art of Software Testing» - 1979

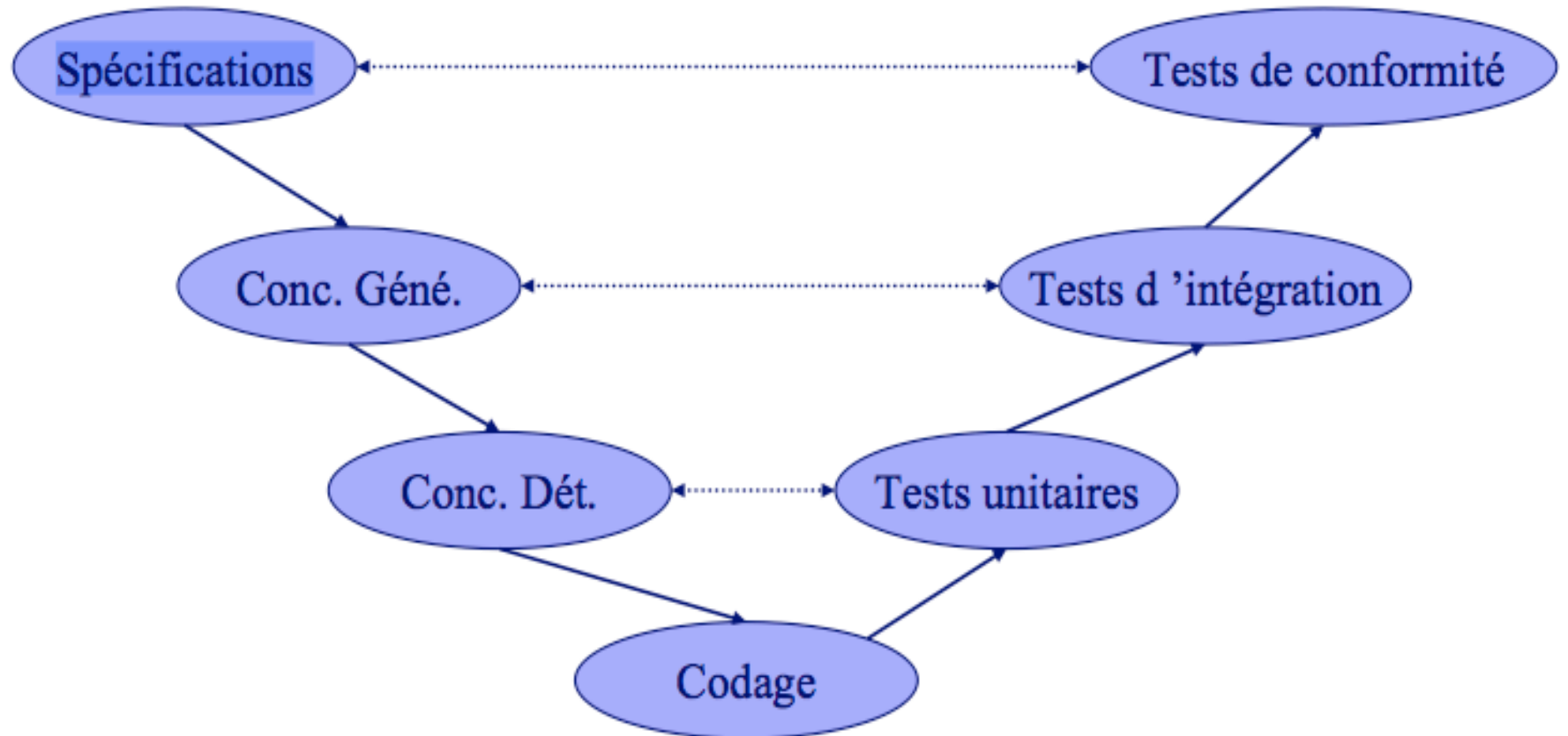
1. Cas scalène valide ( $\langle 1,2,3 \rangle$  et  $\langle 2,5,10 \rangle$  ne sont pas valides)
2. Cas équilatéral valide
3. Cas isocèle valide ( $\langle 2,2,4 \rangle$  n'est pas valide)
4. Cas isocèle valide avec les trois permutations (e.g.  $\langle 3,3,4 \rangle$ ;  $\langle 3,4,3 \rangle$ ;  $\langle 4,3,3 \rangle$ )
5. Cas avec une valeur à 0
6. Cas avec une valeur négative
7. Cas où la somme de deux entrées est égale à la troisième entrée
8. 3 cas pour le test 7 avec les trois permutations
9. Cas où la somme de deux entrées est inférieure à la troisième entrée
10. 3 cas pour le test 9 avec les trois permutations
11. Cas avec les trois entrées à 0
12. Cas avec une entrée non entière
13. Cas avec un nombre erroné de valeur (e.g. 2 entrées, ou 4)
14. Pour chaque cas de test, avez-vous défini le résultat attendu ?

## Un petit exemple « Trityp » (suite)

- Chacun de ces 14 tests correspond à un défaut constaté dans des implantations de l'exemple « Trityp »
- La moyenne des résultats obtenus par un ensemble de développeurs expérimentés est de 7.8 sur 14.

=> La conception de tests est une activité complexe, à fortiori sur de grandes applications

