

# Tests des logiciels

*Jean-François Pradat-Peyre*

*2016*

***4 : Techniques de test dynamiques :***

***Tests boîtes blanches et couverture***

## ■ Tests dynamiques

- On exécute le programme avec des valeurs en entrée et on observe le comportement

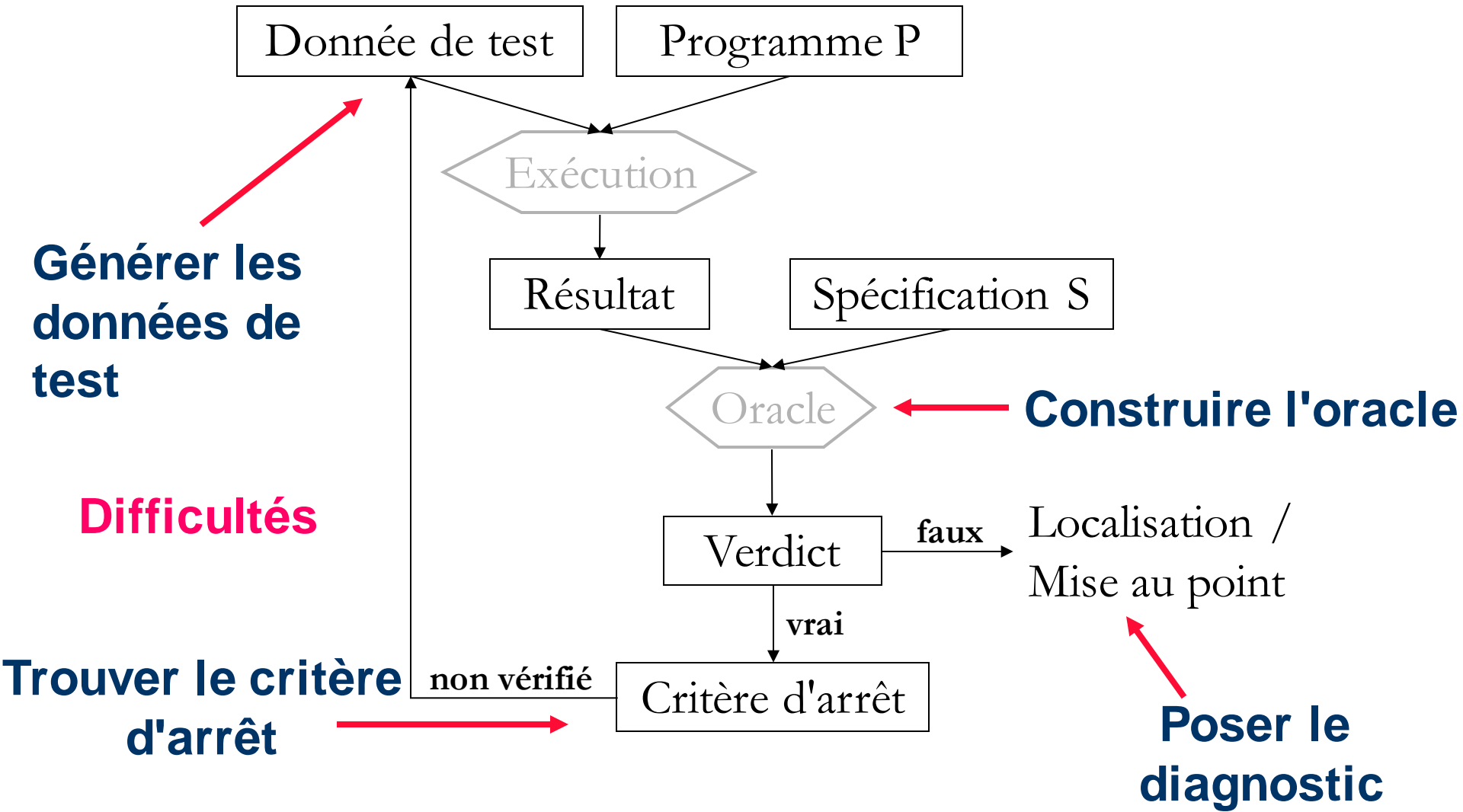
## ■ Tests statiques

- Revue (analyse sans exécuter le programme ou faire fonctionner le produit)
- Analyse automatique (vérification de propriétés, règles de codage...)

## Techniques de tests – Techniques dynamiques

- ❖ Panorama et problématique
- ❖ Tests « boîte noire »
- ❖ Tests « boîte blanche » et couverture de code

# Processus de test « dynamique »



# La problématique du test dynamique

- Soit **D** le domaine d'entrée d'un programme **P** spécifié par **S** ;  
on voudrait pouvoir garantir que

- $\forall x \in D, P(x) = S(x)$

i.e. pour toute donnée valide, le programme se comporte comme sa spécification

- Problème : le test exhaustif est **impossible** dans la plupart des cas

- Domaine D trop grand, voire infini
  - Trop long et trop coûteux

- Recherche d'un ensemble de données de test **TI** tel que
  - TI est inclus dans D, *fini et bien plus petit*
  - Si pour tout x dans **TI**,  $P(x) = S(x)$  alors **pour tout x dans D**,  $P(x) = S(x)$
- Le critère d'arrêt des test est :
  - {données de test} = **TI**
- Problème :
  - Comment construire **TI** ?

# Génération des données de tests :

## Tests « boîte noire » vs « boîte blanche »

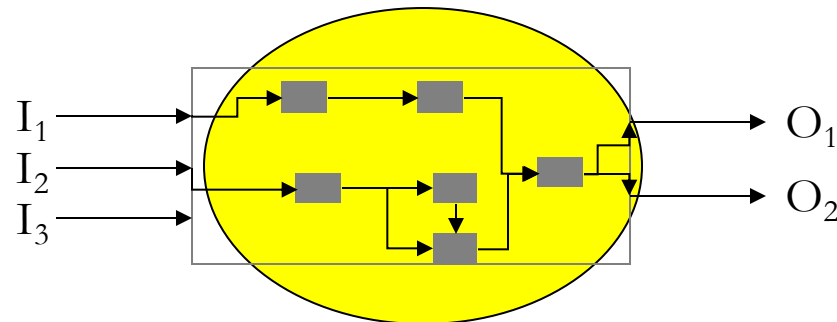
Test fonctionnel (test boîte noire)

Utilise la description des fonctionnalités du programme



Test structurel (test boîte blanche)

Utilise la structure interne du programme



## **Techniques de tests – Techniques dynamiques**

- ❖ Panorama et problématique
- ❖ Tests « boîte noire »
- ❖ Tests « boîte blanche »



- Utilisent la structure pour dériver des cas de tests
  - Au niveau unitaire : utilise le code (instructions, conditions, branchement)
  - Au niveau intégration : utilise le graphe d'appel entre modules
  - Au niveau système : utilise les menus, les processus
  
- Complètent les tests boîte noire en étudiant la réalisation (et non la spécification)
  
- Sont associés à des critères de couverture
  - Couverture de tous les appels de sous programmes / méthodes
  - Couverture de toutes les instructions
  - Couverture de toutes les conditions
  - Couverture de toutes les utilisations d'une variable
  - etc.

# Utilisation des tests boîte blanche en pratique

- On conçoit initialement les scénarios de tests à partir des spécifications (tests boîte noire, conception manuelle)
- On déduit les cas de tests (jeux de valeurs) à partir des spécifications et des méthodes d'abstraction (pairwise, partition, table de décisions, etc., conception manuelle)
- On joue les jeux de tests et on **mesure** la couverture obtenue par ces jeux de tests (utilisation importante d'outils)
- On complète les cas de tests (boîte noire) construits initialement par des cas de tests (boîte blanche) permettant d'obtenir une meilleure couverture ou permettant de mettre en avant un défaut structurel (utilisation d'outils recommandé)

Deux familles de méthodes sont utilisées

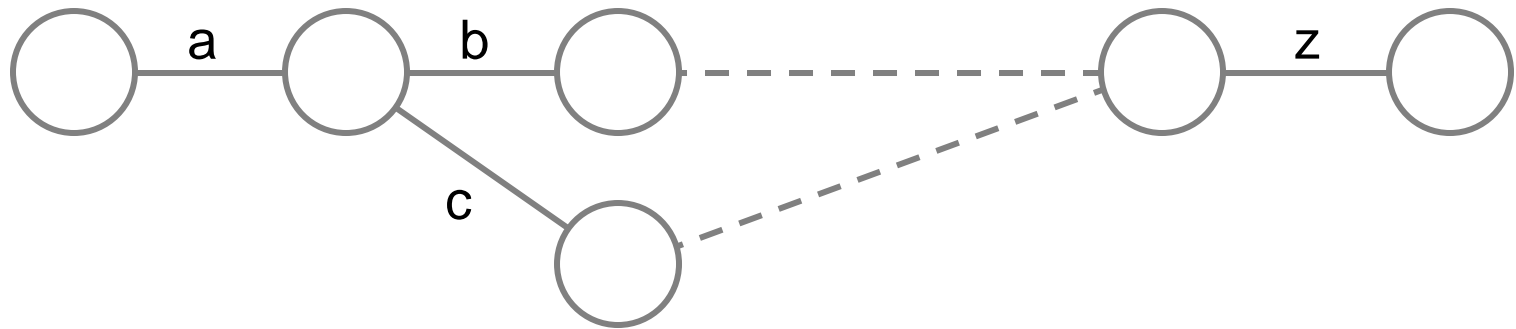
- Dérivation de jeux de tests à partir du graphe de contrôle
  - On suit l'enchaînement des opérations qui sont représentées par un graphe
  - On cherche à couvrir toutes les instructions, tous les chemins ou des critères affaiblis
  
