

Mettre en œuvre une démarche de test en développement logiciel

Fabrice AMBERT – fabrice.ambert@femto-st.fr
Fabrice BOUQUET – fabrice.bouquet@femto-st.fr
Fabien PEUREUX – fabien.peureux@femto-st.fr
Ivan ENDERLIN, Jean-Marie GAUTHIER
Cédric JOFFROY, Alexandre VERNOTTE

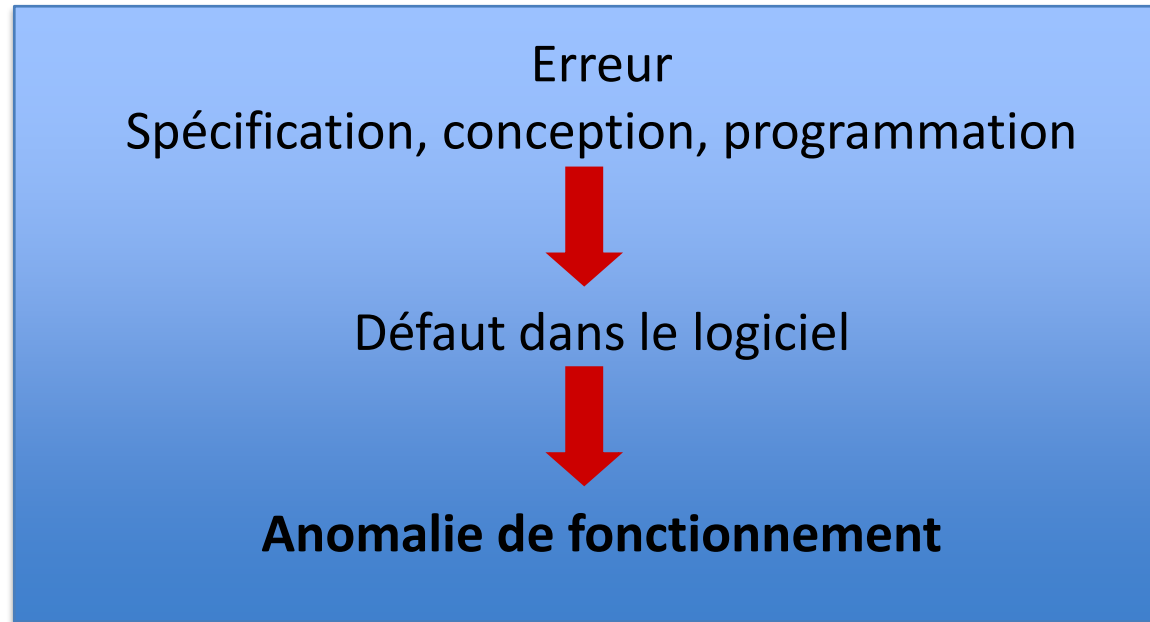
Plan de la présentation



- Introduction au test logiciel
- Test structurel
- Test fonctionnel
- Calcul des données de test
- Synthèse

Motivations du test

- Complexité croissante des architectures et des comportements
- Coût d'un bug (Ariane 5, carte à puces allemande bloquée, prime de la PAC...)



- Coût des bugs informatiques : ≈ 60 milliards \$ / an
- 22 milliards économisés si les procédures de test de logiciels étaient améliorées.

(source : NIST - National Institute of Standards and Technology)



Validation & Vérification

- V & V
 - Validation : Est-ce que le logiciel offre les services attendues ?
 - Vérification : Est-ce que les artefacts utilisés sont corrects ?
- Méthodes de V & V
 - Test statique : Revue 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

Définition du test

- « Le test est *l'exécution ou l'évaluation d'un système* ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il *répond à ses spécifications* ou *identifier les différences entre les résultats attendus et les résultats obtenus*. »

IEEE (Standard Glossary of Software Engineering Terminology)

- « Tester, c'est exécuter le programme *dans l'intention d'y trouver des anomalies ou des défauts*. »

G. Myers (The Art of Software testing)

- « *Testing can reveal the presence of errors but never their absence.* » Edsger W. Dijkstra (Notes on Structured Programming)

La pratique du test (1)



- Le test appartient à l'activité de Validation du logiciel :
est-ce-que le logiciel fait les choses bien et les bonnes choses ?
- Activité historiquement peu populaire en entreprise
- Difficultés d'ordre psychologique ou « culturel » :
 - L'activité de programmation est un processus constructif : on cherche à établir des résultats corrects
 - Le test est un processus destructif : un bon test est un test qui trouve une erreur

Méthodes de test



- **Test statique**

Traite le code du logiciel sans l'exécuter sur des données réelles.

- **Test dynamique**

Repose sur l'exécution effective du logiciel pour un sous ensemble bien choisi du domaine de ses entrées possibles.

La réalité du test



- Le test constitue aujourd'hui le vecteur principal de l'amélioration de la qualité du logiciel
- Actuellement, le test dynamique est la méthode la plus diffusée
- Il peut représenter jusqu'à 60 % de l'effort complet de développement d'un logiciel
- Coût moyen de l'activité de test :
 - 1/3 durant le développement du logiciel
 - 2/3 durant la maintenance du logiciel



Test statique

- Exemples :
 - *Lectures croisées / inspections*
Vérification collégiale d'un document (programme ou spécification du logiciel)
 - *Analyse d'anomalies*
Corriger de manière statique les erreurs (typage impropre, code mort, ...)
- Avantages :
 - Méthodes efficaces et peu coûteuses
 - 60% à 95% des erreurs sont détectées lors de contrôles statiques
- Inconvénients :
 - Ne permet pas de valider le comportement d'un programme au cours de son exécution

→ Les méthodes de test statiques sont **nécessaires**, mais **pas suffisantes**



Test dynamique - niveaux

- Repose sur l'exécution du programme à tester
- 4 niveaux complémentaires :
 - Test de composants (unitaire)
 - Test d'intégration des composants
 - Test du système global
 - Test d'acceptation (recette)



Test dynamique - techniques

- Deux techniques :
 - **Test structurel**
Jeu de test sélectionné en s'appuyant sur une analyse du code source du logiciel (*test boîte blanche / boîte de verre*)
 - **Test fonctionnel**
Jeu de test sélectionné en s'appuyant sur les spécifications (*test boîte noire*)
- En résumé, les méthodes de test dynamique consistent à :
 - Exécuter le programme sur un ensemble fini de données d'entrées
 - Contrôler la correction des valeurs de sortie en fonction de ce qui est attendu

Test de logiciels – auto-évaluation

L'exemple du triangle

- 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 (ou une erreur si les données ne correspondent pas aux longueurs d'un triangle).

G. Myers (The Art of Software testing - 1979)

- Donnez un ensemble de cas de test que vous pensez adéquat pour tester pour ce programme...

Test de logiciels – auto-évaluation

L'exemple du triangle

- Avez-vous un cas de test pour :
 1. Cas scalène valide (1,2,3 et 2,5,10 ne sont pas valides)
 2. Cas équilatéral valide
 3. Cas isocèle valide (2,2,4 n'est pas valide)
 4. Cas isocèle valide avec les trois permutations (e.g. 3,3,4; 3,4,3; 4,3,3)
 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. Trois cas pour le test 7 avec les trois permutations
 9. Cas où la somme de deux entrées est inférieure à la troisième entrée
 10. Trois 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 valeurs (e.g. 2 entrées, ou 4)
 14. Pour chaque cas de test, avez-vous défini le résultat attendu ?

Test de logiciels – auto-évaluation

L'exemple du triangle

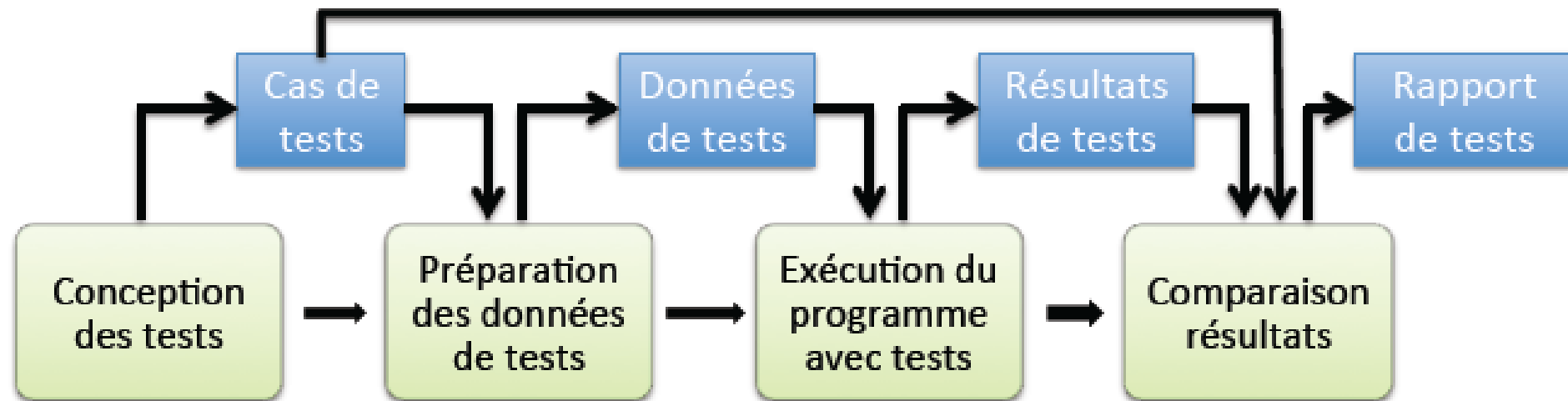
- Chacun de ces 14 tests correspond à un défaut constaté dans des implantations de cet exemple du triangle
 - La moyenne des résultats obtenus par un ensemble de développeurs expérimentés est de 7.8 sur 14.
- => Le test est une activité complexe, à fortiori sur de grandes applications



Test dynamique – 4 activités

- Sélection d'un jeu de tests : choisir un sous-ensemble des entrées possibles du logiciel
- Soumission du jeu de tests
- Dépouillement des résultats : consiste à décider du succès ou de l'échec du jeu de test (*verdict*):
 - Fail, Pass, Inconclusive
- Évaluation de la qualité et de la pertinence des tests effectués (déterminant pour la décision d'arrêt de la phase de test)

Test et cycle de vie du processus de test



Test structurel (white box)

- Les données de test sont produites à partir d'une analyse du code source

Critères de test :

- tous les chemins,
- toutes les instructions,
- etc...