# Test et cycle de vie (1)



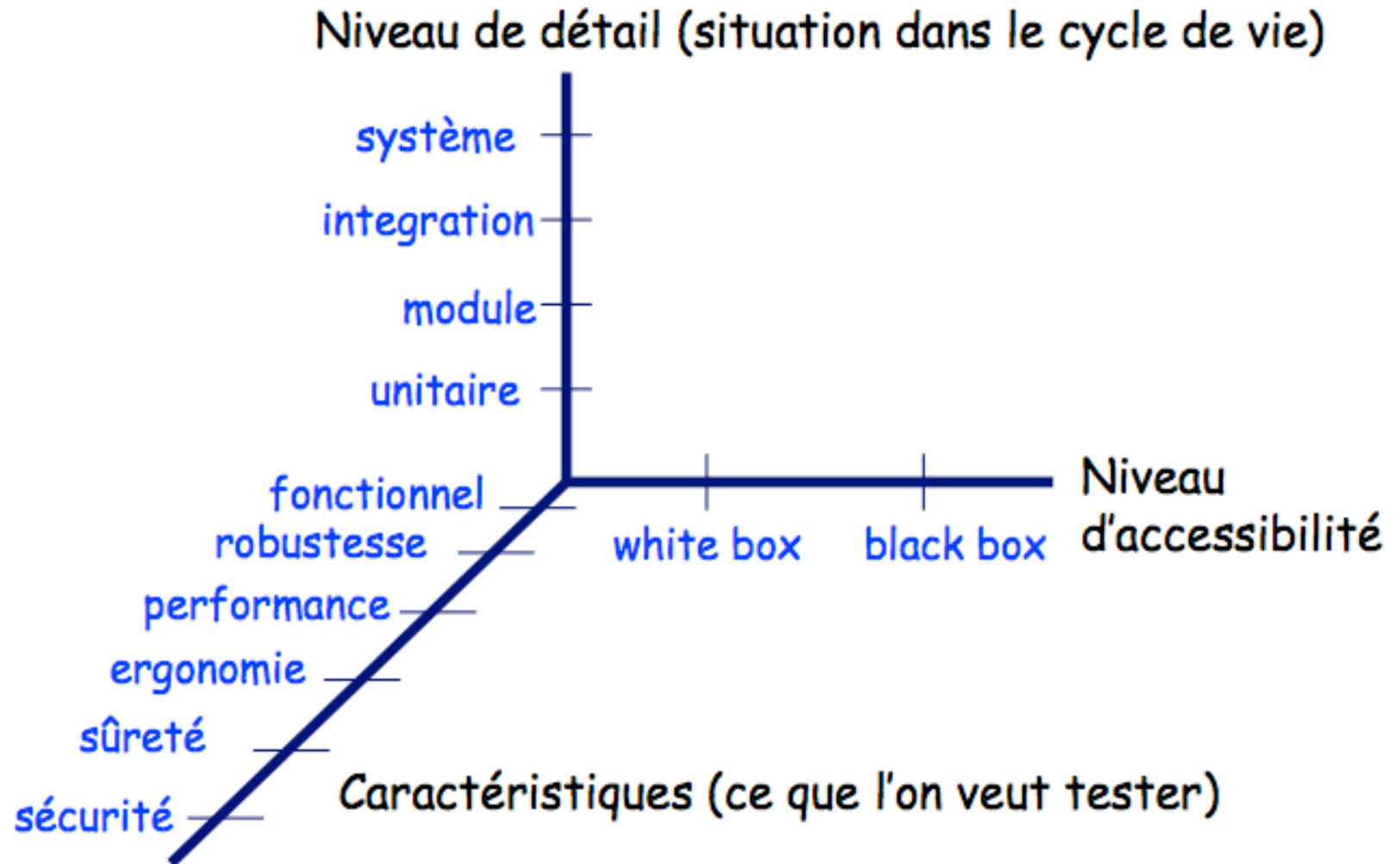
## Différents niveaux de test

- **Le test unitaire** : *Test de procédures, de modules, de composants*  
Ce test contrôle chaque unité logicielle (le plus petit niveau de test) pour savoir si elle correspond à ses spécifications, s'il y a des erreurs de logique
- **Le test d'intégration** : *Test de bon comportement lors de la composition de procédures et modules*  
Ce test cherche à vérifier la stabilité et la cohérence des interfaces et interactions de l'application. Il est réalisé par l'équipe de développement ou une équipe indépendante
- **Le test système** : *Validation de l'adéquation aux spécifications*  
On teste ici la fiabilité et la performance de l'ensemble du système, tant au niveau fonctionnel que structurel, plutôt que de ses composants. On teste aussi la sécurité, les sorties, l'utilisation des ressources et les performances.

## Différents niveaux de test (suite)

- **Le test de non-régression** : *Vérification que les corrections ou évolutions dans le code n'ont pas créées d'anomalies nouvelles*  
Ce test fait suite à une modification de l'application, il faut vérifier que l'application n'a pas perdu de fonctionnalités lors de l'ajout de nouvelles fonctionnalités ou la modification de fonctionnalités existantes.
- **Tests de bon fonctionnement** (*test nominal*): *Les cas de test correspondent à des données d'entrée valides => Test-to-pass*
- **Tests de robustesse** : Les cas de test correspondent à des données d'entrée invalide => Test-to-fail  
Règle : Les tests nominaux sont passés avant les tests de robustesse.
- **Test de performance**
  - Load testing (test avec montée en charge)
  - Stress testing (soumis à des demandes de ressources anormales)

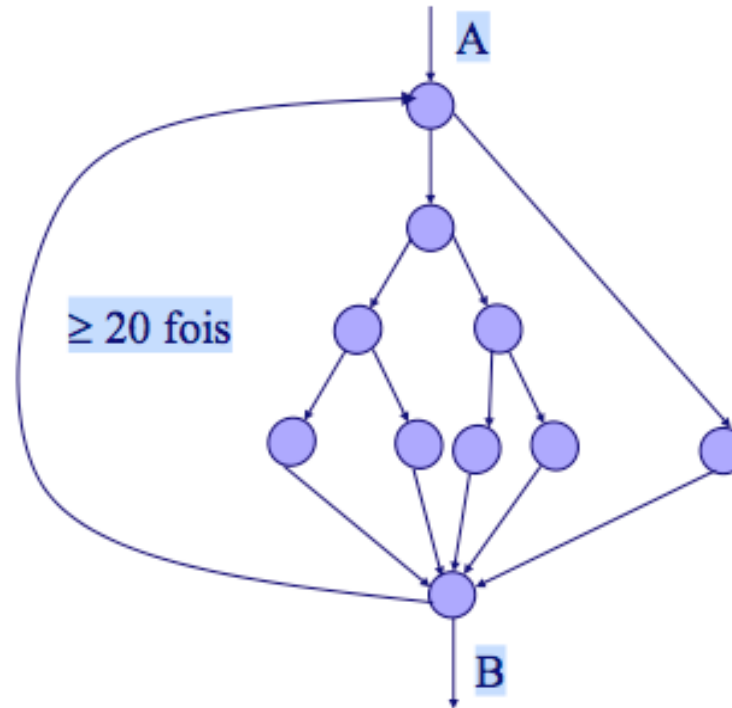
# Type de tests



D'après J. Tretmans – Univ. Nijmegen

## Test structurel (ou boîte de verre) – « White Box Testing »

Les données de test sont produites à partir d'une analyse du code source ou objet  
**Critères de test** : tous les chemins, toutes les branches, toutes les instructions,...





## Test structurel (ou boîte de verre)

- **Test structurel (ou boîte blanche) est basé sur le code du programme**
  - Le test peut être vu comme **un chemin dans le diagramme de flot de contrôle** : tous les détails d'implémentation des composants sont connus
  - Les jeux de test sont choisis de manière à remplir certains critères de couverture (toutes les branches, tous les nœuds, ...)
- **Limites**
  - Impossible de détecter les incomplétudes par rapport à la spécification
  - Lors d'une modification du programme, il est souvent difficile de réutiliser les jeux de test précédents
  - En cas de test à partir du code objet, le GC n'est pas disponible

**Le problème de l'Oracle est un problème critique ...**

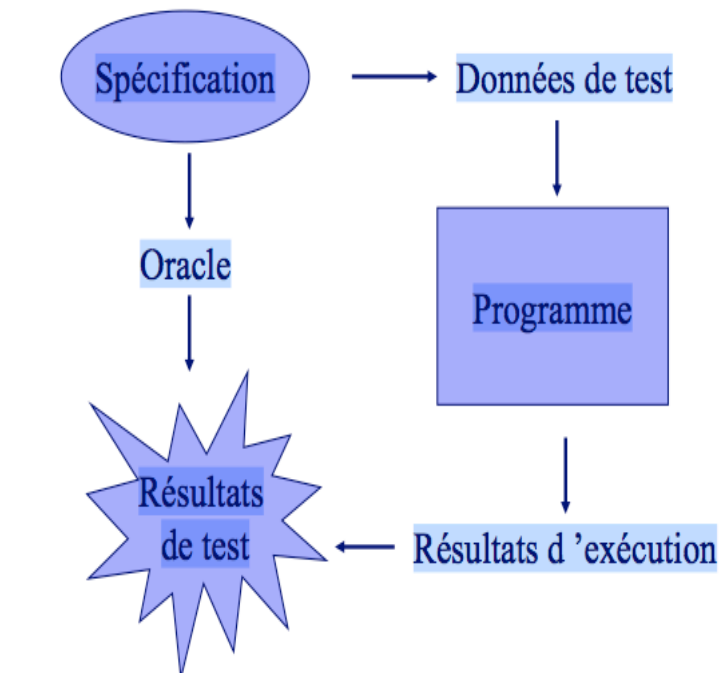
# Graphe de flot de contrôle (GFC)