- Dérivation de jeux de tests à partir du flot de données
  - On suit la vie des variables au cours de l'exécution du programme : définition, utilisation, destruction
  - On cherche à couvrir toutes les affectations, toutes les utilisations, ou des critères affaiblis

# Test partir du graphe de contrôle

- C'est la technique de test structurel la plus ancienne
- C'est aussi la technique de base pour beaucoup d'autres tests
- Détaillée et complexe
- Utilisée principalement pour les tests unitaires et les tests de composants
- Basée sur les chemins d'exécutions
- De très nombreux types de couverture peuvent être visés
- Associée à de nombreuses normes de certifications du logiciel

# Chemins

- Un chemin est une suite d'arcs reliant un point d'entrée du programme à un point de sortie
- Il y a **beaucoup** de chemins possibles!



- Certains chemins sont impossibles

Ex:  $a = (x:=y, y>0)$  ;  $c = (x<0 ?)$

Logique de comportement

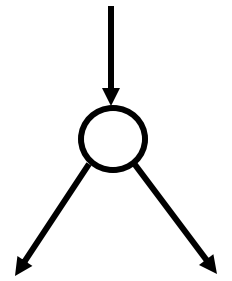
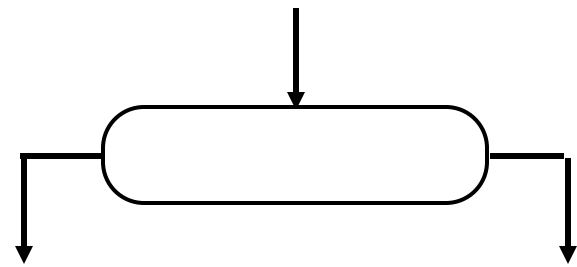
Logique structurelle

- Un graphe de contrôle est constitué d'arcs et de sommets
- Chaque sommet représente une ou plusieurs instructions
- Chaque arc représente un saut d'instruction:
  - soit un départ conditionnel (sur 2 branches ou plus)
  - soit la jonction de branches incidentes
- Un graphe de contrôle permet de construire les chemins

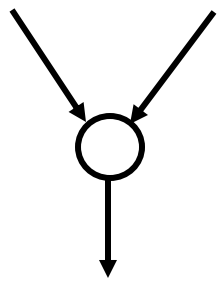
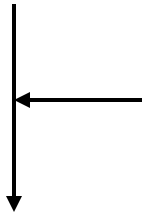
# Construction d'un graphe de contrôle

Un graphe de contrôle est un organigramme simplifié:

points de décision

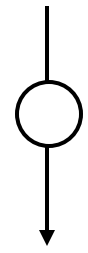
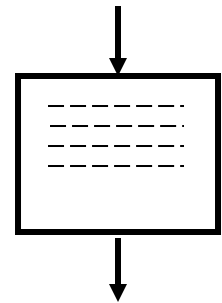


Jonctions



Blocs séquentiels

i.e suite d'instructions sans :  
entrées « latérales » (goto)  
rupture de séquence (break)



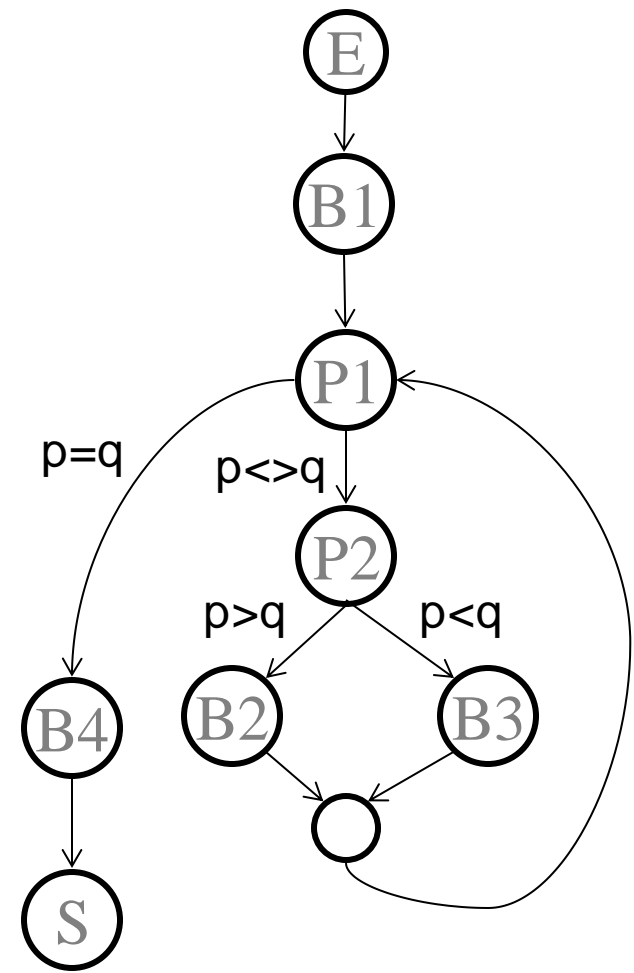
# Exemple de graphe de contrôle

## PGCD de 2 nombres

**Précondition:** p et q entiers naturels positifs

pgcd(p,q): integer is  
begin

read(p, q)	B1
while p<> q do	P1
if p > q	P2
then	
p := p-q	B2
else	
q:= q-p	B3
end -- if	
end -- while	
return p	B4
end	

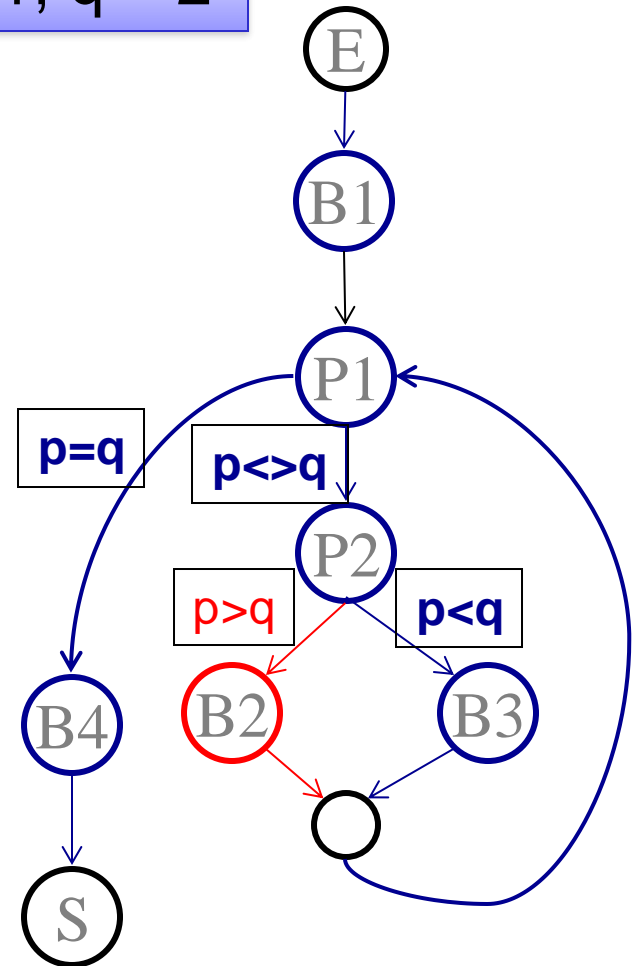




$$p = 1, q = 2$$

Analyser pour un **ensemble** de cas / valeurs de test la partie du graphe / code