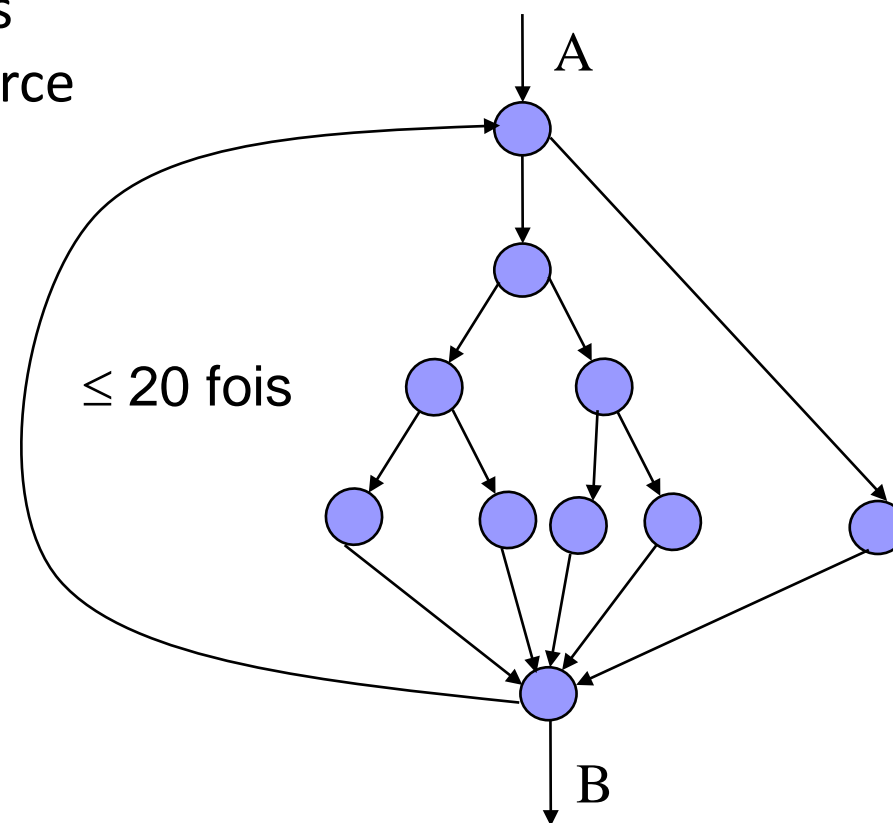
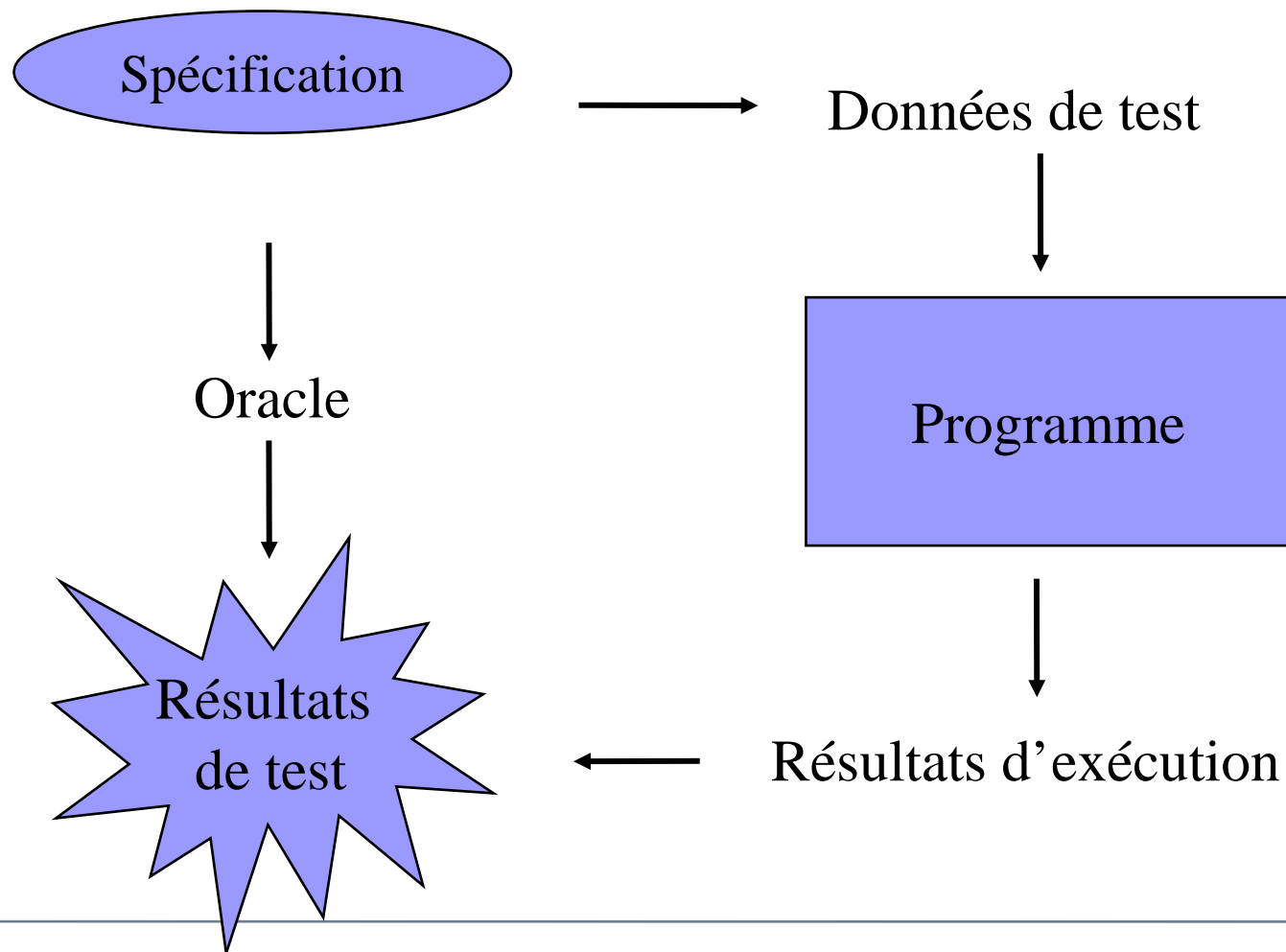


Fig. 1 : Flot de contrôle d'un petit programme

Test fonctionnel (black-box)

- Test de conformité par rapport à la spécification





Complémentarité (1)