Graphe orienté et connexe  $(N, A, e, s)$  où

- $N$  : ens. de sommets (bloc d'instructions exécutés en séquence)
- $A$  : relation de  $N \times N$ , débranchements possibles du flot de contrôle
- $e$  : sommet "d'entrée" du programme
- $s$  : sommet de sortie du programme

## Test fonctionnel (ou boîte noire)

Test de conformité par rapport à la spécification



⇒ Seul le **comportement extérieur** est testé : les détails d'implémentation des composants ne sont pas connus

⇒ Directement **dérivés de la spécification** ils permettent de palier aux inconvénients des test structurels

**Le problème de l'Oracle est aussi un problème critique ...**

# Complémentarité des tests fonctionnels et structurels

*Les techniques fonctionnelles et structurelles sont utilisées de façon  
Complémentaire*

Soit le programme suivant censé calculer la somme de deux entiers

```
function sum(x,y : integer) :  
integer;  
begin  
  if (x = 600) and (y = 500) then sum:=  
  x-y  
  else sum:= x+y;  
end
```

Une approche fonctionnelle détectera difficilement le défaut alors qu'une approche par analyse de code pourra produire la DT : **x = 600**, **y = 500**

## Complémentarité des tests fonctionnels et structurel (suite)

En examinant ce qui a été réalisé, on ne prend pas forcément en compte ce qui aurait du être fait :

- Les approches structurelles détectent plus facilement les **erreurs commises**
- Les approches fonctionnelles détectent plus facilement les **erreurs d'omission**

## Difficultés du test (1)

### Le test exhaustif est impossible à réaliser

⇒ En test fonctionnel, l'ensemble des données d'entrée est en général infini ou très grande taille

*Un logiciel avec 5 entrées analogiques sur 8 bits admet  $2^{40}$  valeurs différentes en entrée*

⇒ En test structurel, le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire

*Le nombre de chemins logiques dans le graphe de la figure 1 est supérieur à  $10^{14} \approx 5^{20} + 5^{19} + \dots + 5^1$*

⇒ **Le test est une méthode de vérification partielle de logiciels**

⇒ **La qualité du test dépend de la pertinence du choix des données de test**

## Difficultés du test (2)

### Difficultés d'ordre psychologique ou «culturel»

- **Le test est un processus destructif :**
  - un bon test est un test qui trouve une erreur
  - alors que l'activité de programmation est un processus constructif : on cherche à établir des résultats corrects
  
- **Les erreurs peuvent être dues à des incompréhensions de spécifications ou de mauvais choix d'implantation**
  
- ⇒ **L'activité de test s'inscrit dans l'assurance qualité, indépendante du développement**

## Difficultés du test (3) : problème de l'oracle

Comment vérifier les sorties calculées ?

### **Théorie :**

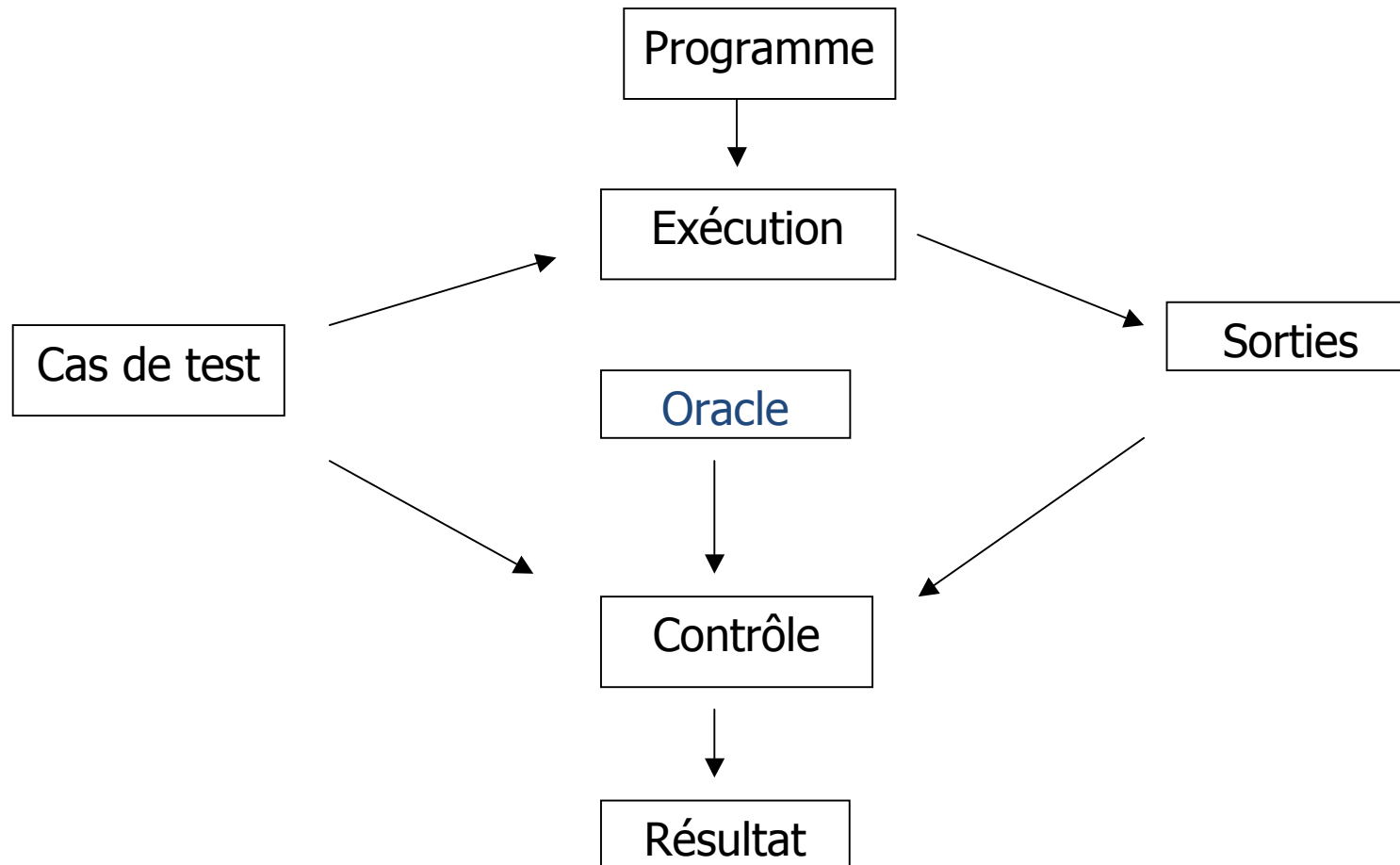
- Par prédiction du résultat attendu
- A l'aide d'une formule issue de la spécification
- A l'aide d'un autre programme