- **Couverte** / non couverte
- **Testée** / non testée



# Notion de couverture (suite)

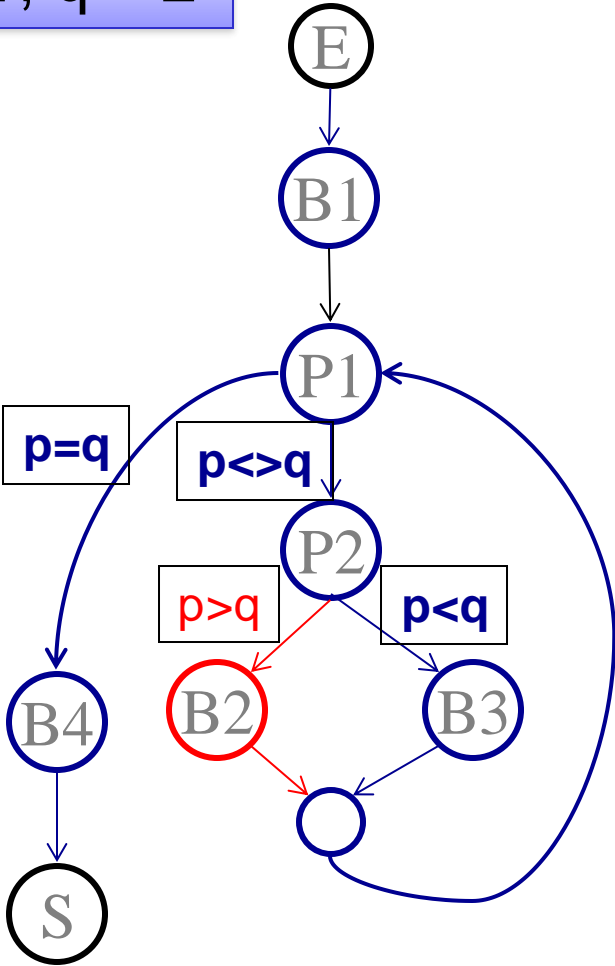
## PGCD de 2 nombres

**Précondition:** p et q entiers naturels positifs

pgcd(p,q): integer is  
begin

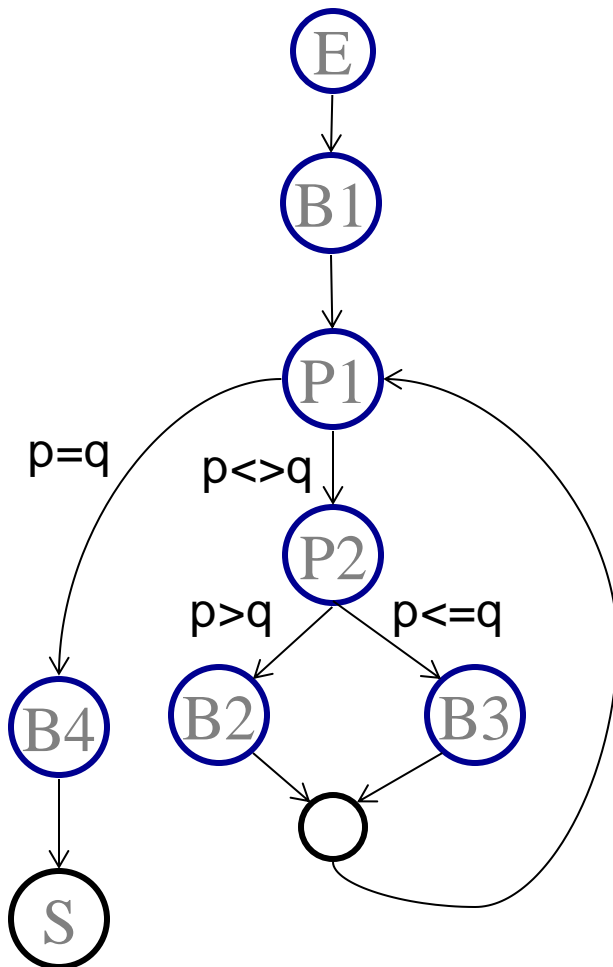
<b>read(p, q)</b>	<b>B1</b>
<b>while p&lt;&gt; q do</b>	<b>P1</b>
<b>if p &gt; q</b>	<b>P2</b>
<b>then</b>	
<b>p := p-q</b>	<b>B2</b>
<b>else</b>	
<b>q:= q-p</b>	<b>B3</b>
<b>end -- if</b>	
<b>end -- while</b>	
<b>return p</b>	<b>B4</b>
<b>end</b>	

p = 1, q = 2



- On définit plusieurs niveaux (objectifs) de couverture simples
  - **C0** : tester toutes les instructions (nœuds « fonction ») au moins une fois
  - **C1** : tester toutes les décisions possibles au moins une fois (ou toutes les branches)
  - **C-*i*** : tester toutes les branches (C1) mais en passant *i* fois dans chaque boucle : (i-chemins)
  - **C $\infty$**  : tester tous les chemins possibles
  
- Remarques :
  - **C $\infty$**  est virtuellement impossible
  - **C1** et **C0** sont différents
  - En pratique, le test d'un programme simple consiste à satisfaire **C0** et **C1**.
  - Dans le cas des systèmes critiques il faut aller plus loin que C0 et C1

# Exemples de séquences par rapport à une couverture



**Tous les nœuds (instructions) (C0):**

(E, B1, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, B4, S)

**Toutes les décisions (toutes les branches) (C1) :**

idem

**Toutes les 1-chemins (C-1) :**

idem + (E, B1, P1, B4, S)

**Tous les 2-chemins (C-2) :**

idem +

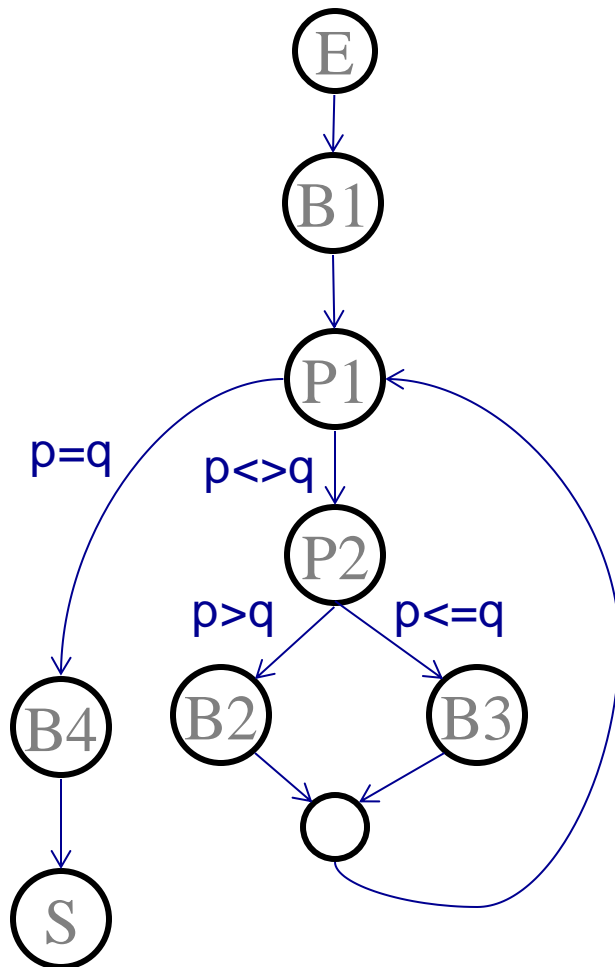
(E, B1, P1, P2, B2, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B2, P1, P2, B3, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B3, P1, B4, S)

# Exemples de séquences par rapport à une couverture



**Tous les nœuds (instructions) (C0):**

(E, B1, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, B4, S)

**Toutes les décisions (toutes les branches) (C1) :**

idem

**Toutes les 1-chemins (C-1) :**

idem + (E, B1, P1, B4, S)

**Tous les 2-chemins (C-2) :**

idem +

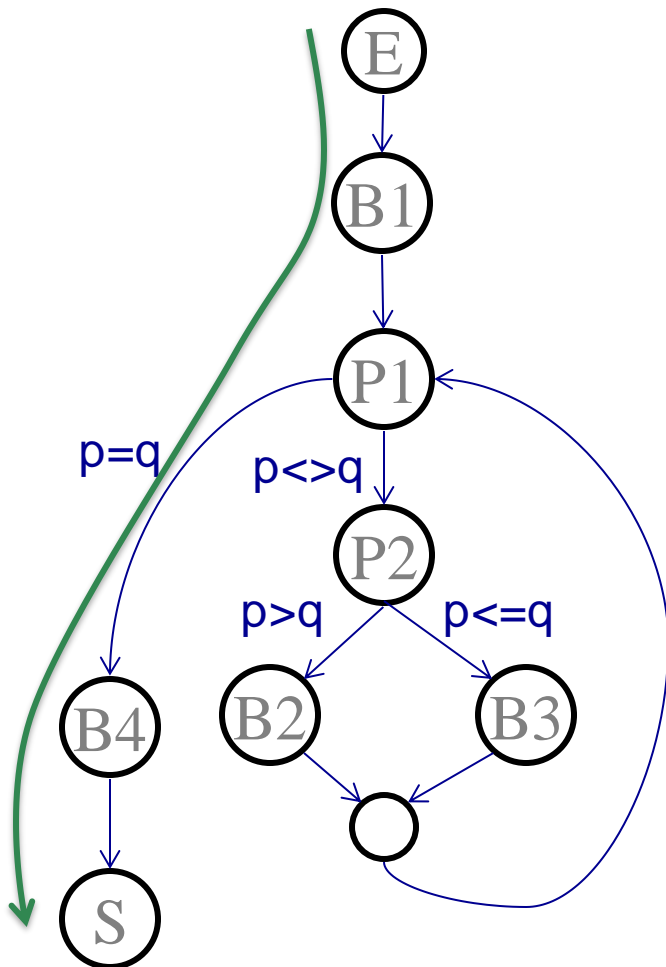
(E, B1, P1, P2, B2, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B2, P1, P2, B3, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B3, P1, B4, S)