test fonctionnel / structurel

- Les 2 approches sont utilisées de façon complémentaire
- Exemple : soit le programme suivant, censé calculer la somme de 2 entiers :

```
function sum (x,y : integer) : integer;  
  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 donnée de test : $x = 600$, $y = 500$.



Complémentarité (2)

test fonctionnel / structurel

- 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 dans le programme
 - Les approches fonctionnelles détectent plus facilement les erreurs d'omission et de spécification
- Une difficulté du test structurel consiste dans la définition de l'oracle de test.

Difficultés du test

- Le test exhaustif est en général impossible à réaliser
 - En test structurel, le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire

Exemple : le nombre de chemin logique dans le graphe de la figure 1 est supérieur à $10^{14} \approx 5^{20} + 5^{19} + \dots + 5^1$

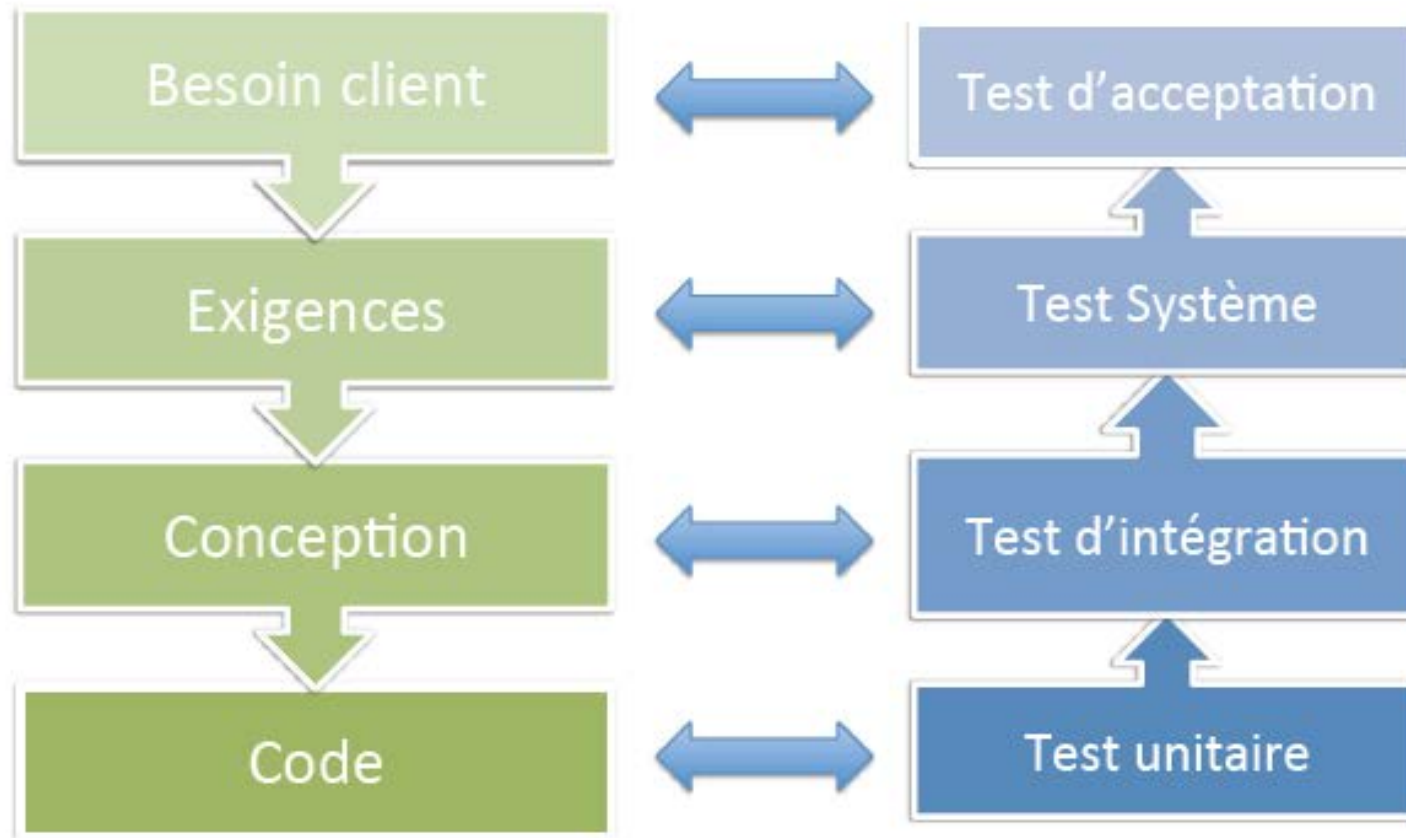
- En test fonctionnel, l'ensemble des données d'entrée est en général infini ou très grande taille

Exemple : un logiciel avec 5 entrées analogiques sur 8 bits admet 2^{40} valeurs différentes en entrée

=> le test est une méthode de validation partielle de logiciels

=> la qualité du test dépend de la pertinence du choix des données de test

Développement et niveaux de tests

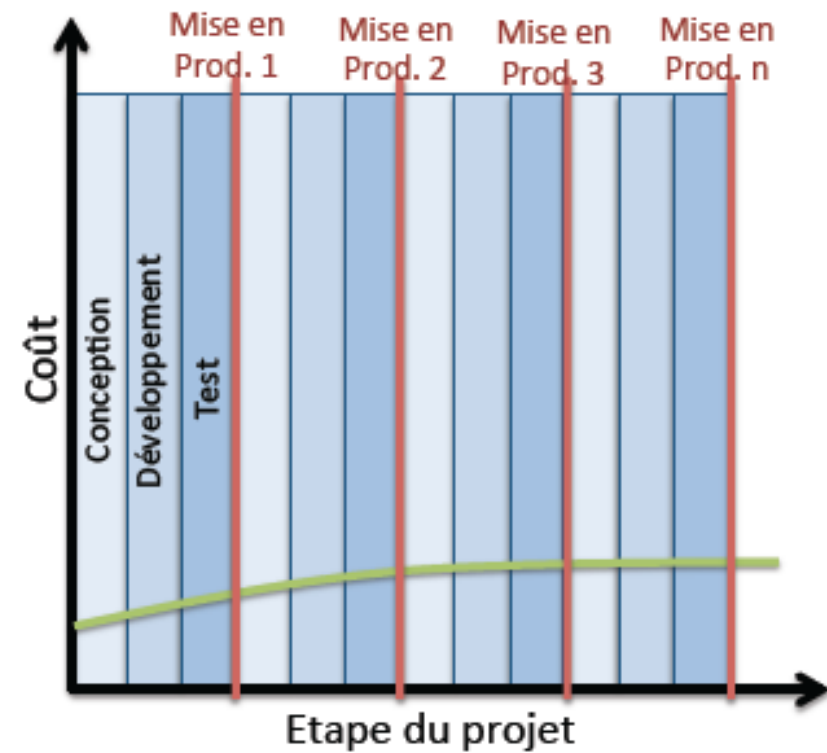
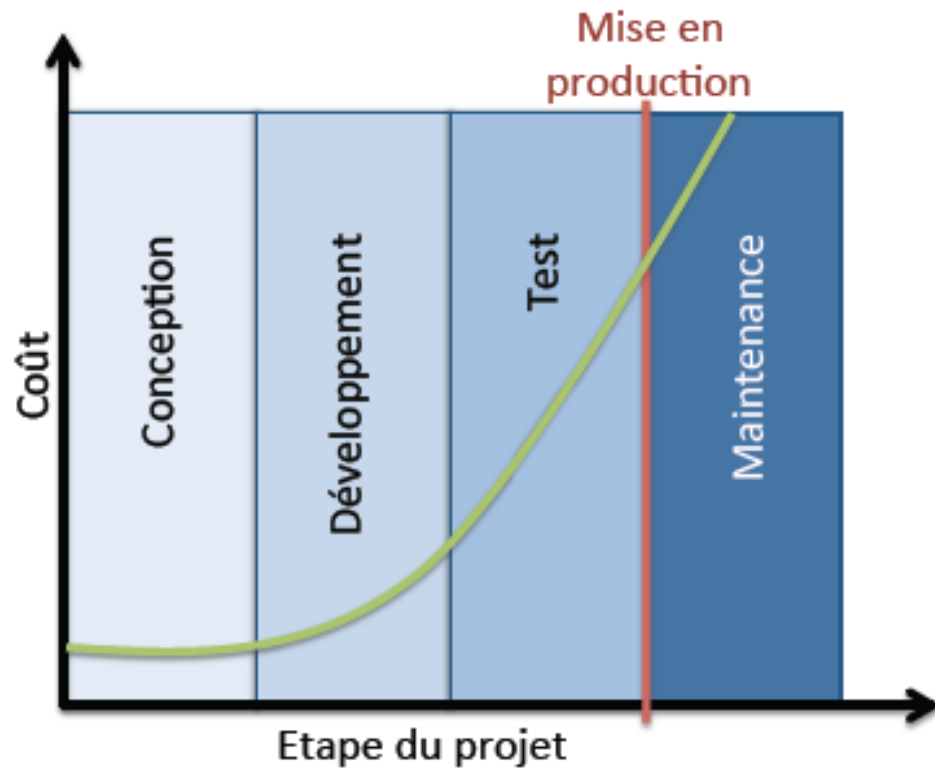


Niveaux de tests (renseignent l'objet du test)



- **Tests (structurels) unitaires**
Test de procédures, de modules, de composants
(coût : 50% du coût total du développement initial correspondant)
- **Tests d'intégration**
Test de bon comportement lors de la composition des procédures, modules ou composants
(coût d'un bug dans cette phase : 10 fois celui d'un bug unitaire)
- **Tests système / de recette**
Validation de l'adéquation aux spécifications
(coût d'un bug dans cette phase : 100 fois celui d'un bug unitaire)

Coût d'un bug



Types de tests (1)

(renseignent la nature du test mené)

- **Tests fonctionnels**

Valide les résultats rendus par les services

- **Tests non-fonctionnels**

Valide la manière dont les services sont rendus

- **Tests nominaux / de bon fonctionnement**

Vérifie que le résultat calculé est le résultat attendu, en entrant des données valides au programme (*test-to-pass*)

- **Tests de robustesse**

Vérifie que le programme réagit correctement à une utilisation non conforme, en entrant des données invalides (*test-to-fail*)

Types de tests (2)

(renseignent la nature du test mené)

- **Test de performance**
 - Load testing (test avec montée en charge)
 - Stress testing (soumis à des demandes de ressources anormales)
- **Test de non-régression**

Vérifie que les corrections ou évolutions dans le code n'ont pas créé d'anomalies nouvelles
- **Test de confirmation**

Valide la correction d'un défaut
- ...

Catégories de tests

Niveau de détail (situation dans le cycle de vie)

système
intégration
module
unitaire

Techniques
(comment on teste)

fonctionnel
non fonctionnel
performance

white box black box

ergonomie

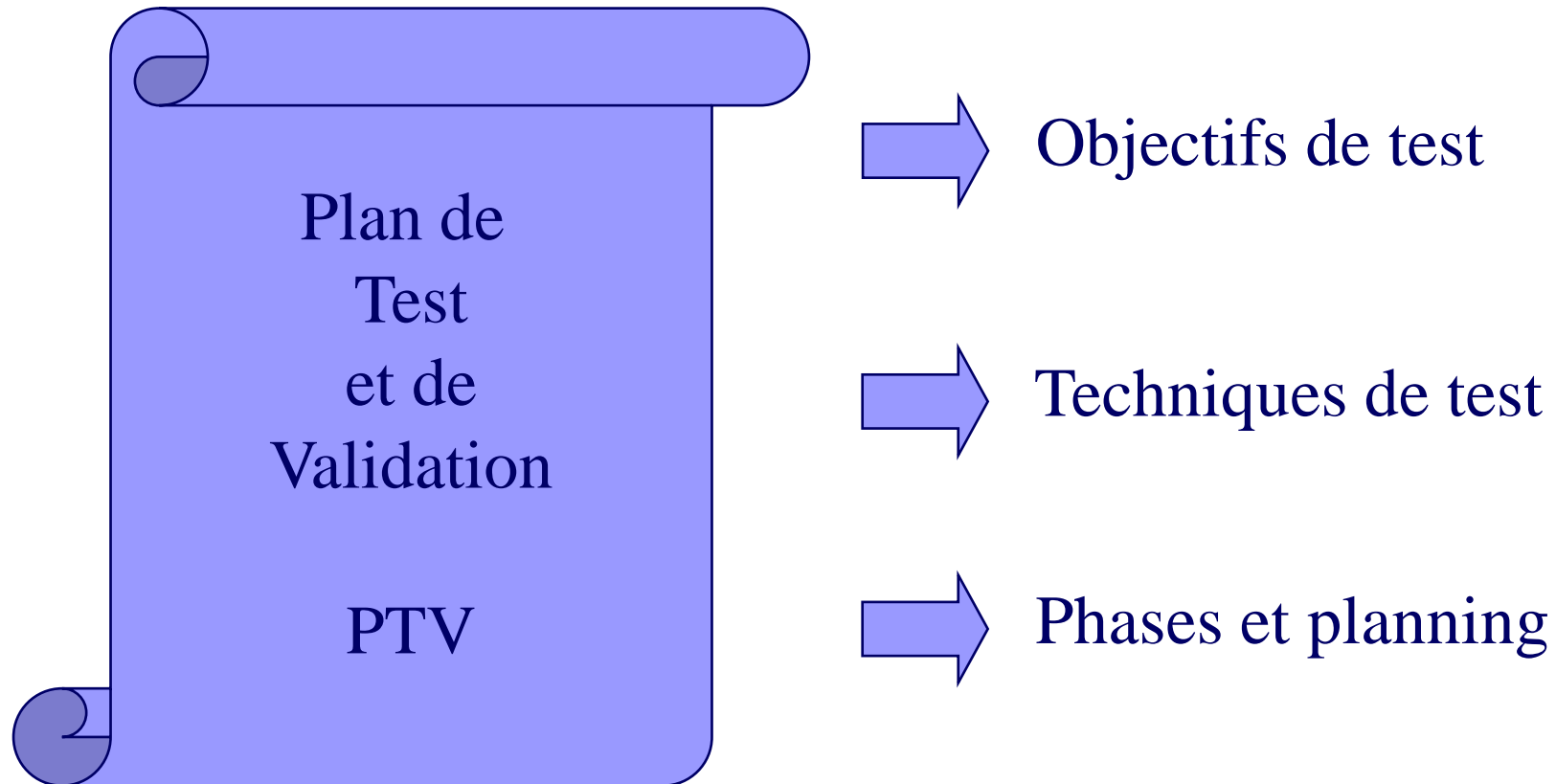
robustesse

sécurité

type (ce que l'on veut tester)

Stratégie de test

- En début de projet, définition d'un Plan de Test et de Validation (PTV)



Acteur du test

- Deux situations :
 - Je teste un programme que j'ai écrit
 - Je teste un programme que quelqu'un d'autre a écrit
- Trois questions :
 - Comment choisir la technique de test ?
=> **boite blanche** ou **boite noire** ?
 - Comment obtenir le résultat attendu ?
=> problème de l'**oracle** du test
 - Comment savoir quand arrêter la phase de test ?
=> problème de l'**arrêt**

Plan de la présentation



- Introduction au test logiciel
- Test structurel
- Test fonctionnel
- Calcul des données de test
- Synthèse

Les bases du test structurel



- Le test structurel s'appuie sur l'analyse du code source de l'application pour établir les tests en fonction de critères de couverture
 - ⇒ Basés sur le graphe de contrôle (toutes les instructions, toutes les branches, tous les chemins, ...)
 - ⇒ Basés sur la couverture du flot de données (toutes les définitions de variables, toutes les utilisations, ...)

Graphe de contrôle



- Permet de représenter n'importe quel algorithme
- Les nœuds représentent des blocs d'instructions
- Les arcs représentent la possibilité de transfert de l'exécution d'un nœud à un autre
- Une seule entrée (nœud à partir duquel on peut visiter tous les autres) et une seule sortie

Graphe de contrôle – Exemple

Soit le programme P1 suivant :

if $x \leq 0$ **then**

$x := -x;$

else

$x := 1 - x;$

if $x = -1$ **then**

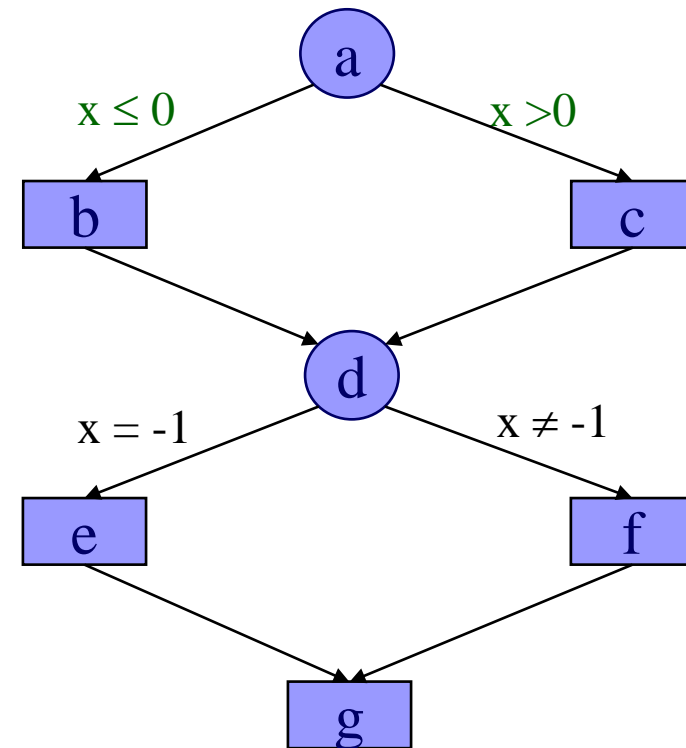
$x := 1;$

else

$x := x + 1;$

writeln(x);

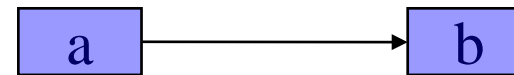
Et son graphe de contrôle G1 :



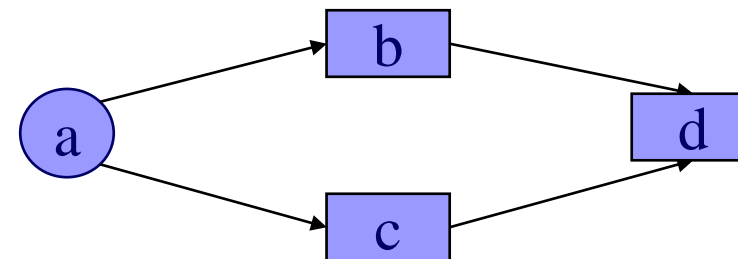
Expression des chemins de contrôle

- On associe une opération d'addition ou de multiplication à toutes les structures primitives apparaissant dans le graphe de flot de contrôle

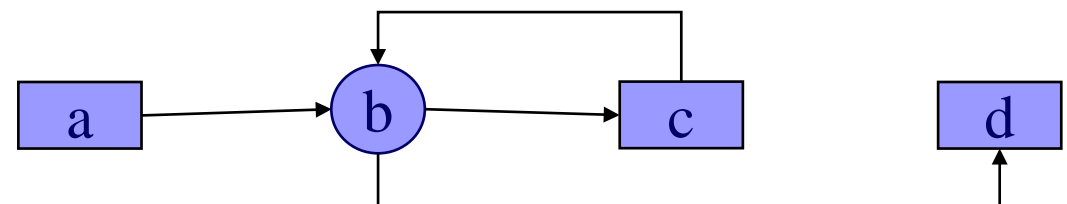
- Forme séquentielle : ab



- Instruction de décision : $a(b+c)d$

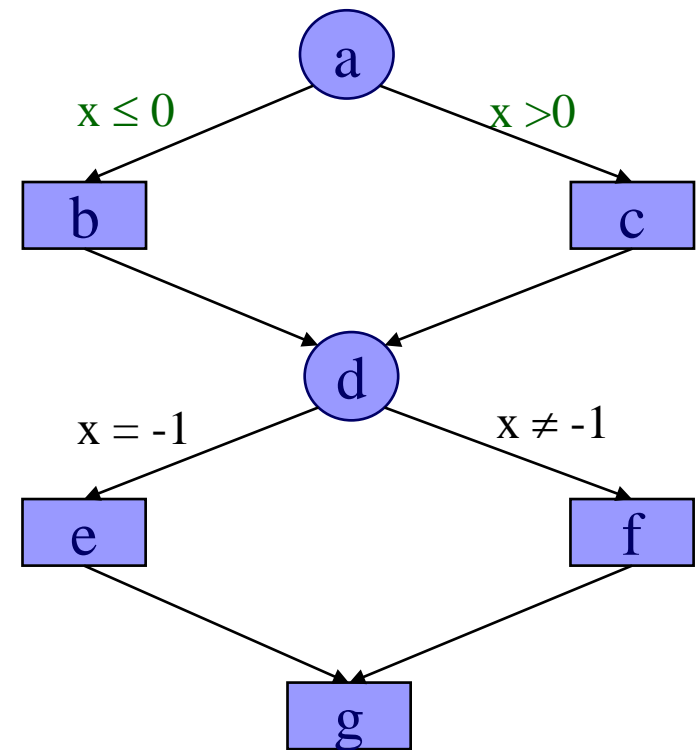


- Structures itératives : $ab(cb)^*d$



Chemins dans le graphe de contrôle

- Le graphe G1 est un graphe de contrôle qui admet une **entrée** (le nœud a) et une **sortie** (le nœud g).
 - le chemin $[a, c, d, e, g]$ est un chemin de contrôle,
 - le chemin $[b, d, f, g]$ n'est pas un chemin de contrôle.
- Le graphe G1 comprend 4 chemins de contrôle :
 - $\beta_1 = [a, b, d, e, g]$
 - $\beta_2 = [a, b, d, f, g]$
 - $\beta_3 = [a, c, d, e, g]$
 - $\beta_4 = [a, c, d, f, g]$

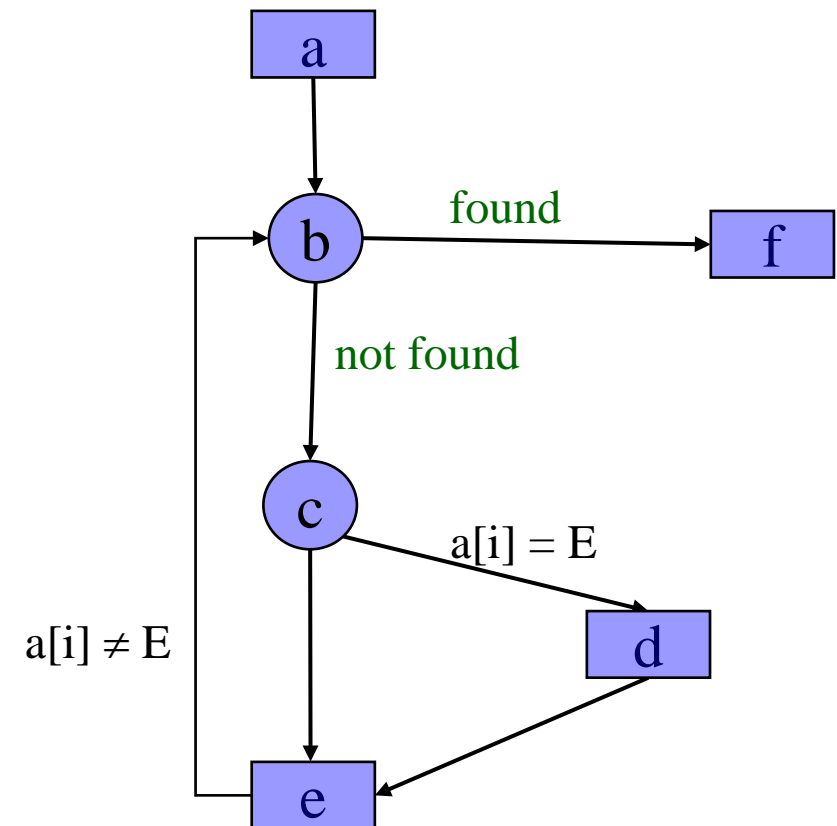


Graphe de contrôle – Exercice

Soit le programme P2 suivant :