### **Pratique :**

- Prédictions approximatives (à cause des calculs avec des nombres flottants,...)
- Formules inconnues (car programme = formule)
- **Oracle contient des erreurs**

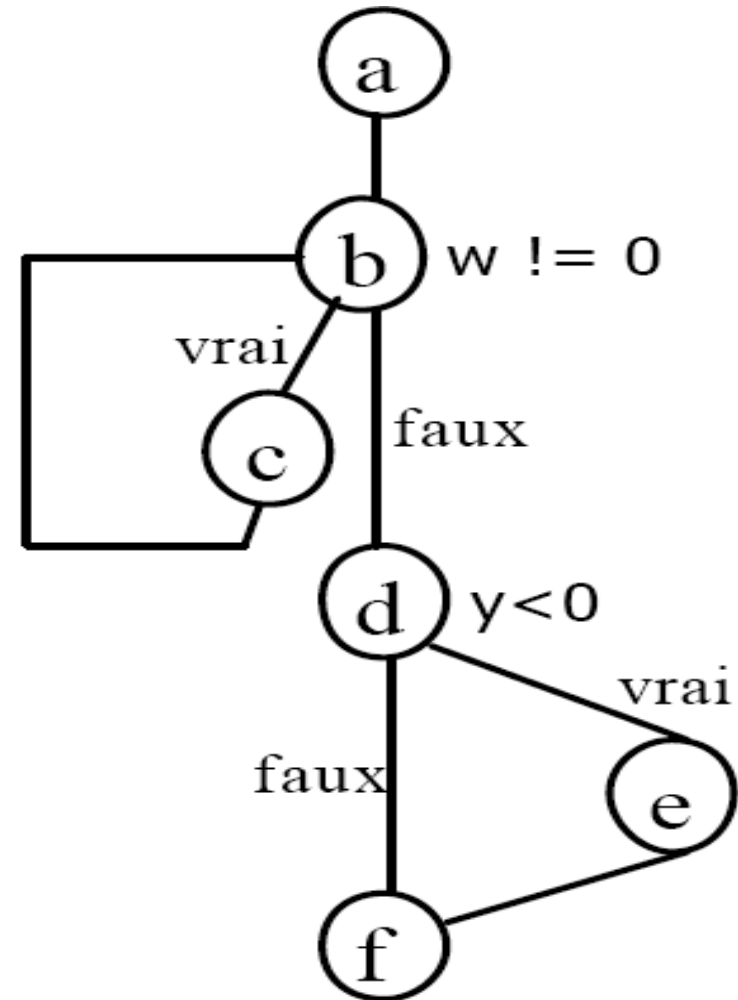


## 2. Le test structurel ou "boîte blanche"


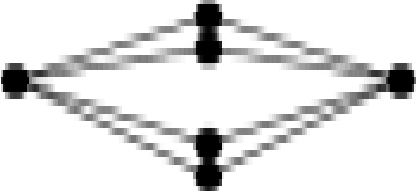
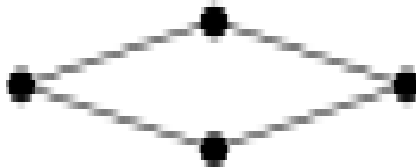
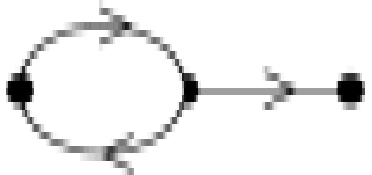
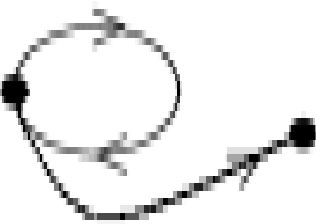
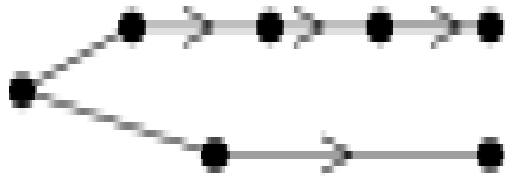


## Graphe de flot de contrôle : exemple

```
double P(short x, short y) {  
    short w = abs(y) ;  
    double z = 1.0 ;  
  
    while ( w != 0 )  
    {  
        z = z * x ;  
        w = w - 1 ;  
    }  
    if ( y < 0 )  
        z = 1.0 / z ;  
    return  
}
```



## Flot de contrôle d'une procédure

<p>➤ Suite linéaire :</p> 	<p>➤ Case :</p> 
<p>➤ If :</p> 	<p>➤ Until :</p> 
<p>➤ While (for) :</p> 	<p>➤ Else :</p> 

# Le test structurel unitaire est obligatoire

## Normes et critères

- Normes de développement
- Contraintes du client
- Stratégie de test
- Plan de test
  
- Toutes\_les\_instructions
- Toutes\_les\_branches, toutes\_les\_décisions
- MC/DC (Multiple Condition/Decision Coverage)
- ...

## Test structurel : les différentes approches

- Les inspections ou walkthrough
- Les critères de couverture du flot de contrôle
- Les critères de couverture du flot de données

# Les inspections ou walkthrough

## ➤ Méthodes manuelles

- 30 à 70% des erreurs détectées le sont de cette manière
- Equipes de 3-4 personnes (un modérateur : convoque la réunion, distribue le matériel, consigne les erreurs)

## ➤ Processus de l'inspection

- Exposé par le programmeur qui commente son code ligne par ligne
- Questions sur les erreurs *standard* :
  - Déclaration des variables
  - Variables initialisées avant utilisation
  - Débordement dans les tableaux
  - Conformités des paramètres (nombre, type) pour les appels de procédures

# Les inspections ou walkthrough (suite)

## ➤ Critères de qualité

- Préparation de la revue (familiarisation avec le code)
- Constitution de l'équipe (mélange de reviewers externes et programmeurs)
- Consignation des erreurs et identification des responsables de leur correction
- Facteurs psychologiques
  - Tensions entre les membres du groupe
  - Durée des réunions
  - Gestion des intérêts antagonistes (défendre son code, trouver des erreurs) par le modérateur

## ➤ Le walkthrough

- Jeux de test établis au préalable
- Les testeurs simulent le comportement du code