# Exemples de séquences par rapport à une couverture



**Tous les nœuds (instructions) (C0):**

(E, B1, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, B4, S)

**Toutes les décisions (toutes les branches) (C1) :**

idem

**Toutes les 1-chemins (C-1) :**

idem + (E, B1, P1, B4, S)

**Tous les 2-chemins (C-2) :**

idem +

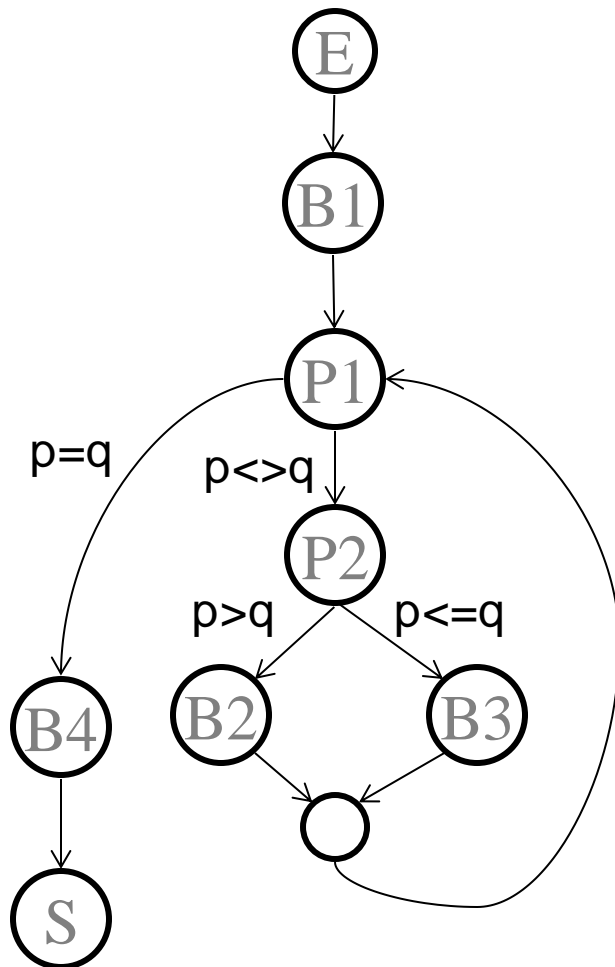
(E, B1, P1, P2, B2, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B2, P1, P2, B3, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B3, P1, B4, S)

# Exemples de séquences par rapport à une couverture



**Tous les nœuds (instructions) (C0):**

(E, B1, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B3, P1, B4, S)

**Toutes les décisions (toutes les branches) (C1) :**

idem

**Toutes les 1-chemins (C-1) :**

idem + (E, B1, P1, B4, S)

**Tous les 2-chemins (C-2) :**

idem +

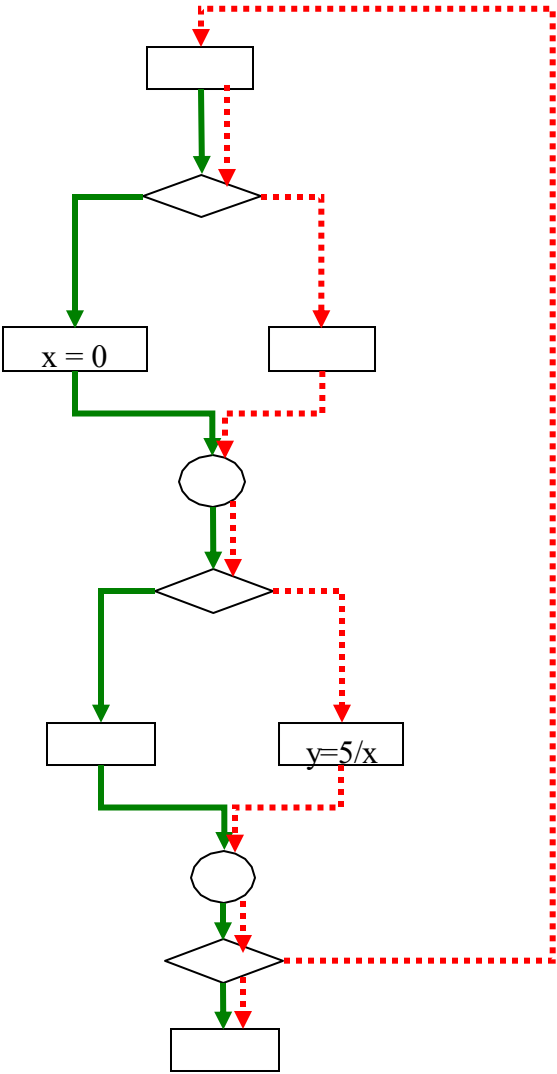
(E, B1, P1, P2, B2, P1, P2, B2, P1, B4, S)

(E, B1, P1, P2, B2, P1, P2, B3, P1, B4, S)

(E, B1, P1, P2, B3, P1, P2, B2, P1, B4, S)

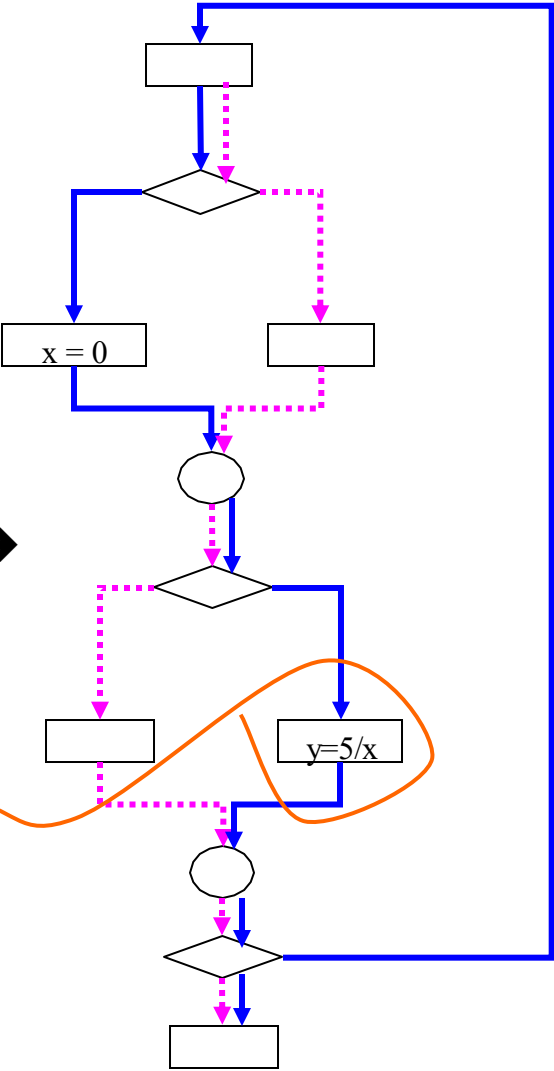
(E, B1, P1, P2, B3, P1, P2, B3, P1, B4, S)

# La couverture C1 n'est souvent pas suffisante



← Couverture C1 à 100%  
mais sans jamais  
exécuter les chemins de droite →

On ne verra donc pas le  
problème de la division  
par zéro ici  
(quand on emprunte le  
chemin bleu)