```
i := 1;  
found := false;  
while (not found) do  
  begin  
    if (a[i] = E) then  
      begin  
        found := true;  
        s := i;  
      end;  
    i := i + 1;  
  end;
```

Et son graphe de contrôle G2 :



Production des jeux de test à partir du graphe de contrôle

- On considère les chemins du graphe de contrôle (tous ou certains suivant le critère de couverture sélectionné)
- Des données d'entrée, permettant d'activer ces chemins, sont produites
- Le programme est exécuté dans ces configurations
- On recherche les anomalies de fonctionnement qui sont potentiellement détectables sur les chemins considérés

Critères de couverture basés sur le graphe de contrôle



- Couverture de *tous-les-noeuds*,
- Couverture de *tous-les-arcs*,
- Couverture des *chemins limites et intérieurs*
- Couverture de *tous les i-chemins*,
- Couverture de *tous les chemins indépendants*
- Couverture de *tous les chemins*

Couverture de tous-les-noeuds (ou couverture de toutes les instructions)

- Chaque noeud (chaque bloc d'instructions) est atteint par au moins l'un des chemins parmi les chemins qui constituent le jeu de tests
- Lorsqu'un jeu de test permet de couvrir tous les nœuds du graphe, on dit qu'il satisfait $TER=1$ ou $TER1$ (Test Effectiveness Ratio 1)

$TER = 1 \iff$ le critère *tous-les-nœuds* est satisfait
 \iff tous les nœuds du graphe de contrôle ont été couverts
 \iff toutes les instructions ont été exécutées

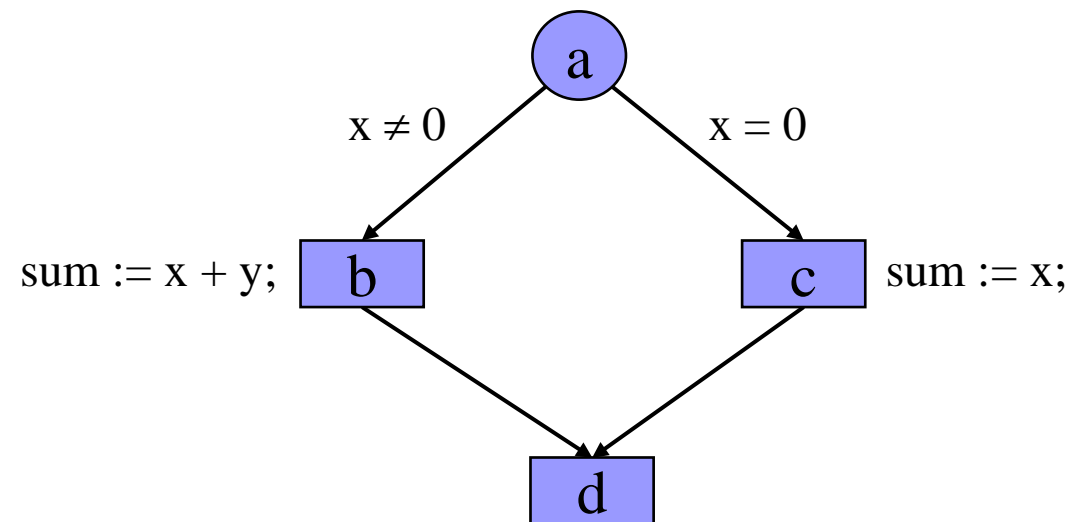
Couverture de tous-les-nœuds

Exemple

► Taux de couverture : $\frac{\text{nb de nœuds couverts}}{\text{nb total de nœuds}}$

► Soit le programme suivant (somme avec erreur) :

```
function sum (x,y : integer) : integer;  
  begin  
    if (x = 0) then  
sum := x ;  
    else  
sum := x + y ;  
    end ;
```



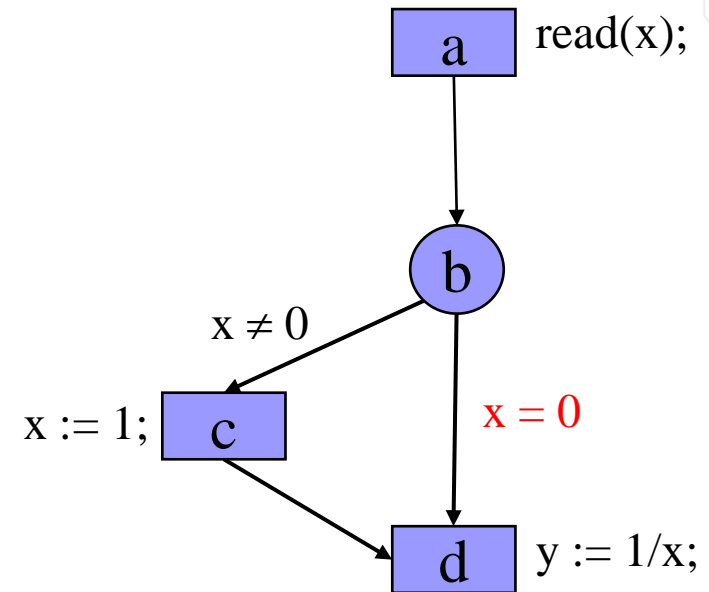
⇒ L'erreur est détectée par l'exécution du chemin [acd]

Couverture de tous-les-nœuds

Limites de ce critère

Soit le programme suivant (avec erreur) :