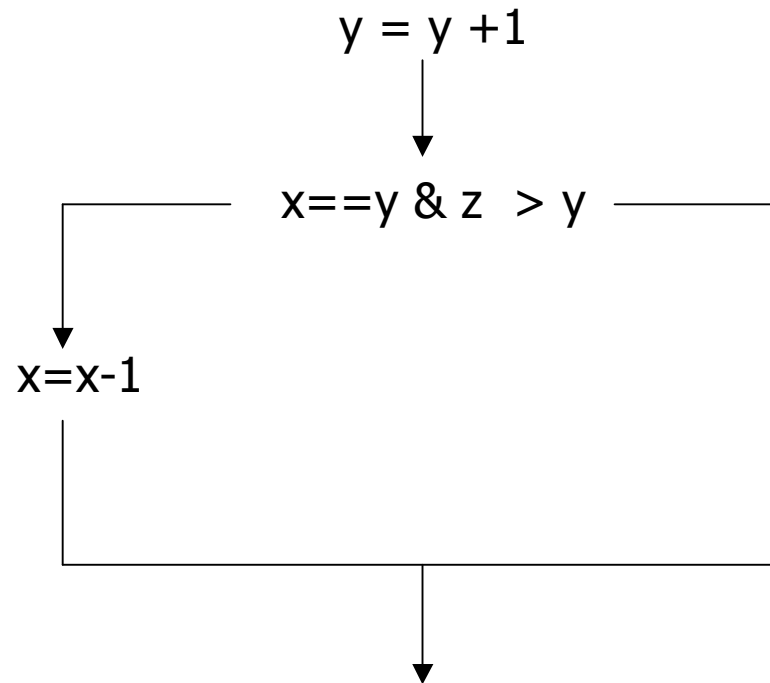
## Les critères de couverture du flot de contrôle

- **Test = chemin dans l'arbre du flot de contrôle**  
(peut être sélectionné en fixant les variables d'entrée)
- La couverture de tous les chemins n'est en général pas possible
- Critères de couverture
  - partitionner les chemins d'exécution en classe d'équivalence



## Couverture des instructions

*Chaque instruction exécutable du programme apparaît au moins une fois dans le cas de test*



Cas de test qui couvre toutes les instructions:  $\{x \leftarrow 2, y \leftarrow 1, z \leftarrow 4\}$

**Remarque : le cas de test ne couvre pas le cas où le prédicat est interprété à faux**

## Couverture des instructions (suite)

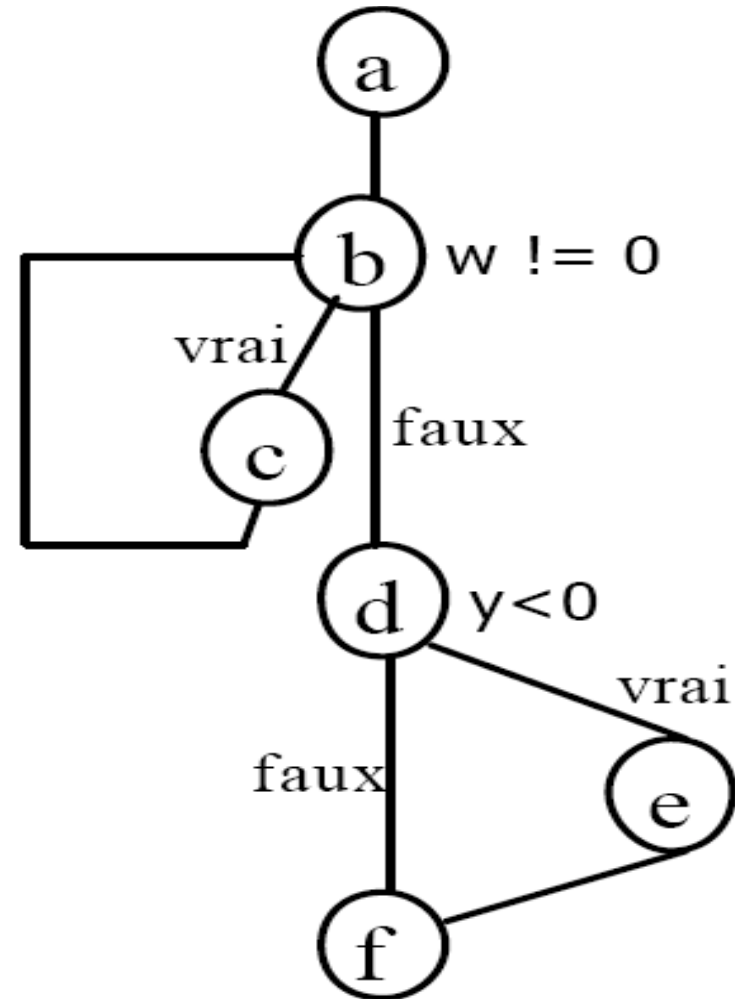
### Définition :

Un ensemble  $C$  de chemins du GFC  $(N, A, e, s)$  satisfait le critère « toutes les instructions » ssi :

$$\forall n \in N, \exists C_i \in C : n \text{ est un sommet de } C_i$$

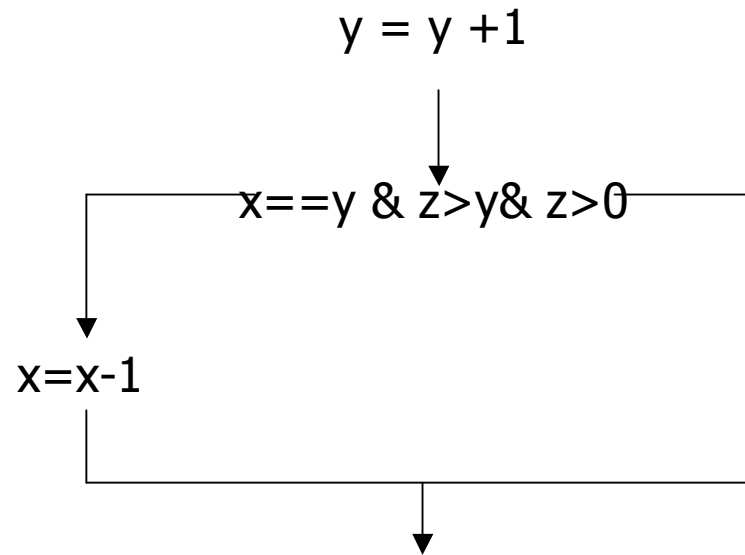
( $\equiv$  **tous\_les\_sommets**)

**Exemple** : un seul chemin suffit  
a-b-c-b-d-e-f [6/6 sommets]



## Couverture des décisions

*Chaque décision exécutable du graphe de flot de contrôle apparaît au moins une fois à vrai et une fois à faux dans le cas de test*



Cas de test qui couvrent toutes les décisions:  $\{x \leftarrow 2, y \leftarrow 1, z \leftarrow 4\}$   
 $\{x \leftarrow 3, y \leftarrow 2, z \leftarrow 2\}$

## Couverture des décisions (suite)

### Définition :

Un ensemble  $C$  de chemins du GFC  $(N,A,e,s)$  satisfait le critère « toutes les décisions » ssi :