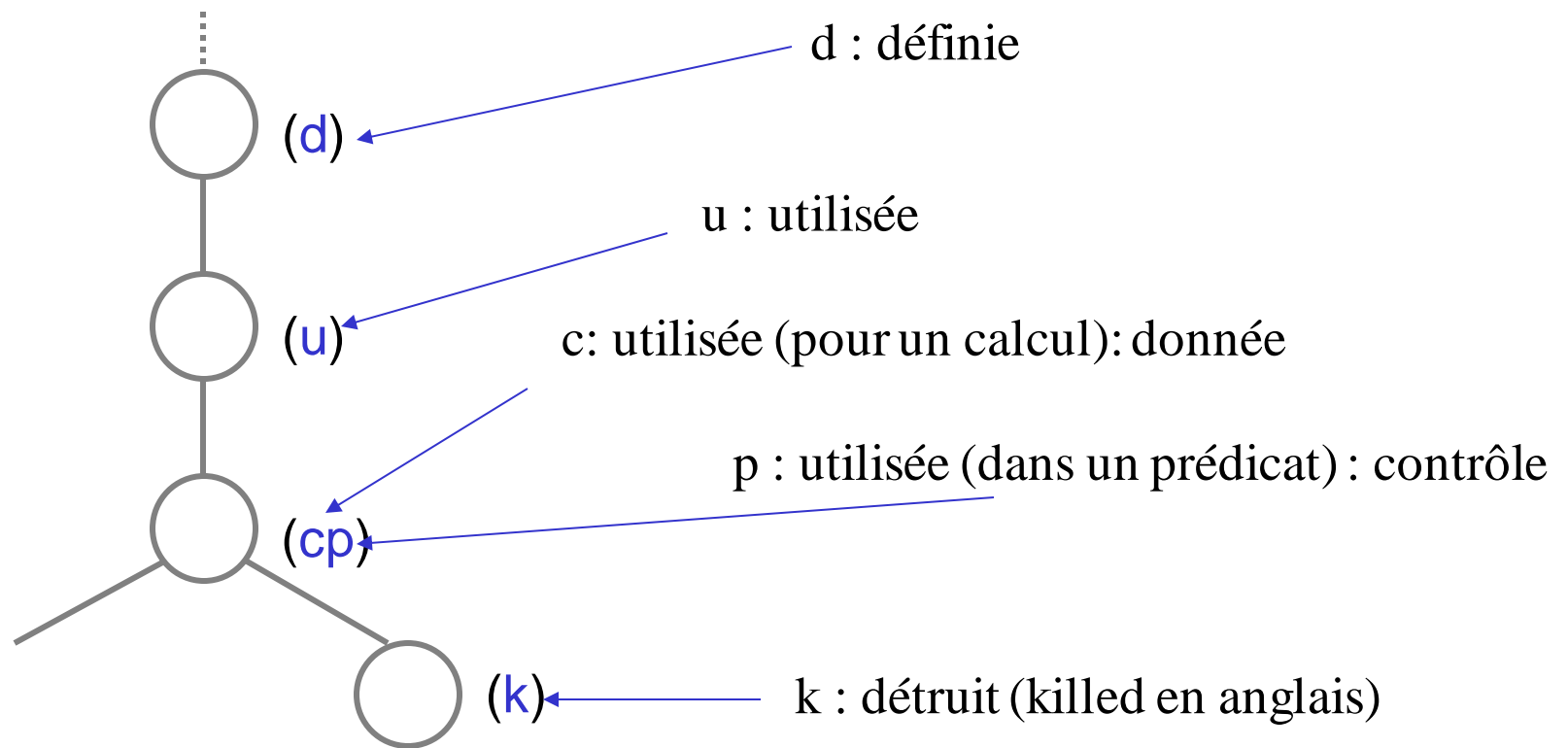
## **Amélioration des critères de couvertures de base (C0 et C1)**

- Le passé dans une exécution (chemin) influe fortement le futur
- Les défauts complexes proviennent d'enchainements particuliers d'actions
- On peut suivre ces enchainements en se focalisant sur l'accès aux données avec la construction d'un graphe de flot de données
- On peut suivre ces enchainements en se focalisant sur les décisions et les branchements en définissant des critères de couverture spécifiques

- Les données modifient fortement le comportement du programme
- On peut améliorer la qualité des tests basés sur le graphe de contrôle en choisissant des chemins dans le graphe avec des critères d'exploration de « l'histoire » des données
- C'est une méthode de test structurel plus sévère que la couverture de toutes les alternatives et/ou de tous les arcs. Elle permet de couvrir l'intervalle entre le test de tous les chemins ( $C_{\infty}$ ) et ceux de toutes les alternatives ( $C_1$ ) et de tous les sommets ( $C_0$ ).
- Typiquement, elle permet de vérifier qu'une variable a bien été initialisée avant d'être utilisée.

# Test partir du flot de données : principes

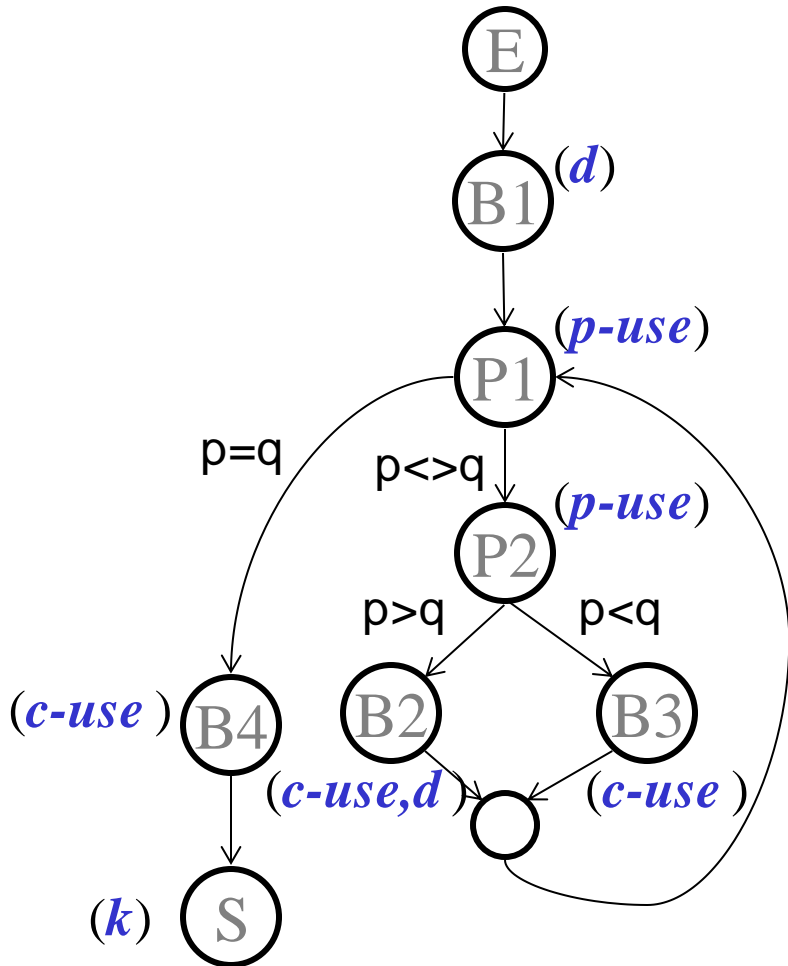
- Pour chaque variable, on annote, le graphe de flots de contrôle avec l'état de la variable (fichier y compris)



# Annotation du graphe de contrôle : exemple

*Par rapport à p*

```
begin
  read(p, q)           B1
  while p<>q do         P1
    if p > q            P2
      then
        p := p-q       B2
      else
        q:= q-p        B3
      end -- if
    end -- while
  return p             B4
end
```



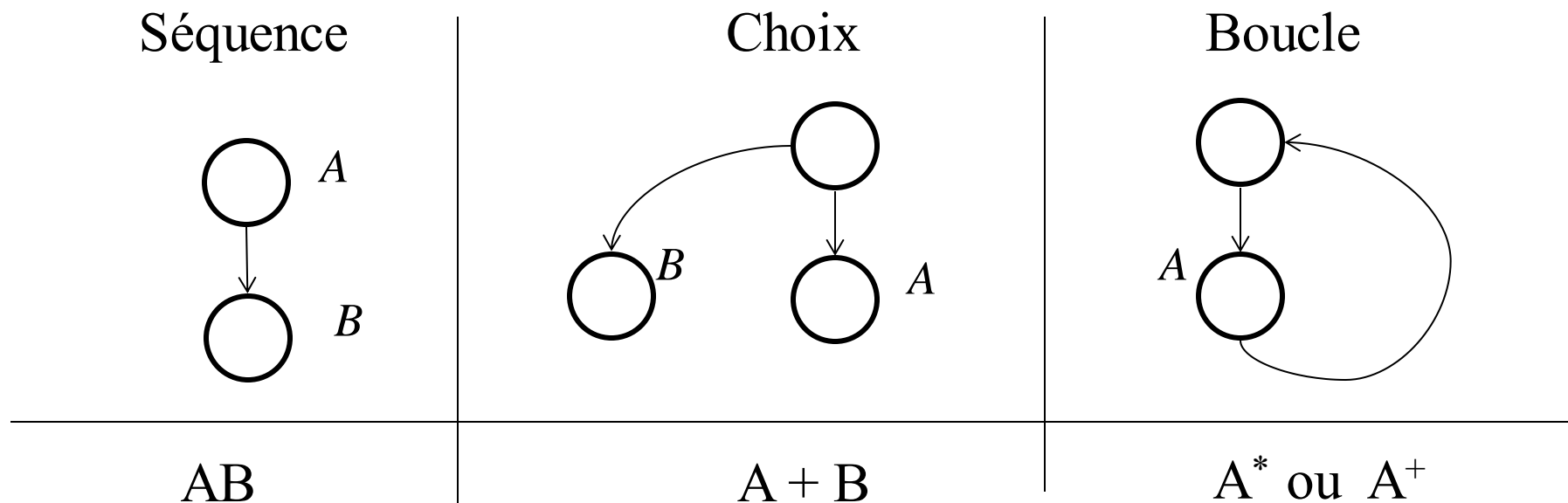
# Utilisation statique du flot de données