```
read (x) ;  
if (x ≠ 0) then  
    x := 1;  
y := 1/x;
```



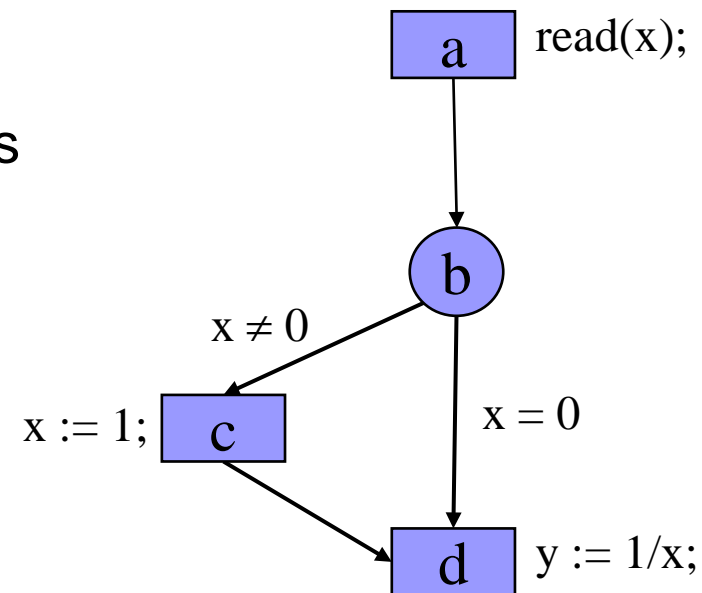
⇒ Le critère *tous-les-nœuds* est satisfait par le chemin [abcd] sans que l'erreur ne soit détectée.

L'unique donnée de test $\{x = 2\}$ permet de couvrir tous les nœuds du graphe sans faire apparaître l'anomalie.

Couverture de tous-les-arcs (ou couverture de tous les enchaînements)

- Taux de couverture :
$$\frac{\text{nb d'arcs couverts}}{\text{nb total d'arcs}}$$
- Impose la couverture de tous les chemins constitués uniquement d'un arc

Sur l'exemple précédent, le jeu de tests doit permettre de couvrir au moins une fois les chemins *ab*, *bc*, *cd* et *bd* pour satisfaire ce critère.



Couverture de tous-les-arcs (ou couverture de tous les enchaînements)

- La couverture de tous-les-arcs équivaut à la couverture de toutes les valeurs de vérité pour chaque nœud de décision (leur valeur de vérité a été au moins une fois vraie et une fois fausse)
- Lorsqu'un jeu de test permet de couvrir tous les arcs du graphe, on dit qu'il satisfait $TER=2$ ou $TER2$ (Test Effectiveness Ratio 2)

$TER = 2 \iff$ le critère *tous-les-arcs* est satisfait

\iff tous les arcs du graphe de contrôle ont été couverts

\iff toutes les décisions ont été exécutées

- Un jeu de test qui satisfait le critère $TER2$, satisfait également le critère $TER1$.

Limites des critères tous-les-arcs

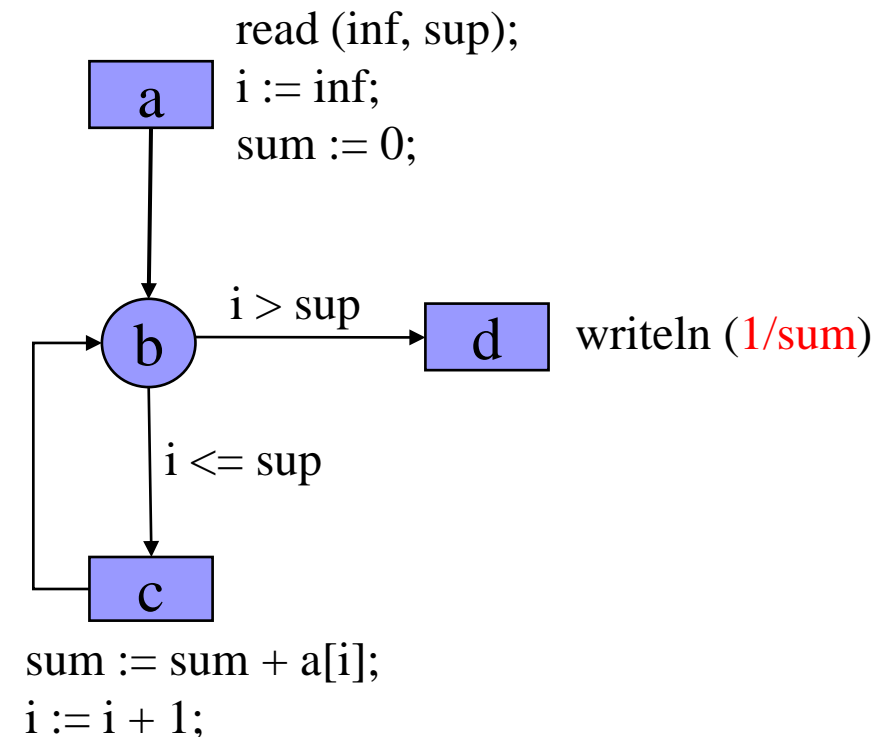
- Pas de détection d'erreurs **en cas de non-exécution d'une boucle**

⇒ Le critère *tous-les-arcs* est satisfait par le chemin [abcbd]

⇒ La donnée de test suivante couvre le critère *tous-les-arcs* :

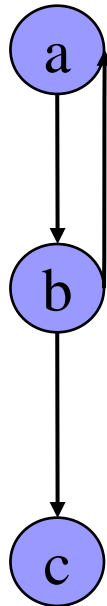
$DT1 = \{a[1]=50, a[2]=60, a[3]=80, inf=1, sup=3\}$

⇒ Problème non détecté par le critère *tous-les-arcs* : si $inf > sup$, erreur sur $1/sum$



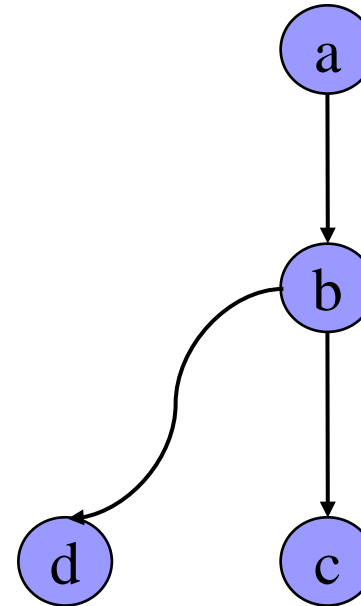
Couverture des boucles

- Chemins limites : traversent la boucle, mais ne l'itèrent pas
- Chemins intérieurs : itèrent la boucle une seule fois



Chemin limite : [abc]

chemin intérieur : [ababc]

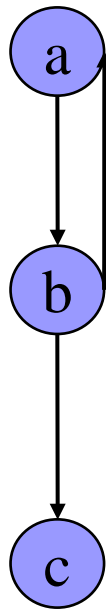


chemin limite : [abd]

chemin intérieur : [abcbd]

Couverture de tous les i-chemins

- Couverture de tous les chemins possibles passant de 0 à i fois dans chaque boucle du graphe de contrôle
- La couverture de *tous les i-chemins* (pour $i > 0$) garantit les critères TER1, TER2 et le critère de couverture des chemins limites et intérieurs



Le jeu de tests constitué de données de test permettant de couvrir les chemins [ABC], [ABABC] et [ABABABC] satisfait le critère *tous les 2-chemins*.

Couverture de tous les chemins indépendants

- Le critère **tous-les-chemins-indépendants** vise à **parcourir tous les arcs dans chaque** configuration possible (et non pas au moins une fois comme dans le critère tous-les arcs)
- Le nombre de chemins indépendants d'un graphe G est donné par le nombre de McCabe, également appelé nombre de cyclomatique, noté $V(G)$
$$V(G) = \text{Nb d'arcs} - \text{Nb de noeuds} + 2$$

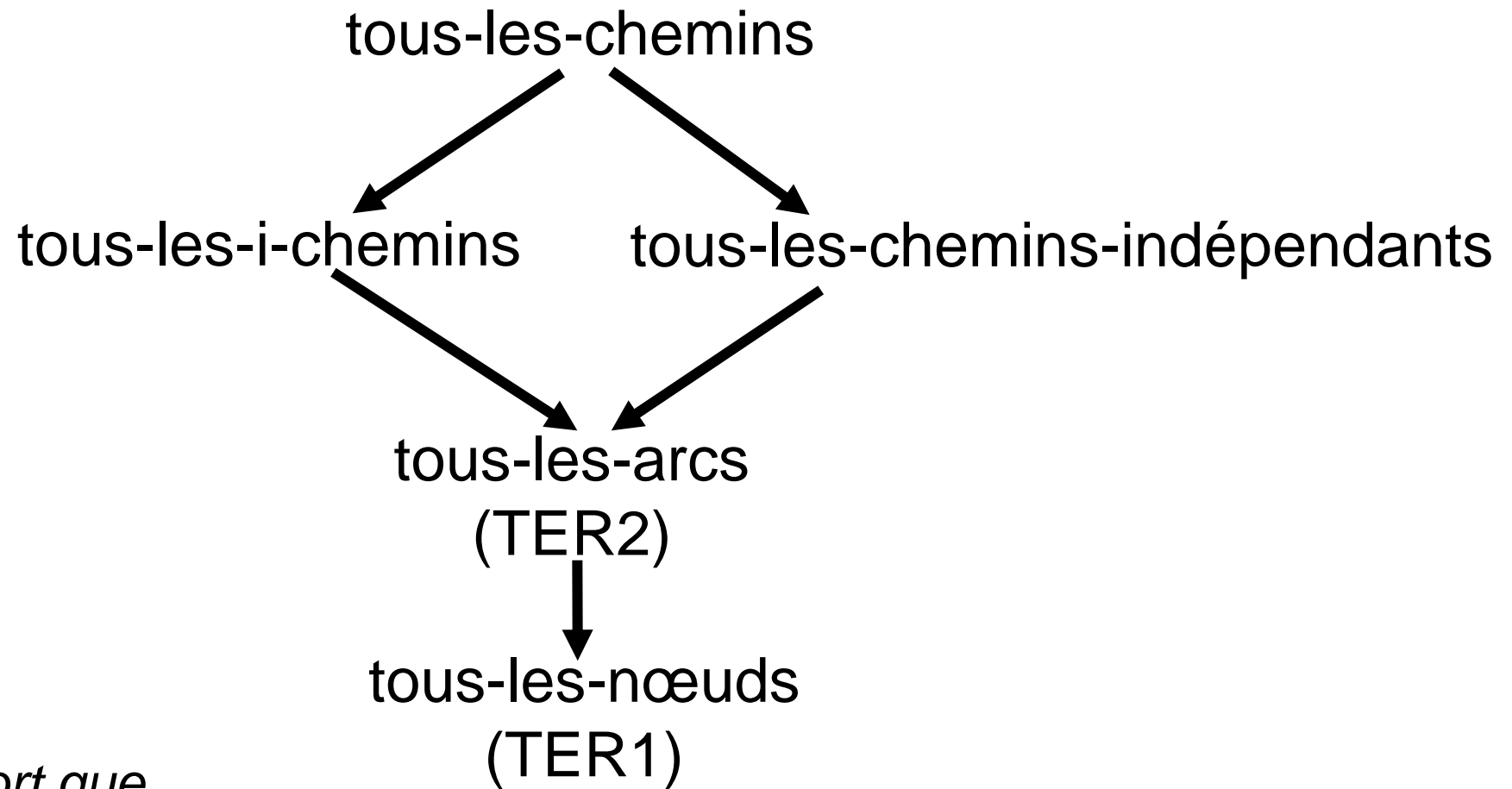
Couverture de tous les chemins indépendants – procédure

1. Evaluer $V(G)$
2. Produire une donnée de test au hasard couvrant le maximum de nœuds de décisions du graphe
3. Produire une donnée de test qui modifie le flot de la première instruction de décision du graphe
4. Vérifier l'indépendance du chemin calculé par rapport aux autres chemins déjà calculés (2 chemins sont indépendants ssi il existe un arc qui est couvert par l'un et pas par l'autre)
5. Recommencer les étapes 3 et 4 jusqu'à la couverture de toutes les décisions
6. S'il n'y a plus de décisions à étudier sur le chemin initial et que le nombre de chemins trouvés est inférieur à $V(G)$, considérer les chemins secondaires déjà trouvés

Couverture de tous les chemins

- En posant $i=n$ (avec n le nombre maximal d'itérations possibles) pour le critère *tous les i -chemins* revient à exécuter tous les chemins possibles du graphe de contrôle
=> Critère *tous-les-chemins*
- En pratique, ce critère est rarement envisageable sur des applications même modestes (explosion du nombre de données de test nécessaires)

Hiérarchie des critères basés sur le graphe de contrôle



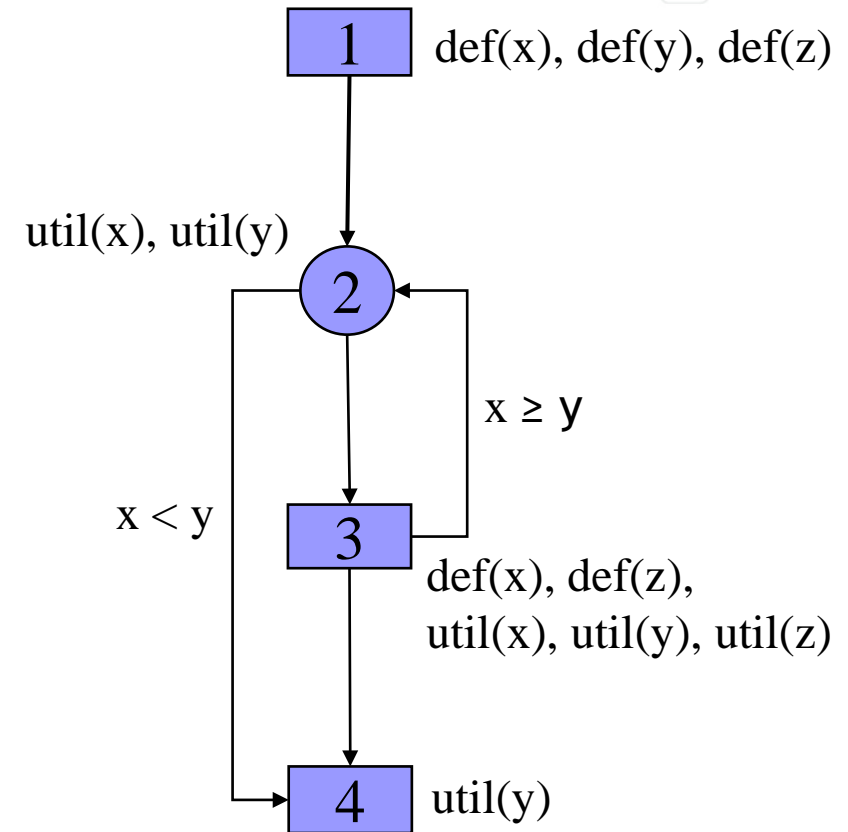
Le flot de données



- Le flot de données est représenté en annotant le graphe de contrôle par certaines informations sur les manipulations de variables par le programme :
 - **Définition** de variables : la valeur de la variable est modifiée
Exemple : membre gauche d'une affectation, paramètre d'une instruction de lecture...