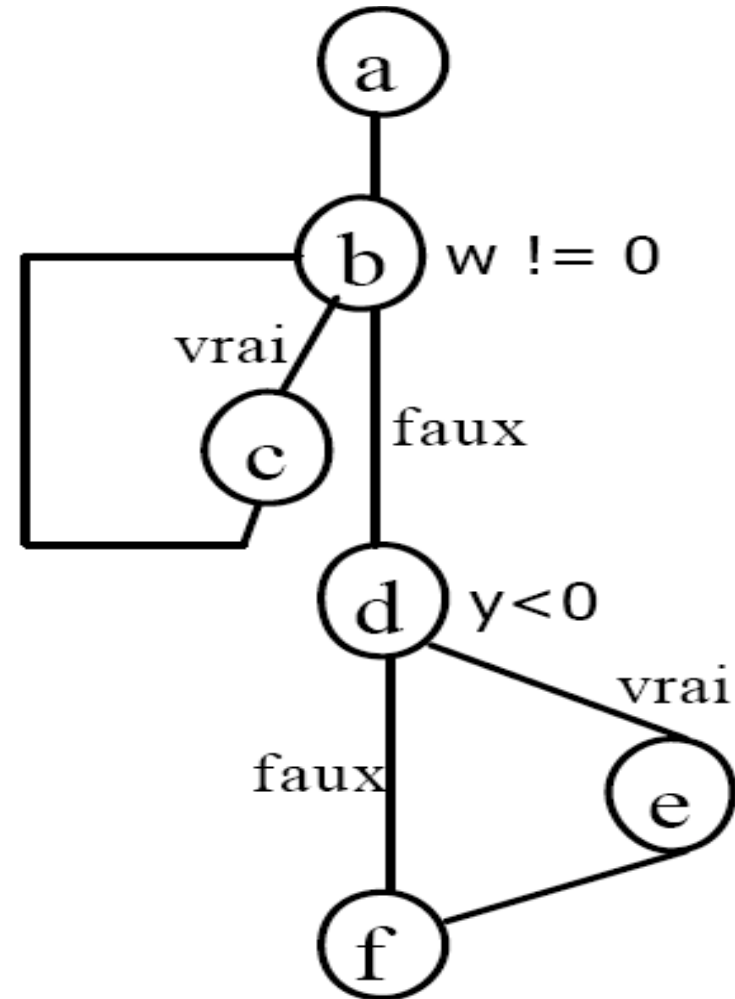
$\forall a \in A, \exists C_i \in C : a$  est un arc de  $C_i$

( $\equiv$  **tous les arcs**)

**Exemple** : deux chemins sont nécessaires

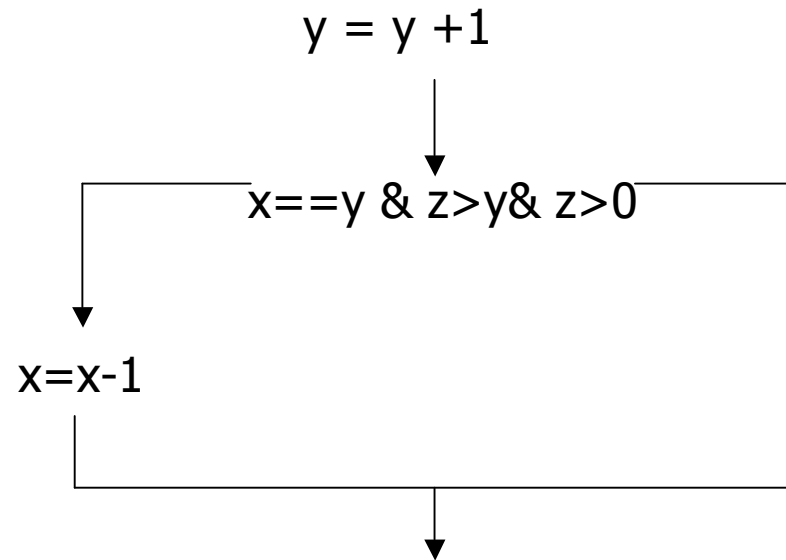
a-b-c-b-d-e-f [6/7 arcs]

a-b-d-f [3/7 arcs]



## Couverture des conditions

*Chaque prédicat atomique du programme apparaît au moins une fois à vrai et une fois à faux dans le cas de test*



Cas de test qui couvrent toutes les conditions:  $\{x \leftarrow 2, y \leftarrow 1, z \leftarrow -1\}$   
 $\{x \leftarrow 3, y \leftarrow 3, z \leftarrow 5\}$

## Couverture des conditions/décisions modifiées

**Objectif** : Démontrer l'action de chaque condition sur la valeur de vérité de la décision

**Principe** : pour chaque condition, trouvez 2 cas de test qui changent la décision lorsque toutes les autres conditions sont fixées

**Exemple** :

```
if( A && ( B || C ) )
```

pour A	A=0, B=1,C=1	-- Décision =0
	A=1, B=1,C=1	-- Décision =1
pour B	A=1, B=1,C=0	-- Décision =1
	A=1, B=0,C=0	-- Décision =0
pour C	A=1, B=0,C=1	-- Décision =1
	<del>A=1, B=0,C=0</del>	<del>-- Décision =0</del>

## Couverture des boucles

Il existe plusieurs stratégies ad hoc pour la couverture des boucles :

- La couverture des ***i*-chemins**

Pour chaque boucle au moins tous les chemins contenant  $j$  répétitions de la boucle, pour  $0 \leq j \leq i$  (*bounded testing*)

- Si la borne  $n$  est connue (itération), tester pour  $n-1$ ,  $n$  et  $n+1$  itérations

# Couverture des Parties Linéaires de Codes suivies d'un Saut (PLCS)

**Motivation:** La distance (en termes d'effort) entre la couverture des branches et la couverture des chemins est (trop) importante -> Nécessité de critères intermédiaires

**LCSAJ:** triplet correspondant à un «sous-chemin» :

- **instruction de départ** : 1<sup>ère</sup> du programme ou destination du branchement
- **instruction de fin** : fin du programme ou instruction contenant un branchement (la partie précédant le branchement est exécutée)
- **instruction de destination** : destination d'un branchement

**Critères** : TER1 (instructions exécutées), TER2 (branches exécutées), ...



## Couverture des chemins

Il existe plusieurs critères de couverture des chemins:

- La couverture de tous les **chemins simples**
- La couverture de tous les **chemins**

La couverture de tous les chemins est en général impossible :

- En cas de boucles, **il existe une infinité de chemins**
- Il existe de nombreux **chemins non exécutables**

(déterminer si un chemin est exécutable est indécidable : réduction au problème de l'arrêt d'une machine de Turing)

## Couverture de tous les chemins simples

### Principe:

Couvrir tous les chemins d'exécution sans itérer plus d'une fois dans les boucles

( $\equiv$  **1-chemins**)