- La succession des états de chaque variable donne des expressions régulières  $(ud)^*$ ,  $(dduru + pu)$
- L'étude de ces expressions permet de détecter des cas pathologiques.
- Les séquences d'erreur et d'alarme sont composées de 2 caractères
  - séquences d'erreur:
    - **ku** référence d'une variable indéfinie
      - se décline en **kc** et **kp** (référence pour calcul ou pour test)
  - Séquences d'alarme:
    - **dd** double définition d'une variable
    - **dk** destruction de la valeur d'une variable

# Les 9 séquences typiques

- **dd** : probablement pas de problème ?
- **dk** : probablement une erreur
- **du**: le cas normal, définition, utilisation
- **kd**: autre cas normal, destruction puis redéfinition
- **kk**: bizarre...
- **ku**: une erreur certainement
- **ud**: pas une erreur si redéfinition
- **uk**: cas normal
- **uu**: cas normal

# Construction des expressions régulières



- Langage reconnu par l'automate correspondant au graphe de contrôle
- Ce langage peut se simplifier avec le Th. de *Huang* (1979) :
  - Soit  $T$  une chaîne de 2 caractères ; si  $T$  est une sous chaîne de  $AB^nC$  (quelque soit  $n>0$ ) alors  $T$  est une sous-chaîne de  $AB^2C$
  - Corolaire : On peut substituer à  $X^*$  la chaîne  $1+X^2$  (suffisant pour détecter si une séquence de caractère est présente ou pas dans  $X^*$ ) et  $X^2$  à  $X^+$



## Exemple : il y a t-il une erreur possible ?

```
Integer x, z;  
get(z);  
for i:=0 to z loop  
    x := A(i);  
end loop;  
if x = 0 then ....
```

- L'expression régulière correspondant à l'utilisation de la variable  $x$  est :  
 $kd^*p$  qui se simplifie en  $k(1+d^2)p = (kp + kd^2p)$
- L'expression (**kp** + k d<sup>2</sup>p) montre une erreur potentielle

*Utilisé dans de nombreux outils (compilateurs)*

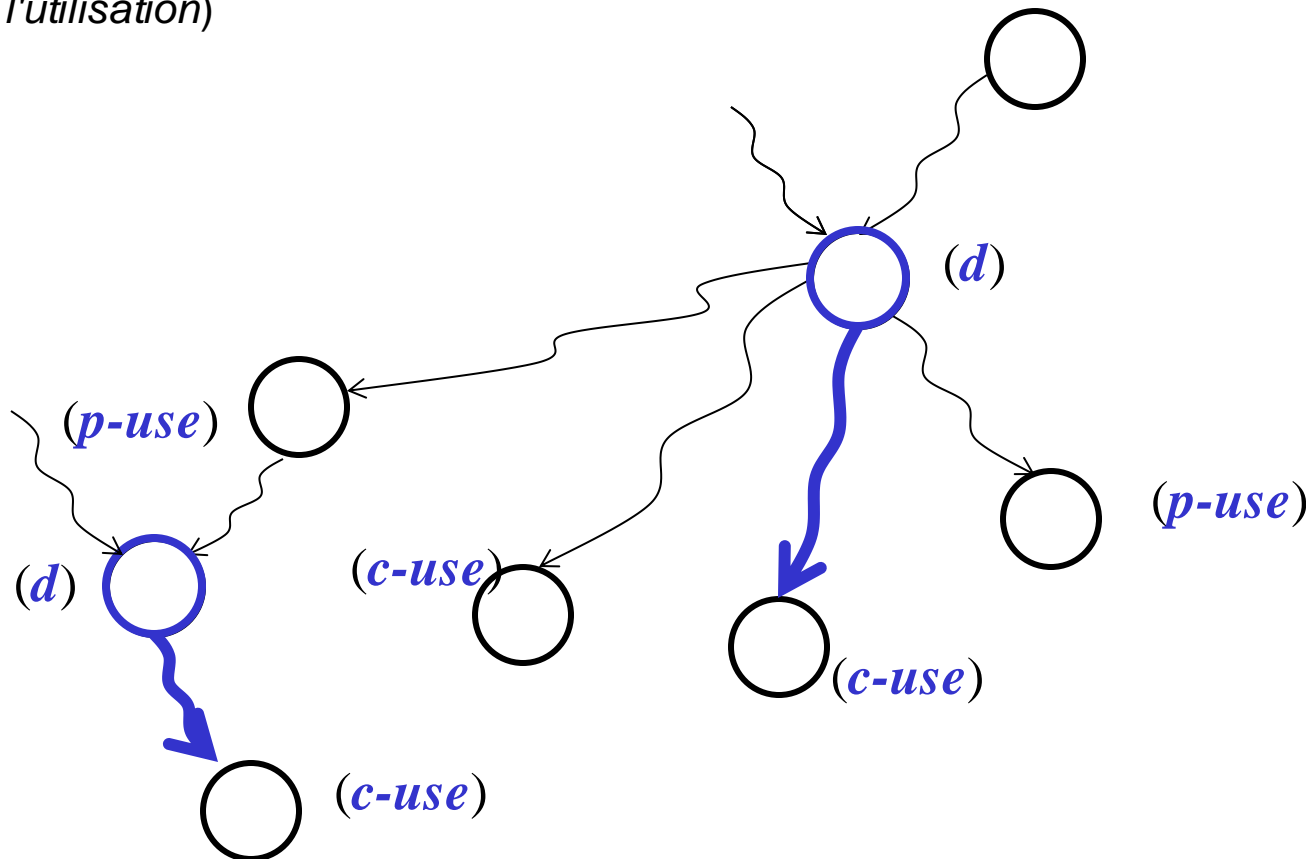
*Sensibles aux « faux » warnings*

- Définir les chemins à parcourir pour suivre l'évolution de ces états et découvrir par tests « dynamiques » des erreurs liées à l'utilisation des variables
- Choix des chemins en fonction de critères de couverture des chemins liant les définitions et les utilisations (C pour mettre en avant les calculs et P pour mettre en avant le contrôle)