 - **Utilisation** de variables : accès à la valeur de la variable
Exemple : membre droit d'une affectation, paramètre d'une instruction d'écriture, indice de tableau, utilisée dans une condition dans une instruction conditionnelle, dans une boucle...
- Si une variable utilisée l'est dans le prédicat d'une instruction de décision (if, while, etc...), il s'agit d'une **p-utilisation**
- Dans les autres cas (par exemple dans un calcul), il s'agit d'une **c-utilisation**.

Définition et utilisation de variables

open (fichier1) ; read (x,fichier1) ; read (y,fichier1) ; z := 0 ;	Bloc 1 Définitions : x, y, z Utilisations : aucune
----- while x ≥ y do -----	Bloc 2 Définitions : aucune Utilisations : x, y
begin x := x-y ; z := z+1 ; end	Bloc 3 Définitions : x, z Utilisations : x, y, z
----- open (fichier2) ; print (y,fichier2) ; close (fichier1) ; close (fichier2) ;	Bloc 4 Définitions : aucune Utilisations : y

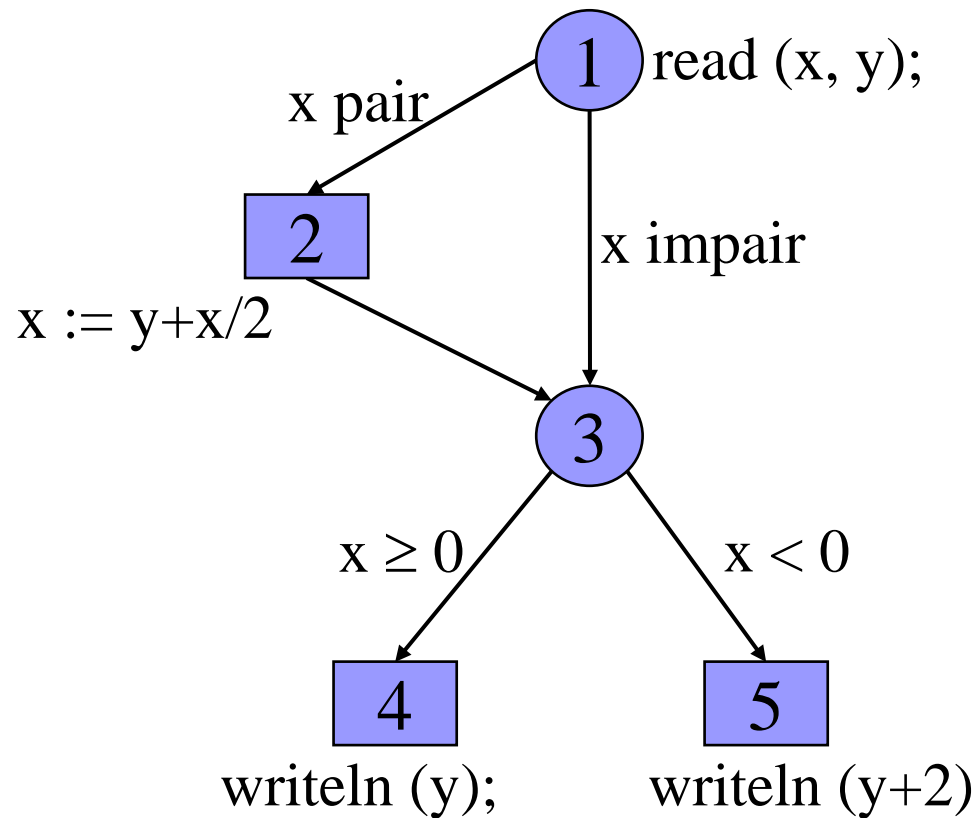


Critères de couverture sur le flot de données

- Toutes les définitions
- Toutes les utilisations
- Tous les DU-chemins

Couverture de toutes les définitions

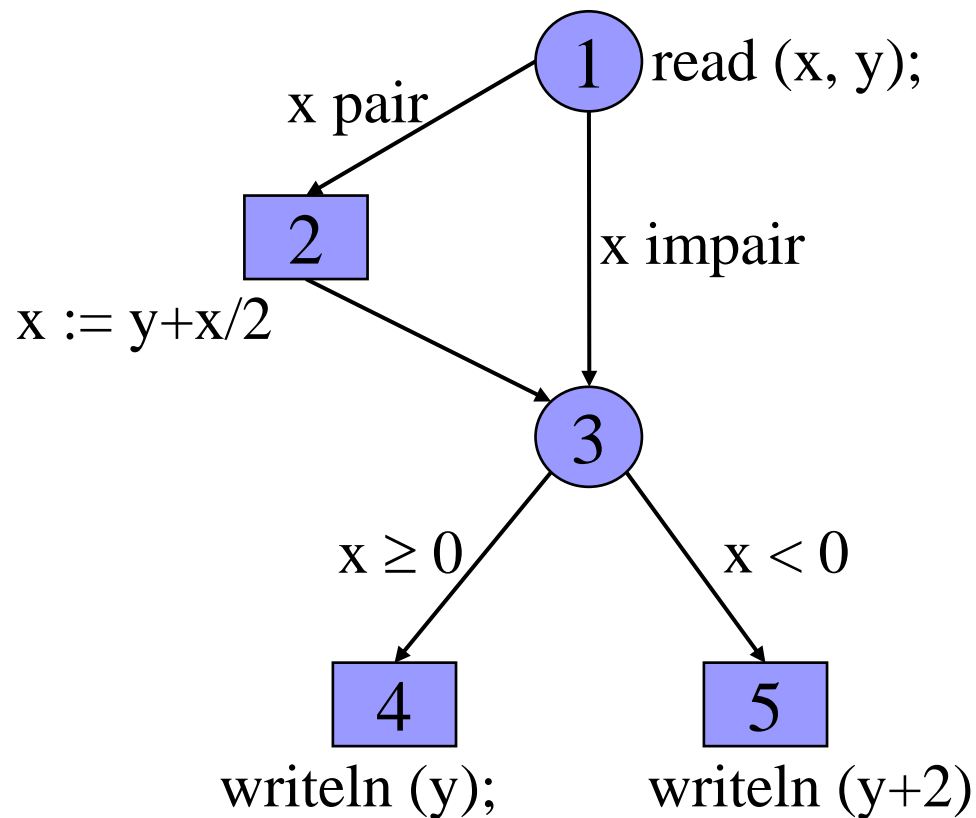
- Pour chaque définition, il existe au moins un chemin qui le couvre dans un test



Couverture du critère
toutes-les-définitions :
- [1,2,3,5]

Couverture de toutes les utilisations

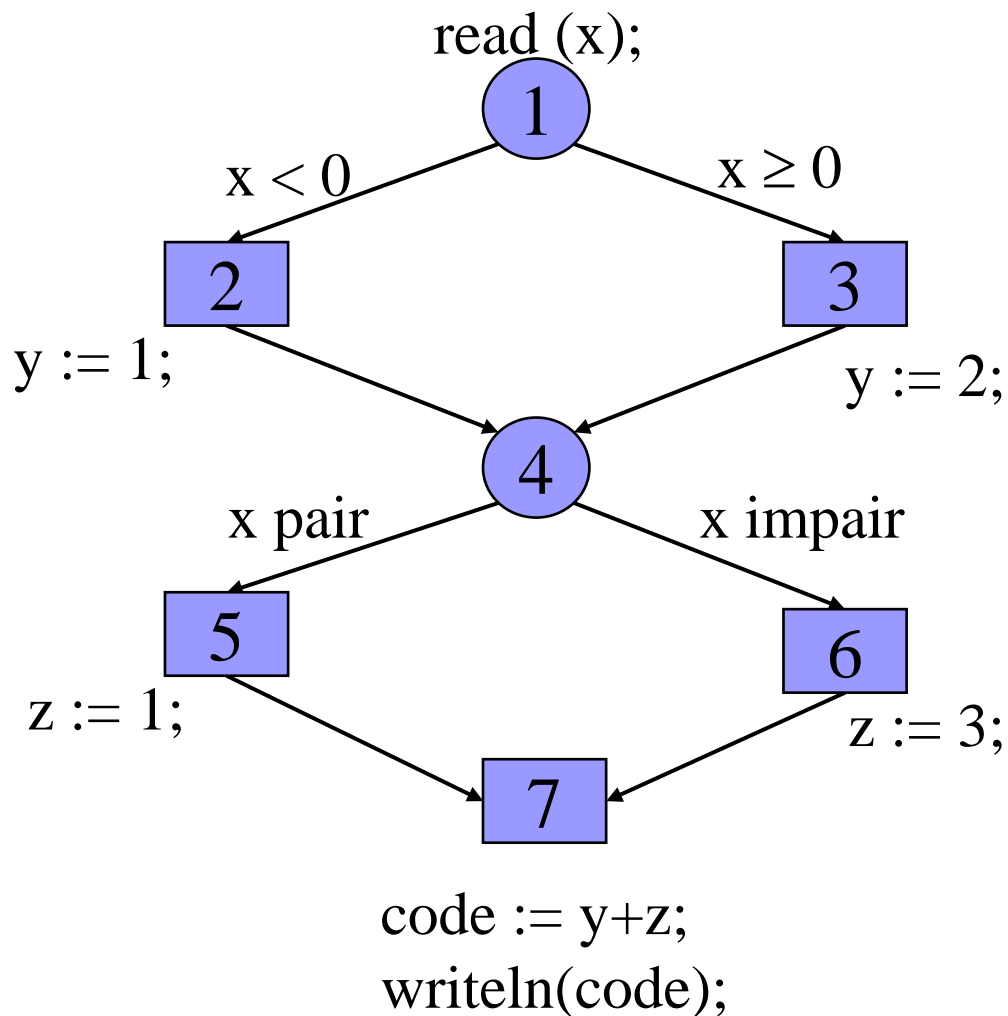
- Pour chaque définition et pour chaque utilisation accessible à partir de cette définition, couverture de toutes les utilisations



Couverture du critère
toutes-les-utilisations :

- [1,3,4]
- [1,2,3,4]
- [1,3,5]
- [1,2,3,5]

Limite du critère toutes-les-utilisations



Couverture du critère
toutes-les-utilisations :

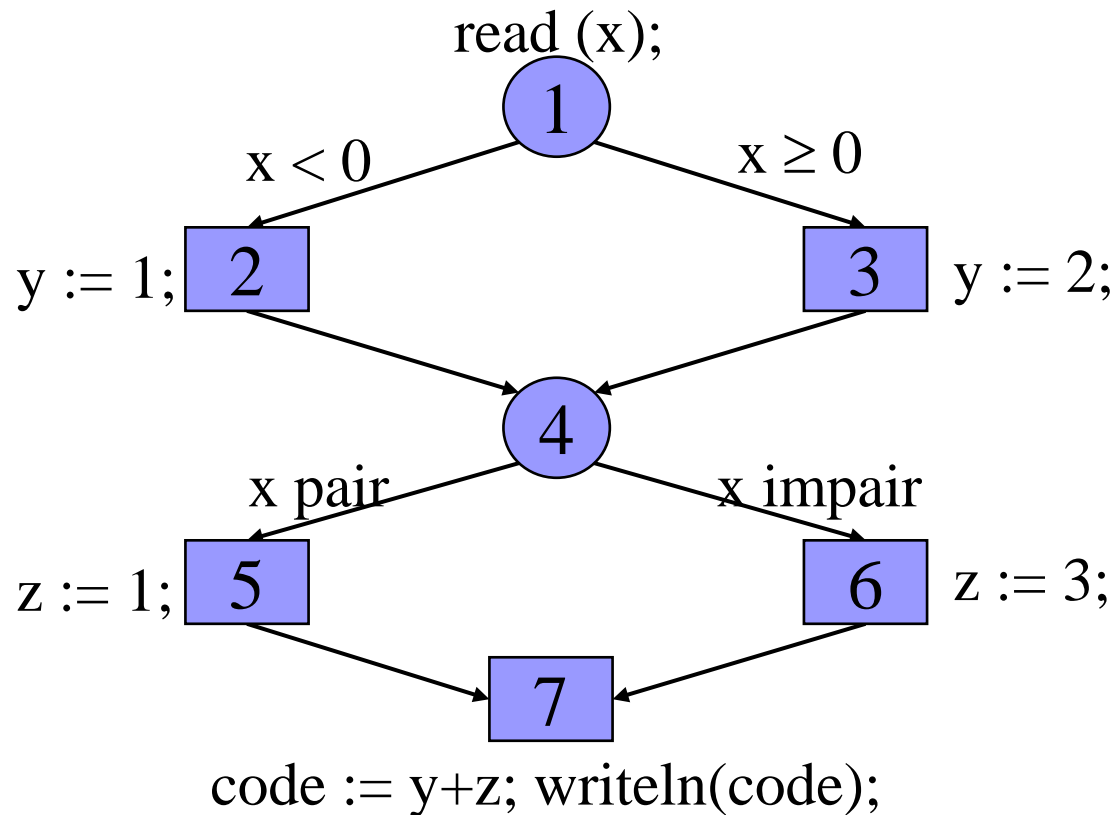
- [1,2,4,5,7]
- [1,3,4,6,7]

Ces deux tests ne couvrent pas
tous les chemins d'utilisation :
(7) peut être utilisatrice
de (2) pour la variable y et
de (6) pour la variable z

=> critère *tous-les-DU-chemins*

Couverture de tous les DU chemins

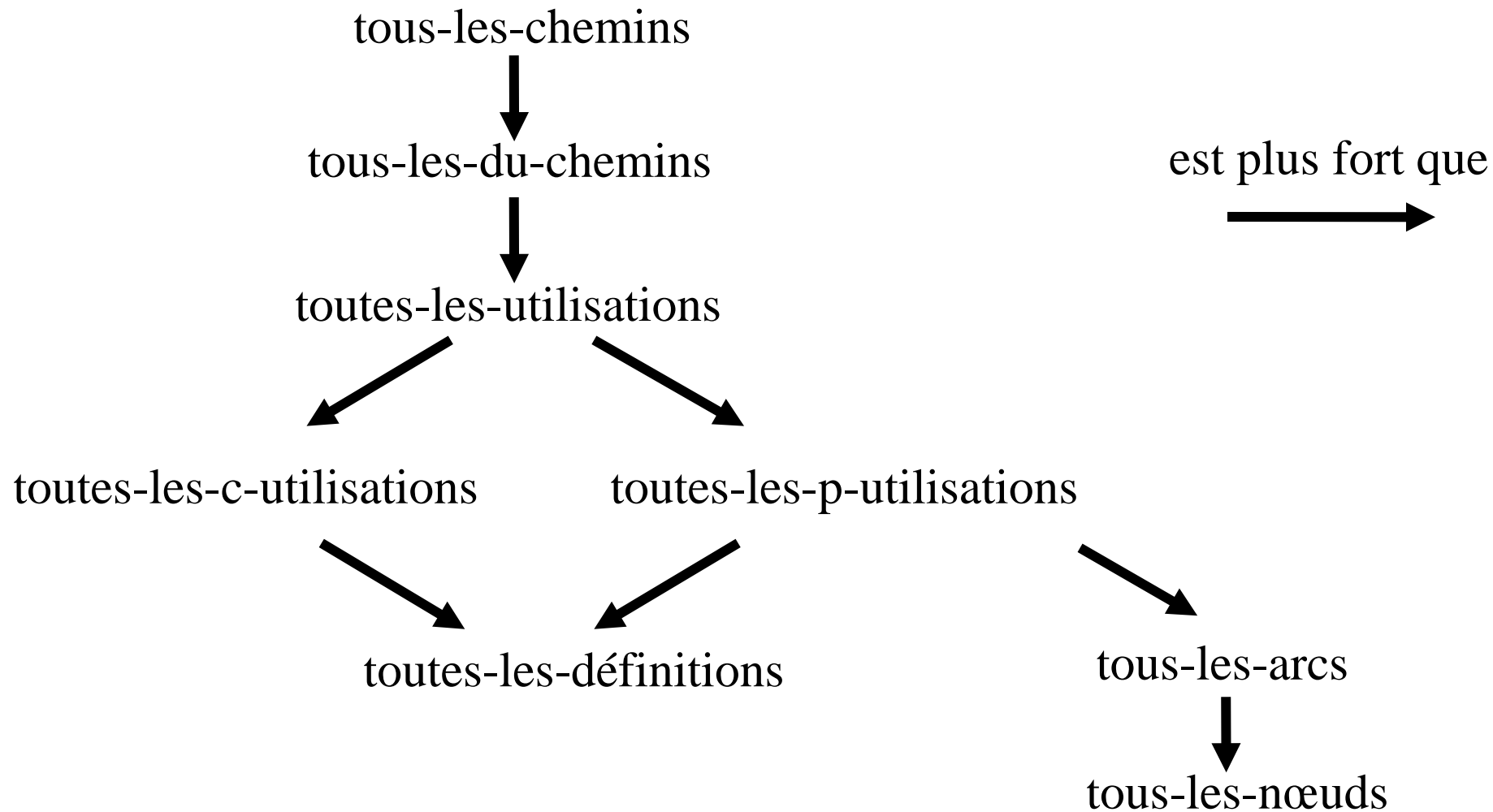
- Ce critère rajoute au critère *toutes-les-utilisations* le fait qu'on doit couvrir tous les chemins possibles entre la définition et la référence, en se limitant aux chemins sans cycle.



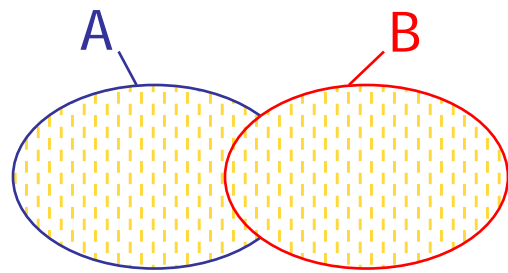
Couverture du critère
tous les DU chemins :

- [1,2,4,5,7]
- [1,3,4,6,7]
- [1,2,4,6,7]
- [1,3,4,5,7]

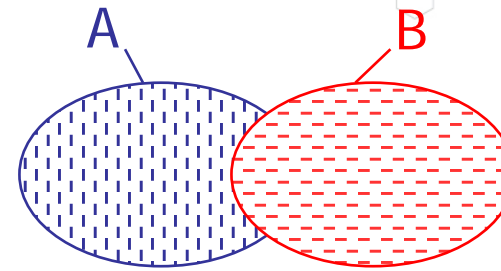
Hiérarchie des critères basés sur le flot de données



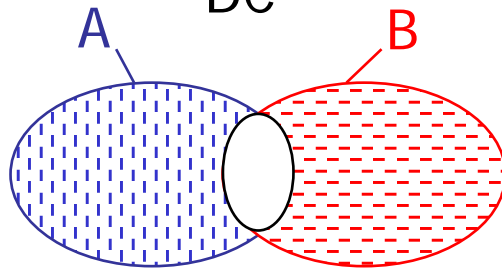
Couverture des Conditions / Decisions



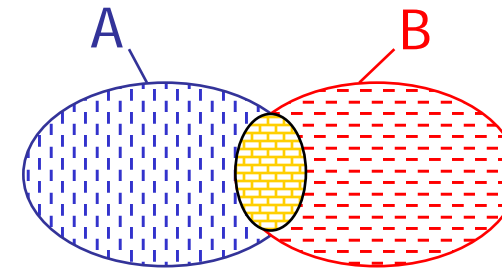
DC



C/DC



FPC



MCC

Décision ($A \vee B$) :

- Couverture des Décisions $\rightarrow A \vee B$
- Couverture des Decisions/Conditions $\rightarrow A, B$
- Full Predicate Coverage $\rightarrow A \wedge \neg B, \neg A \wedge B$
- Multiple Condition Coverage $\rightarrow A \wedge B, A \wedge \neg B, \neg A \wedge B$

Plan de la présentation



- Introduction au test logiciel
- Test structurel
- Test fonctionnel
- Calcul des données de test
- Synthèse

Test fonctionnel – Exemple (1)

Spécification : "Formulaire d'enregistrement pour un site web."



A screenshot of a web registration form. It features three input fields: 'Login:' with the text 'fbouquet', 'Password:' with seven asterisks, and 'Verification:' with seven asterisks. Below the fields are two buttons: 'Register' and 'Cancel'.

5 Cas : 10 tests

- Login (non) vide (2)
- Login (n') existe (pas) (2)
- Password (non) vide (2)
- Password et Verification (ne) sont (pas) les mêmes (2)
- Protocole http(s) (2)

Test fonctionnel – Exemple (2)

Spécification : "Formulaire d'enregistrement pour un site web."

Login: fbouquet

Password: *****

Quality

Verification: *****

Register Cancel

6 Cas : 13 tests

- Login (non) vide (2)
- Login (n') existe (pas) (2)
- Password (non) vide (2)
- Password et Verification (ne) sont (pas) les mêmes (2)
- Protocole http(s) (2)
- Vérifier qualité du password (1 par niveau) : poor, average, good

Test fonctionnel – Exemple (3)

Spécification : "Formulaire d'enregistrement pour un site web."



The image shows a web registration form with the following elements:

- Login:** A text input field containing the text "fbouquet".
- Password:** A text input field containing six asterisks "*****".
- Quality:** A horizontal progress bar with a color gradient from red to green. A black triangle points to the bar, indicating a level of quality.
- Verification:** A text input field containing six asterisks "*****".