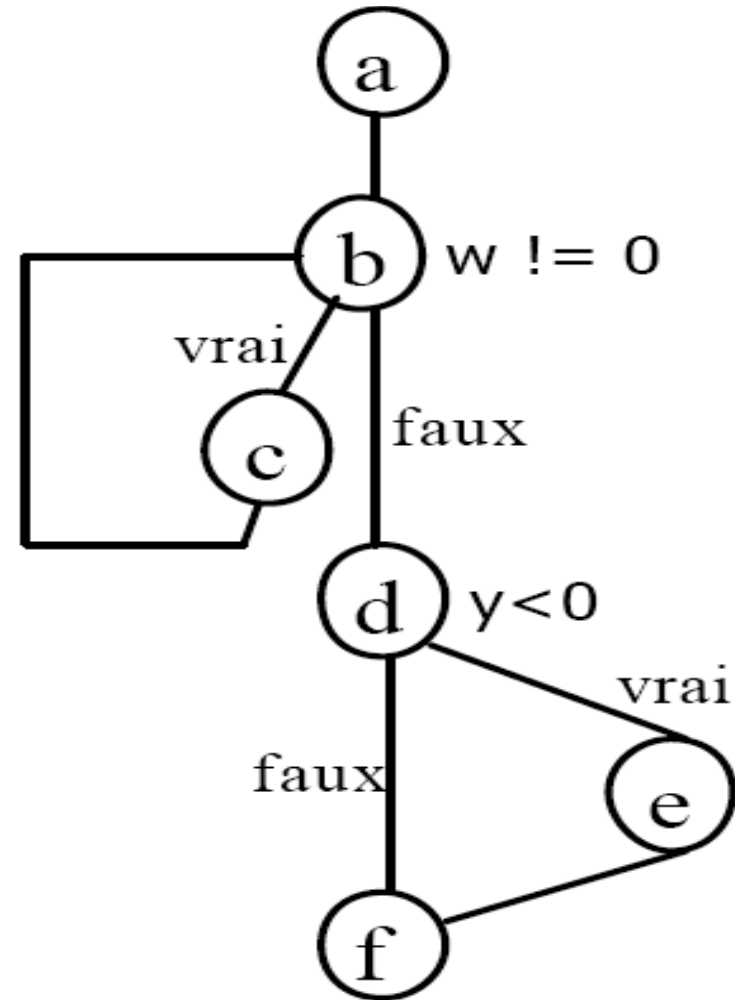
**Exemple** : quatre chemins sont nécessaires

a-b-d-f

a-b-d-e-f

a-b-c-b-d-f

a-b-c-b-d-e-f



# Les critères de couverture du flot de données

## Objectif

Minimiser le nombre de cas de test tout en préservant une probabilité élevée de découvrir des erreurs comme les variables non initialisées, les variables affectées mais non utilisées

## Outil

### Graphe Def/Use [GDU]

= GFC + décoration des sommets/arcs avec des informations sur les définitions/utilisations sur les variables

# Couverture du flot de données : graphe GDU

- **All-defs**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et une de ses utilisations
- **All-p-uses**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et chacune des utilisations de cette variable *dans un prédicat* (condition)
- **All-c-uses**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et chacune des utilisations de cette variable *dans un calcul*
- **All-c-uses-some-p-uses**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et chacune des utilisations de cette variable *dans un calcul, à défaut dans un prédicat*
- **All-p-uses-some-c-uses**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et chacune des utilisations de cette variable *dans un prédicat, à défaut dans un calcul*
- **All-uses**
  - ◆ Exécution d'au moins un chemin sans définition entre chaque définition de variable et chacune des utilisations de cette variable (*dans un calcul ou dans un prédicat*)
- **All-du-paths**
  - ◆ Exécution *de tous les chemins* sans définition entre chaque définition de variable et chacune des utilisations de cette variable (*dans un calcul ou dans un prédicat*)

### 3. Le test "boîte noire" ou test fonctionnel

#### Basé sur la spécification ou l'interface du programme P

On peut appliquer le critère de couverture complète du domaine  
Le programme P est vu comme une fonction des données D dans les résultats R:

$$P : D \rightarrow R$$

On partitionne D en classes  $D_i$

- $D = D_0 \cup D_1 \cup \dots \cup D_{n-1} \cup D_n$
- $\forall i, j < n \ i \neq j \Rightarrow D_i \cap D_j = \emptyset$
- On sélectionne un seul cas de test par classe  $D_i$

#### Hypothèse de test:

$\forall i \ \forall d, d' \in D_i \ P(d)$  est correct si et seulement si  $P(d')$  est correct

## Approche empirique: exemple 1

Soit le programme qui calcule  $F(X) = \text{sqrt}(1/x)$

1<sup>ère</sup> Classe :  $x < 0$   
2<sup>ème</sup> Classe :  $x = 0$   
3<sup>ème</sup> Classe :  $x > 0$

**Remarque** : les partitions sont un peu plus délicates à définir lorsque le calcul s'effectue sur les flottants

**Voir** : What Every Computer Scientist Should Know About Floating-Point Arithmetic, by David Goldberg

[http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html)

## Approche empirique: exemple 2

Si la valeur  $n$  de l'entrée est  $<0$ , alors un message d'erreur est imprimé.

Si  $0 \leq n < 20$ , alors la valeur exacte de  $n!$  est imprimée.

Si  $20 \leq n \leq 200$ , alors une approximation de  $n!$  est imprimée en virgule flottante avec une précision de 0.1%.

Si  $n > 200$ , un message d'erreur est imprimé.

# Solution

## Les classes

$$D_1 = \{n : n < 0\}$$

$$D_2 = \{n : 0 \leq n < 20\}$$

$$D_3 = \{n : 20 \leq n \leq 200\}$$

$$D_4 = \{n : n > 200\}$$

forment une partition des entiers

**Le jeu  $\{-10, 6, 75, 300\}$  satisfait le critère de couverture totale**



## Approche empirique: l'exemple du triangle

**Entrée:** 3 entiers  $> 0$  (longueurs de cotés)

**Sortie:** type de triangle déterminé par les 3 entiers (équilatéral, isocèle, scalène, autre)

$(a,b,c) \longrightarrow \text{Triangle ?} \longrightarrow \text{Type de triangle ?}$

**Hypothèse :** On peut identifier la classe de D par sa valeur  $P(D)$

$D_1 = \{(a, b, c) : abc \text{ est équilatéral}\}$

$D_2 = \{(a, b, c) : abc \text{ est isocèle}\}$

$D_3 = \{(a, b, c) : abc \text{ est scalène}\}$

$D_4 = \{(a, b, c) : abc \text{ n'est pas un triangle}\}$

## Solution plus raffinée

$D_1 = \{(a, b, c): \text{est équilatéral}\}$

$D_{21} = \{(a, b, c): \text{est isocèle et } a=b, a \neq c\}$

$D_{22} = \{(a, b, c): \text{est isocèle et } a=c, b \neq c\}$

$D_{23} = \{(a, b, c): \text{est isocèle et } b=c, a \neq b\}$

$D_3 = \{(a, b, c): \text{est scalène, c'est à dire } a \neq b \neq c\}$

$D_{41} = \{(a, b, c): \text{pas un triangle et } a \geq b+c\}$

$D_{42} = \{(a, b, c): \text{pas un triangle et } b \geq a+c\}$

$D_{43} = \{(a, b, c): \text{pas un triangle et } c \geq a+b\}$

### Exemple de jeu de test :

1. (3, 3, 3)

5. (3, 4, 5)

2. (2, 2, 3)

6. (6, 3, 3)

3. (2, 3, 2)

7. (3, 6, 3)

4. (3, 2, 2)

8. (3, 3, 6)

## Test aux limites

Pour chaque variable, choix de valeurs « critiques » de son domaine, e.g.,

- **Si la variable est comprise dans un intervalle de valeurs**, on choisit la plus petite variation delta, et on sélectionne les 6 valeurs suivantes :
  - Les 2 valeurs correspondant aux **2 limites** du domaine
  - 4 valeurs correspondant aux valeurs des **limites +/- delta**
- Si la variable appartient à un ensemble ordonné de valeurs, on choisit le **premier, le second, l'avant dernier et le dernier élément**

....