- CAPTCHA:** A section with the word "CAPTCHA" in a stylized font, followed by the instruction "Type the word:" and an empty text input field.
- Buttons:** Two buttons at the bottom: "Register" and "Cancel".

7 Cas : 15 tests

- Login (non) vide (2)
- Login (n') existe (pas) (2)
- Password (non) vide (2)
- Password et Verification (ne) sont (pas) les mêmes (2)
- Protocole http(s) (2)
- Vérifier qualité du password (1 par niveau) : poor, average, good
- Vérifier si enregistrement (non) humain (2)



Test fonctionnel

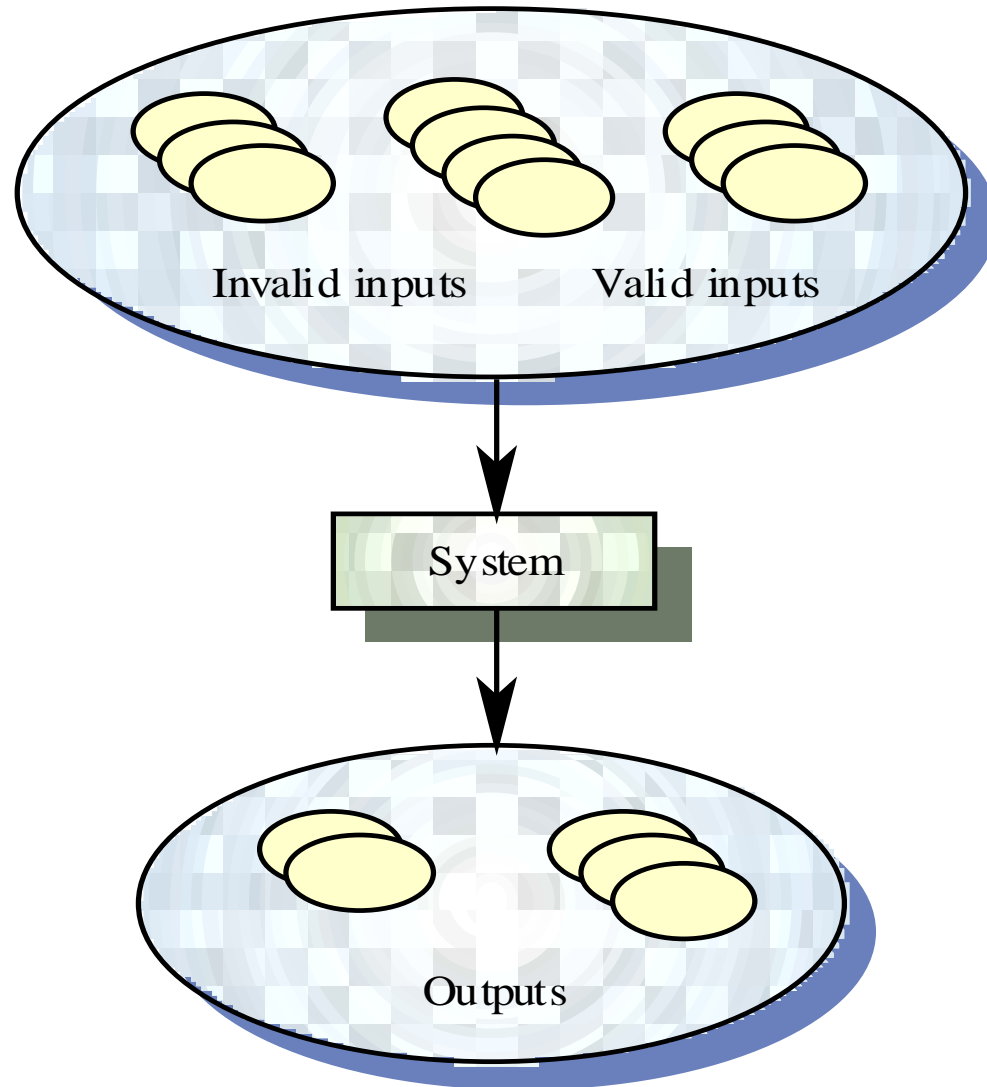
- Le test fonctionnel vise à examiner le comportement fonctionnel du logiciel et sa conformité avec la spécification du logiciel
 - ⇒ Sélection des Données de Tests (DT)
- Méthodes de sélection :
 - Analyse partitionnelle des domaines des données d'entrée
 - Test aux limites
 - Test combinatoire – Algorithmes Pairwise
 - Test aléatoire
 - Génération de tests à partir d'une spécification

Plan de la présentation



- Introduction au test logiciel
- Test structurel
- Test fonctionnel
- Calcul des données de test
- Synthèse

Analyse partitionnelle



Une *classe d'équivalence* correspond à un ensemble de données de tests supposées tester le même comportement, c'est-à-dire activer le même défaut.

Analyse partitionnelle - Méthode

- Trois phases :
 - Pour chaque donnée d'entrée, calcul de classes d'équivalence sur les domaines de valeurs,
 - Choix d'un représentant de chaque classe d'équivalence,
 - Composition par produit cartésien sur l'ensemble des données d'entrée pour établir les données de test.

Règles de partitionnement



- Si la valeur appartient à un intervalle, construire :
 - une classe pour les valeurs inférieures,
 - une classe pour les valeurs supérieures,
 - n classes valides.
- Si la donnée est un ensemble de valeurs, construire :
 - une classe avec l'ensemble vide,
 - une classe avec trop de valeurs,
 - n classes valides.
- Si la donnée est une obligation ou une contrainte (forme, sens, syntaxe), construire :
 - une classe avec la contrainte respectée,
 - une classe avec la contrainte non-respectée



Test aux limites

- Principe : on s'intéresse aux bornes des intervalles partitionnant les domaines des variables d'entrées :
 - pour chaque intervalle, on garde les 2 valeurs correspondant aux 2 limites, et les n valeurs correspondant aux valeurs des limites \pm le plus petit delta possible :
 $n \in 3 \dots 15 \Rightarrow v1 = 3, v2 = 15, v3 = 2, v4 = 4, v5 = 14, v6 = 16$
 - si la variable appartient à un ensemble ordonné de valeurs, on choisit le premier, le second, l'avant dernier et le dernier
 $n \in \{-7, 2, 3, 157, 200\} \Rightarrow v1 = -7, v2 = 2, v3 = 157, v4 = 200$
 - si une *condition d'entrée* spécifie un *nombre de valeurs*, définir les cas de test à partir du nombre *minimum* et *maximum* de valeurs, et des tests pour des nombres de valeurs hors limites invalides.
 $\text{Un fichier d'entrée contient 1-255 records, produire un cas de test pour 0, 1, 255 et 256.}$



Test aux limites - Exemple

- Calcul des limites sur l'exemple du programme de classification des triangles

1, 1, 2	Non triangle
0, 0, 0	Un seul point
4, 0, 3	Une des longueurs est nulle
1, 2, 3.00001	Presque un triangle sans en être un
0.001, 0.001, 0.001	Très petit triangle
99999, 99999, 99999	Très grand triangle
3.00001, 3, 3	Presque équilatéral
2.99999, 3, 4	Presque isocèle
3, 4, 5.00001	Presque droit
3, 4, 5, 6	Quatre données
3	Une seule donnée
	Entrée vide
-3, -3, 5	Entrée négative

Test aux limites – Types de données

- Les données d'entrée ne sont pas seulement des valeurs numériques : booléen, image, son, ...
- Ces catégories peuvent, en général, se prêter à une analyse partitionnelle et à l'examen des conditions aux limites :
 - True / False
 - Fichier plein / Fichier vide
 - Trame pleine / Trame vide
 - Nuances de couleur
 - Plus grand / plus petit
 -

Analyse partitionnelle et test aux limites – synthèse

- L'analyse partitionnelle est une méthode qui vise à diminuer le nombre de cas de test par calcul de classes d'équivalence
 - importance du choix de classes d'équivalence : risque de ne pas révéler un défaut
- Le choix de conditions d'entrée aux limites est une heuristique solide de choix de données d'entrée au sein des classes d'équivalence
 - cette heuristique n'est utilisable qu'en présence de relation d'ordre sur la donnée d'entrée considérée.
- Le test aux limites produit à la fois des cas de test nominaux (dans l'intervalle) et de robustesse (hors intervalle)



Test aux limites - Evaluation

- Méthode de test fonctionnel très productive :
 - le comportement du programme aux valeurs limites est souvent pas ou insuffisamment examiné
- Couvre l'ensemble des types de test
- Inconvénient : caractère parfois intuitif ou subjectif de la notion de limite
 - ⇒ Difficulté pour caractériser la couverture de test.

Méthode pour le test combinatoire

- Les combinaisons de valeurs de domaines d'entrée donne lieu à explosion combinatoire
- Exemple : Options d'une boîte de dialogue MS Word



→ $2^{12} * 3$ (nombre d'item dans le menu déroulant) = 12 288



Test combinatoire : Pair-wise

- Tester un fragment des combinaisons de valeurs qui garantissent que chaque combinaison de 2 variables est testé
- Exemple : 4 variables avec 3 valeurs possibles chacune

OS	Réseau	Imprimante	Application
XP	IP	HP35	Word
Linux	Wifi	Canon900	Excel
Mac X	Bluetooth	Canon-EX	Pwpoint

Toutes les combinaisons : 81

Toutes les paires : 9

Pairwise – Exemple

- 9 cas de test : chaque combinaison de 2 valeurs est testée

	OS	Réseau	Imprimante	Application
Cas 1	XP	ATM	Canon-EX	Pwpoint
Cas 2	Mac X	IP	HP35	Pwpoint
Cas 3	Mac X	Wifi	Canon-EX	Word
Cas 4	XP	IP	Canon900	Word
Cas 5	XP	Wifi	HP35	Excel
Cas 6	Linux	ATM	HP35	Word
Cas 7	Linux	IP	Canon-EX	Excel
Cas 8	Mac X	ATM	Canon900	Excel
Cas 9	Linux	Wifi	Canon900	Pwpoint

L'idée sous-jacente : la majorité des fautes sont détectées par des combinaisons de 2 valeurs de variables



Test combinatoire

- L'approche Pair-wise se décline avec des triplets, des quadruplets, mais le nombre de tests augmente très vite
- Une dizaine d'outils permette de calculer les combinaisons en Pairwise (ou n-valeurs) :
 - <http://www.pairwise.org/default.html>
 - Prise en charge des exclusions entre valeurs des domaines et des combinaison déjà testée
- Problème du Pair-wise :
 - le choix de la combinaison de valeurs n'est peut-être pas celle qui détecte le bug ...
 - Le résultat attendu de chaque test doit être fourni manuellement



Test aléatoire ou statistique

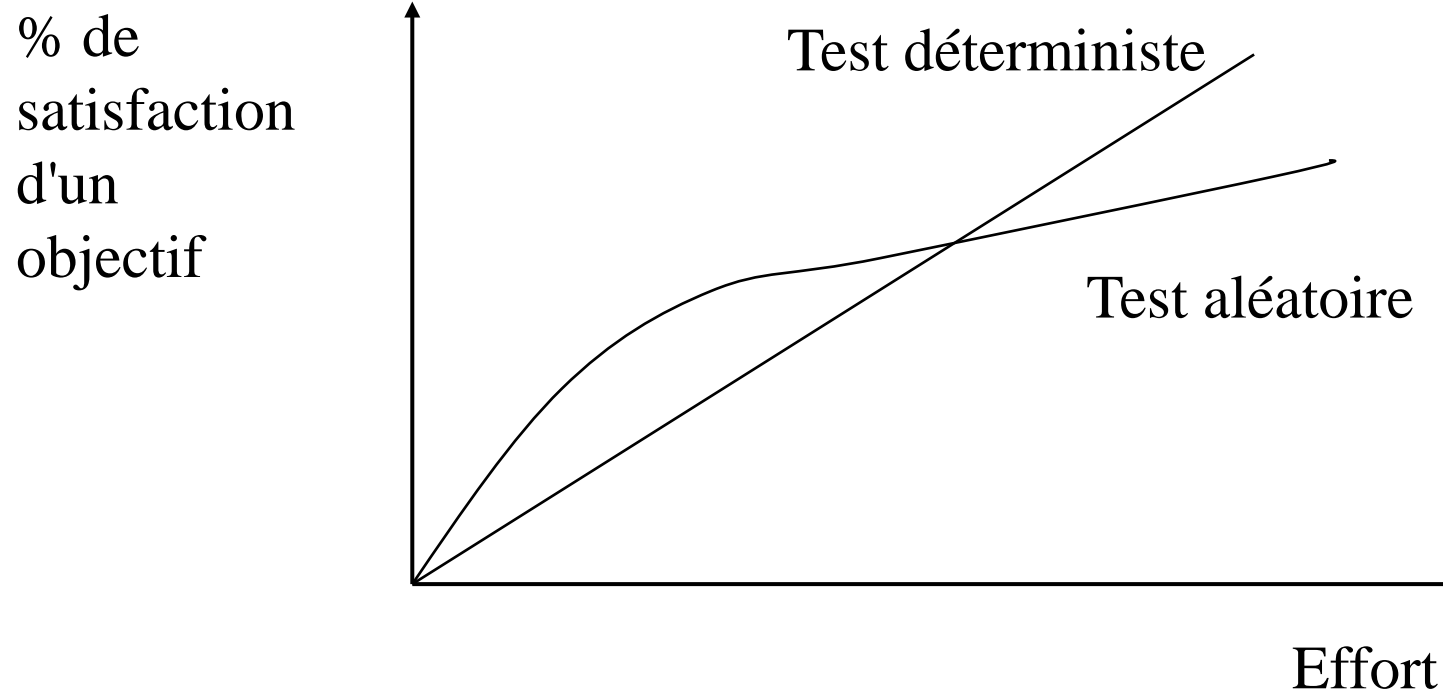
- Principe : utilisation d'une fonction de calcul pour sélectionner les données de test :
 - fonction aléatoire : choix aléatoire dans le domaine de la donnée d'entrée,
 - utilisation d'une loi statistique sur le domaine.
- Exemples :
 - Echantillonnage de 5 en 5 pour une donnée d'entrée représentant une distance,
 - Utilisation d'une loi de Gauss pour une donnée représentant la taille des individus,
 - ...

Evaluation du test aléatoire



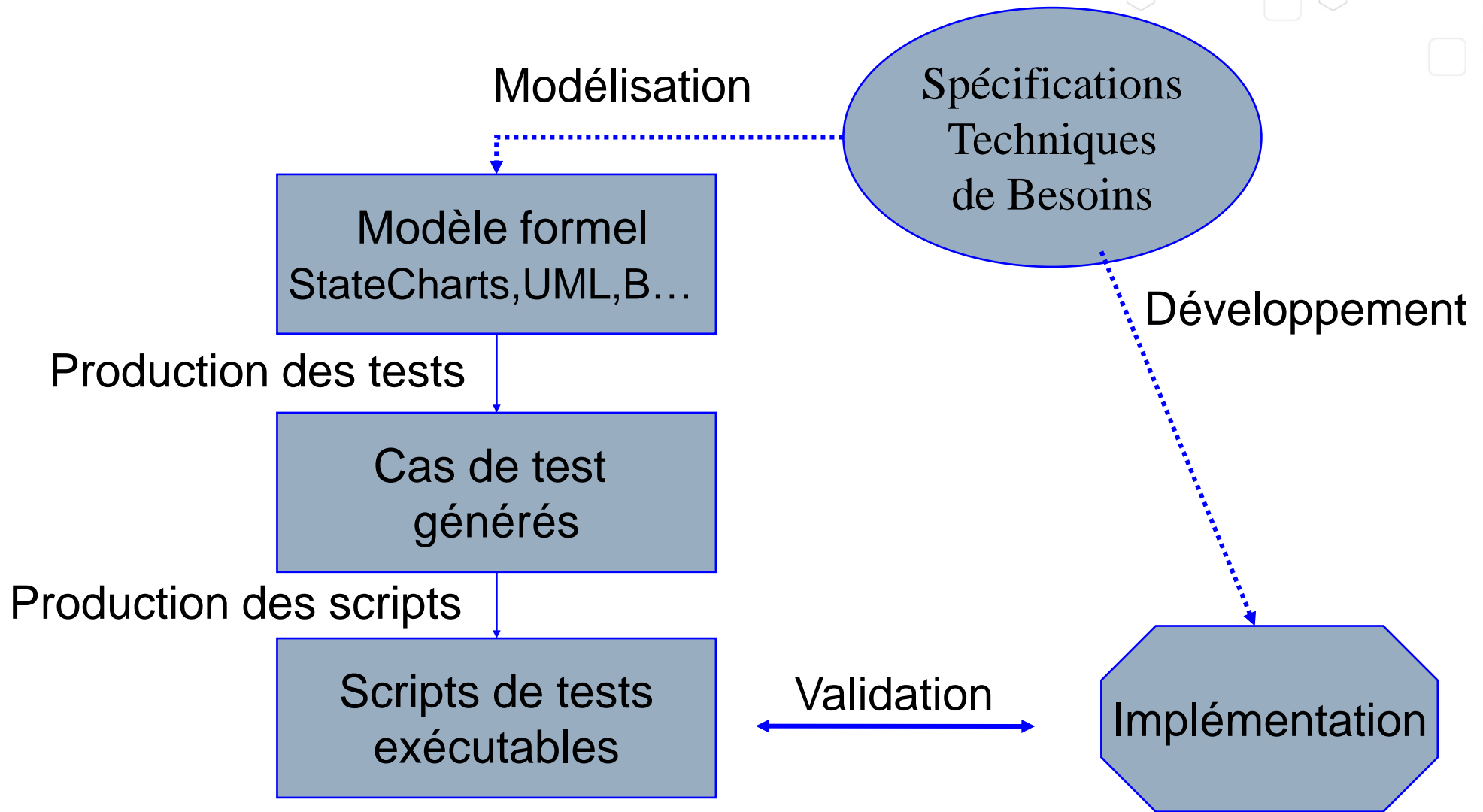
- Intérêts de l'approche statistique :
 - facilement automatisable pour la sélection des cas de test,
(plus difficile pour le résultat attendu)
 - objectivité des données de test.
- Inconvénients :
 - fonctionnement en aveugle,
 - difficultés de produire des comportements très spécifiques.

Productivité du test aléatoire ou statistique



Les études montrent que le test statistique permet d'atteindre rapidement 50 % de l'objectif de test mais qu'il a tendance à plafonner ensuite.

Test à partir de modèles

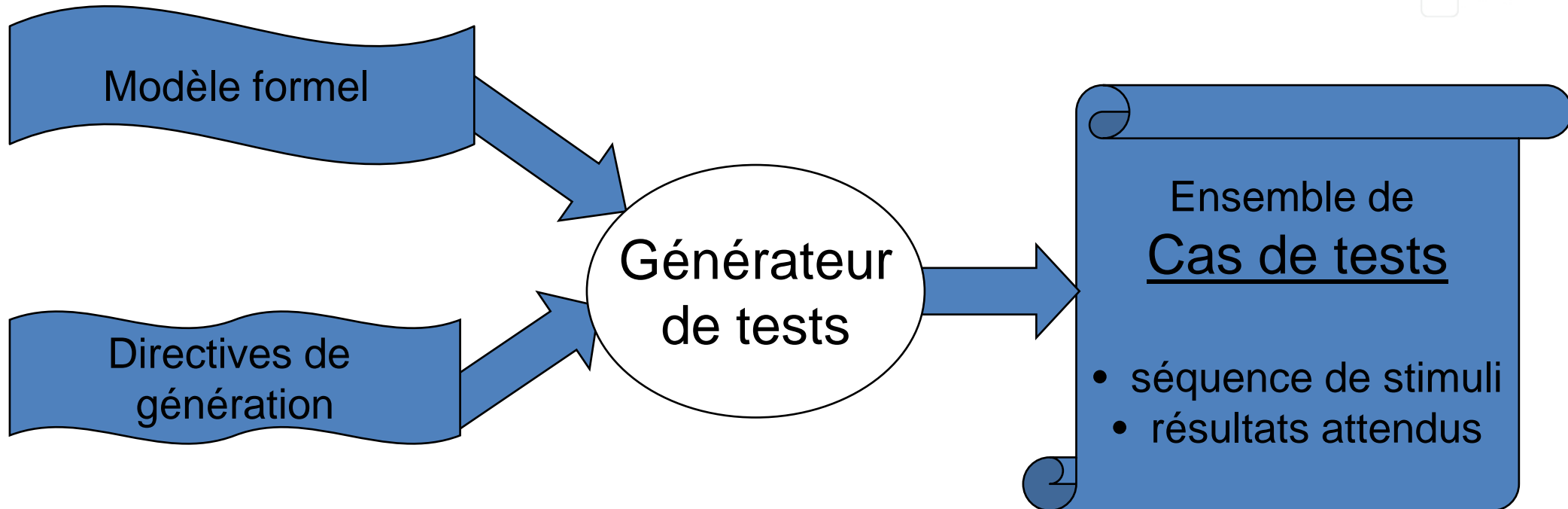


Langages de modélisation



- De nombreux paradigmes
 - Systèmes de transitions (Etats / Transitions / Evénements)
 - Flow Charts
 - Data Flow Diagrams
 - Diagramme d'état
 - Diagrammes objet & association (Entité-Relation, héritage, ...)
 - Diagramme de classe (UML)
 - Entité-Relation
 - Représentation Pré-Post conditions
 - OCL (UML)
 - Machine Abstraite B
- Représentation :
 - Graphique (plus « intuitive »)
 - Textuelle (plus précise)

Model Based Test Generation



- Directives de génération (définition de scénarii de test) :
 - Critères de couverture du modèle
 - Critères de sélection sur le modèle

Plan de la présentation



- Introduction au test logiciel
- Test structurel
- Test fonctionnel
- Calcul des données de test
- Synthèse

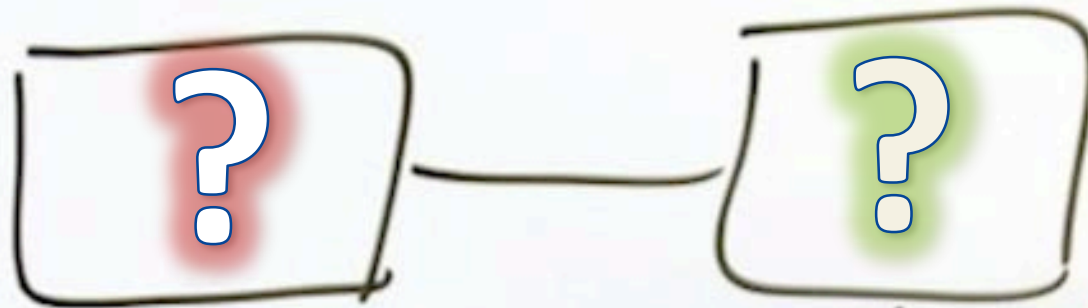
Synthèse

- La production de test s'appuie (généralement) sur une analyse du programme (test structurel) ou de sa spécification (test fonctionnel)
- Différentes stratégies permettent de sélectionner des données de test pertinentes.
- Ces stratégies ne sont que des heuristiques !
Elles ne fournissent aucune garantie de sélectionner la valeur qui révélera les erreurs du programme.

Programme

- Mardi 10/12/13 : test structurel
 - Java : Junit, Mockito, Jacoco
 - C++ : CPPUnit, GoogleMock, Gcov
- Mercredi 11/12/13 : test fonctionnel
 - ALL : Zest, Squash TM
 - Java : Junit, Concordion
 - C++ : Cucumber, GTest
- Jeudi 12/12/13 : retours d'expériences

Merci pour votre attention...



"Testing is always model-based!"
Robert Binder