## Test aux limites : exemple du triangle

<b>Données de test aux limites</b>	<b>Remarques</b>
1, 1, 2	Non-triangle
0, 0, 0	Un seul point
4, 0, 3	Une des longueurs est nulle
1, 2, 3.00001	Presque triangle
0.001, 0.001, 0.001	Très petit triangle
88888, 88888, 88888	Très grand triangle
3.0001, 3, 3	Presque équilatéral
2.9999, 3, 4	Presque isocèle
3, 4, 5.00001	Presque droit
3, 4, 5, 6	Quatre données
3	Une seule donnée
3.14.5, 6, 7	Deux points
5, 5, A	Caractère au lieu de chiffre
	Entrée vide
-3, -3, 5	Longueurs négatives
3,, 4, 5	Deux virgules

# Grammaire BNF

Une **grammaire** est un ensemble de règles qui définit toutes les constructions valides d'un langage.

Une grammaire est formée:

1. Un ensemble de symboles **terminaux**
2. Un ensemble de symboles **non-terminaux**
3. Un ensemble de **productions** de la forme  $x:=y$   
où  $x$  est un symbole non-terminal et  $y$  une chaîne quelconque de symboles  
(terminaux ou non).
4. Un symbole **initial** (non-terminal distingué)

## Langage d'une calculatrice

Terminaux: 0, 1, 2, ..., 9, +, -, \*, /, =, .

**Non-terminaux:** calcul, expression, value, number, unsigned, digit, sign, operator

### Productions:

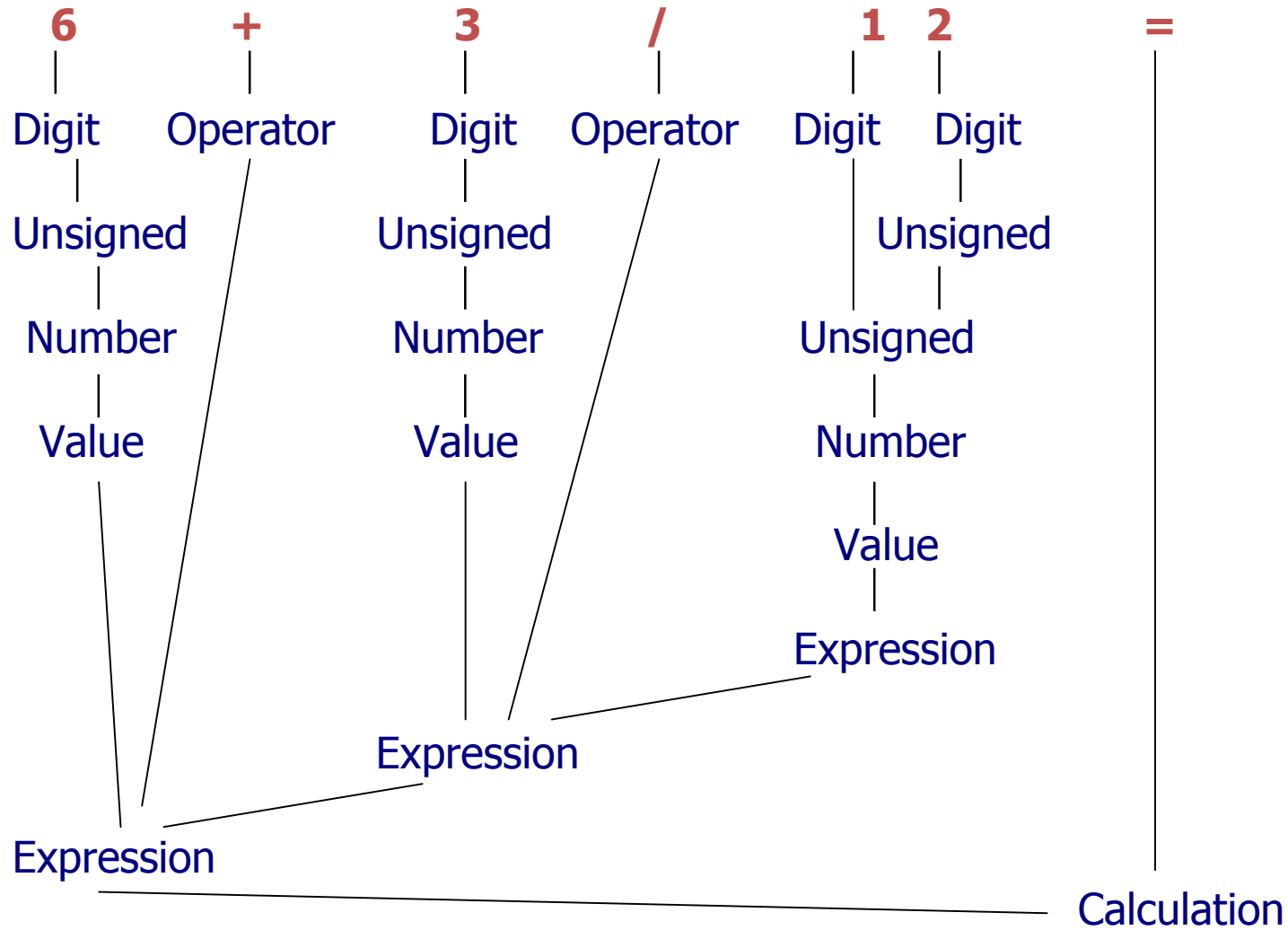
1.  $\langle \text{calcul} \rangle := \langle \text{expression} \rangle =$
2.  $\langle \text{expression} \rangle := \langle \text{value} \rangle \mid \langle \text{value} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle$
3.  $\langle \text{value} \rangle := \langle \text{number} \rangle \mid \langle \text{sign} \rangle \langle \text{number} \rangle$
4.  $\langle \text{number} \rangle := \langle \text{unsigned} \rangle \mid \langle \text{unsigned} \rangle . \langle \text{unsigned} \rangle$
5.  $\langle \text{unsigned} \rangle := \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{unsigned} \rangle$
6.  $\langle \text{digit} \rangle := 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
7.  $\langle \text{sign} \rangle := + \mid -$
8.  $\langle \text{operator} \rangle := + \mid - \mid * \mid /$

## Langage d'une calculatrice (suite)

**Critère de test:** couverture complète des règles à partir du symbole initial :

- Pour chaque règle, générer une entrée valide en partant du symbole initial, telle que la règle est appliquée
- **Procédure:** Dérivation descendante (top-down)
- S'assurer que tous les terminaux apparaissent dans au moins une règle.
- **Utilisation typique:** validation de données

# Génération de tests: exemple





## 4 Méthodes stochastiques de génération de tests

### Principes :

- **Loi de probabilité uniforme** sur les domaines d'entrée  
→ Utilisation de générateurs de valeurs pseudo aléatoires
- **Disponibilité d'un oracle** pour vérifier automatiquement les résultats (e.g. spécification formelle « exécutable »)
- Condition d'arrêt : pourcentage de couverture, nombre de tests générés

### Avantages et limites

- Génération simplifiée
- **Certains cas sont très difficiles à atteindre** → introduction de biais