# Choix de chemins « flots de données » : AD

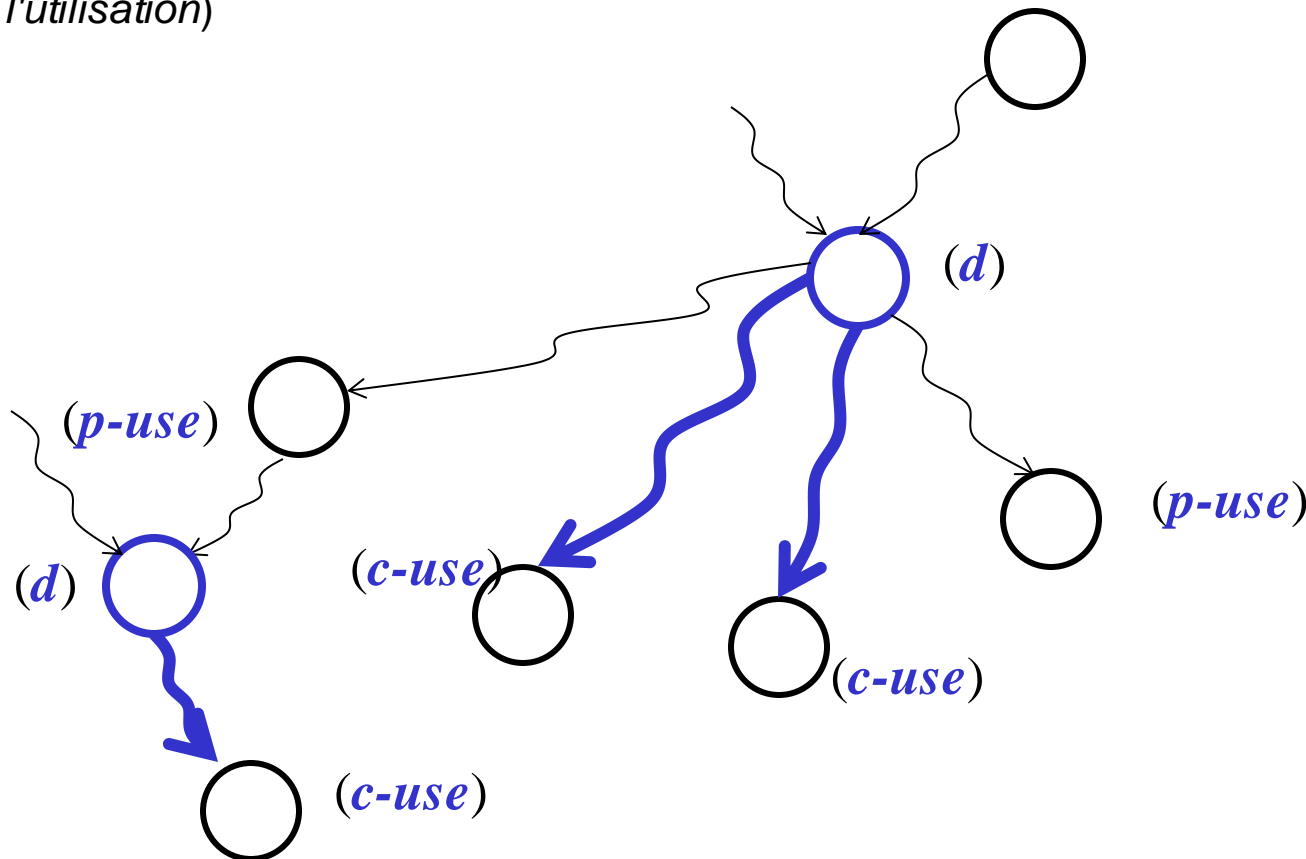
## ■ Chemins **AD** : All Definitions

- Les chemins qui couvrent, pour chaque variable chaque définition avec **au moins une** utilisation (C ou P) après la définition (*sans nouvelle définition avant l'utilisation*)



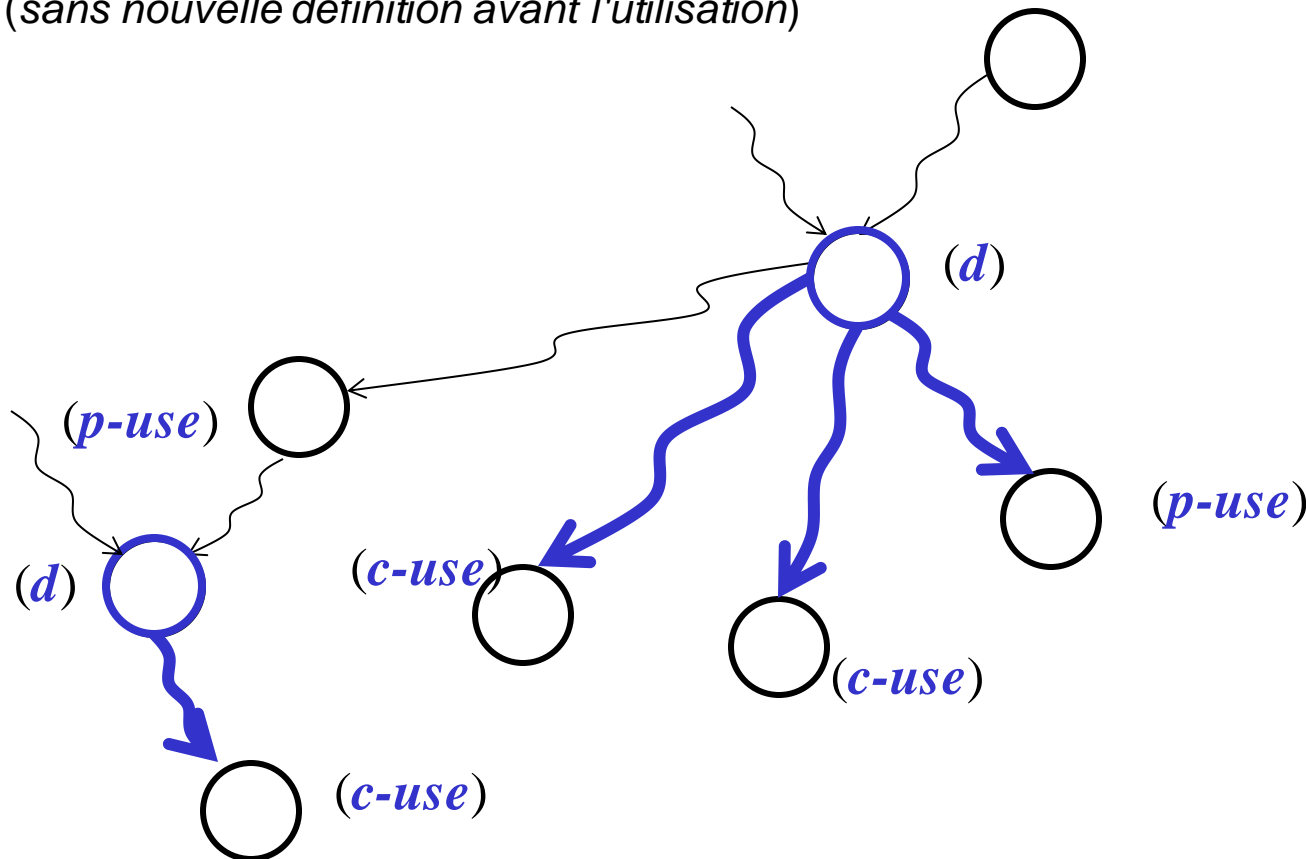
## ■ Chemins **ACU** : All Computation Uses

- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** les utilisations de calcul (C) après la définition (*sans nouvelle définition avant l'utilisation*)



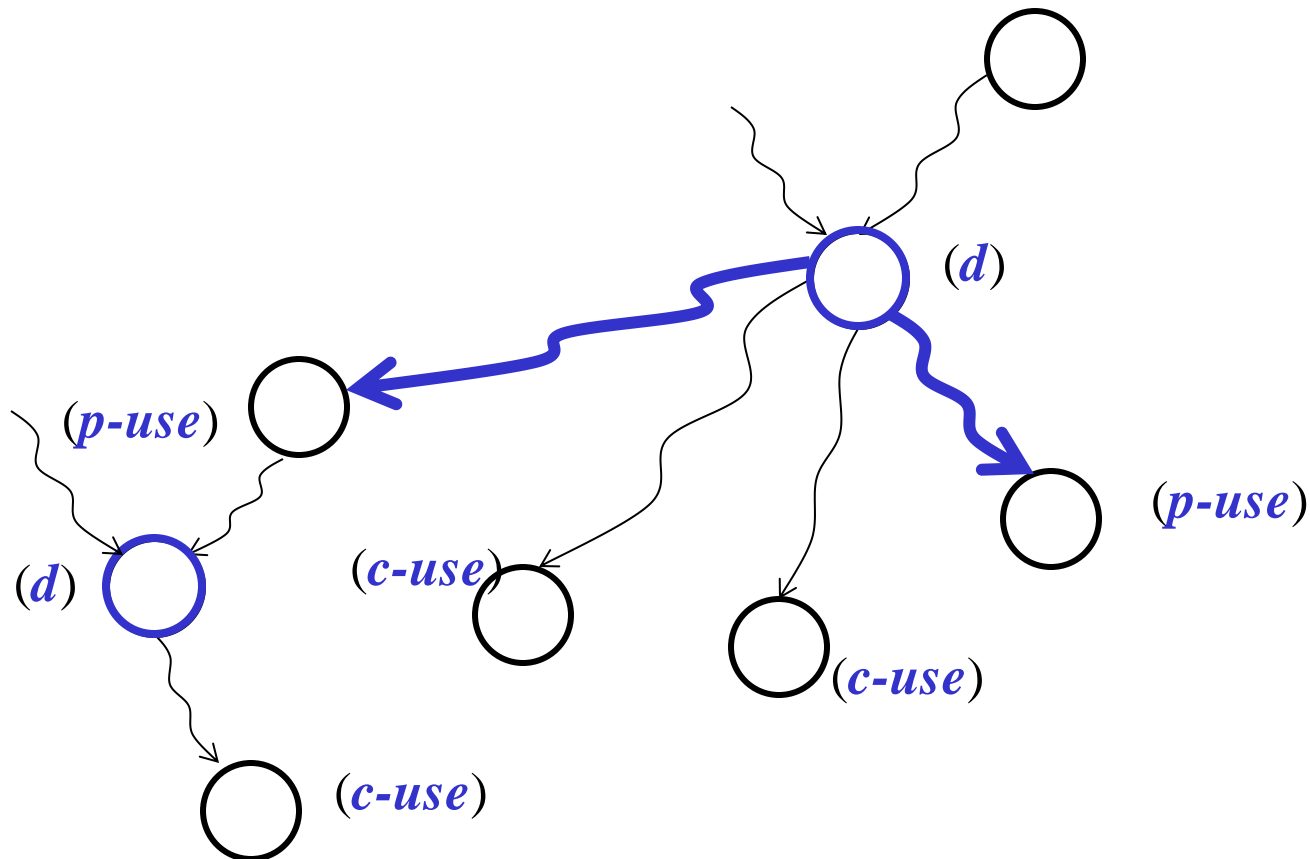
## ■ Chemins **ACU+P** : All Computation Uses + Predicate

- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** les utilisations de calcul (C) après la définition **et au moins une** utilisation prédicat (P) (sans nouvelle définition avant l'utilisation)



## ■ Chemins **APU** : All Predicate Uses

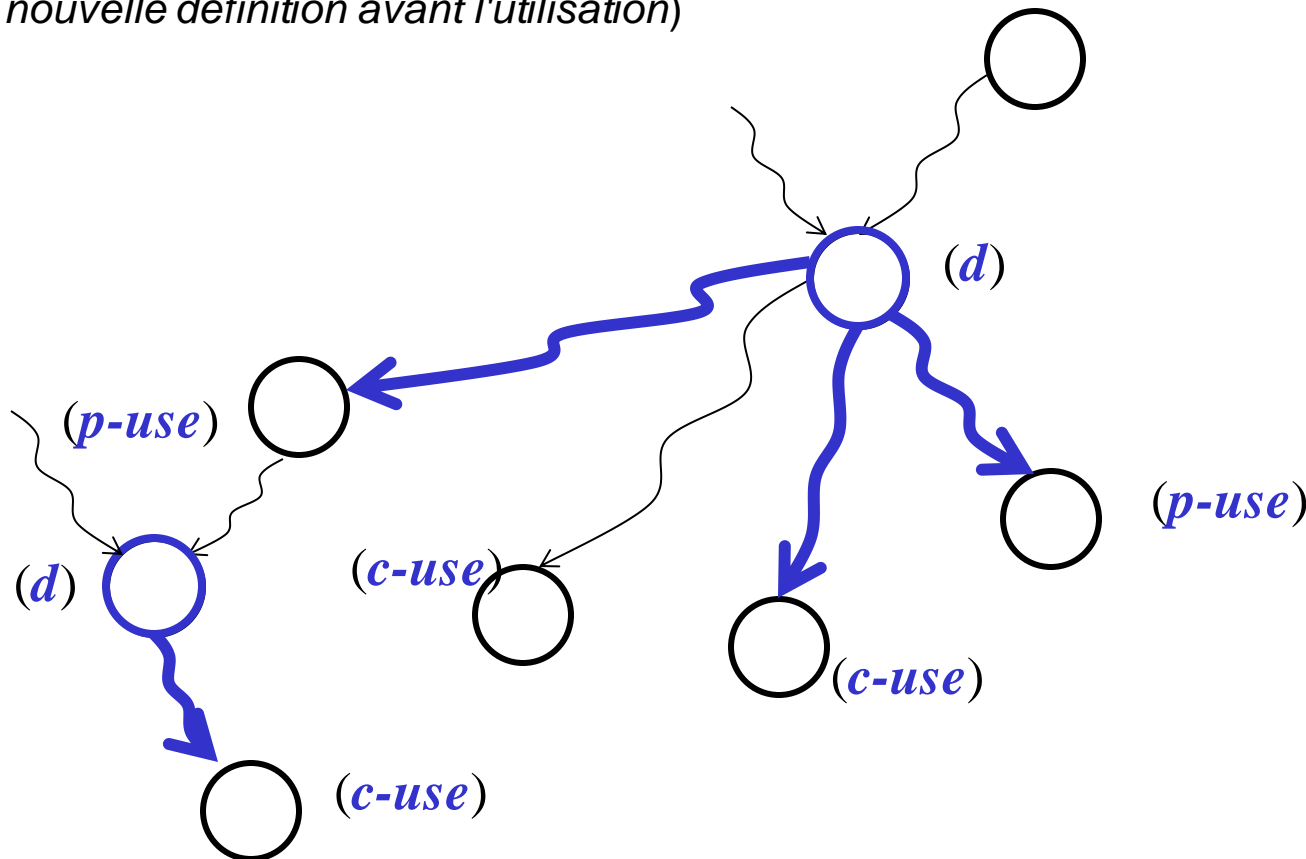
- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** les utilisations de prédicats (P) (*sans nouvelle définition avant l'utilisation*)



# Choix de chemins « flots de données » : APU + C

## ■ Chemins **APU + C** : All Predicate Uses + Computation

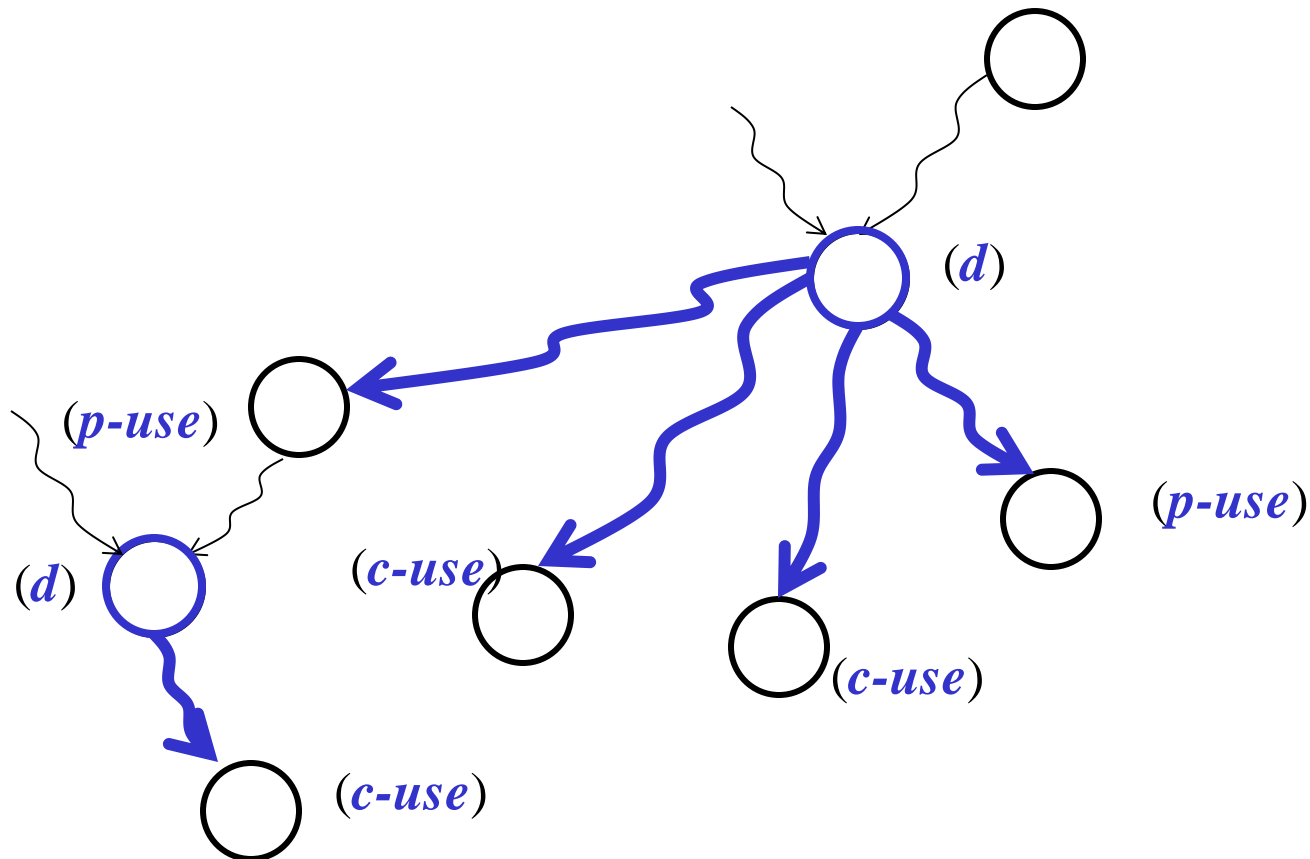
- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** les utilisations de prédicats (P) **et au moins une** utilisation de calcul (C) (*sans nouvelle définition avant l'utilisation*)



# Choix de chemins « flots de données » : AU

## ■ Chemins **AU** : All Uses

- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** ses utilisations (P+C) (*sans nouvelle définition avant l'utilisation*)

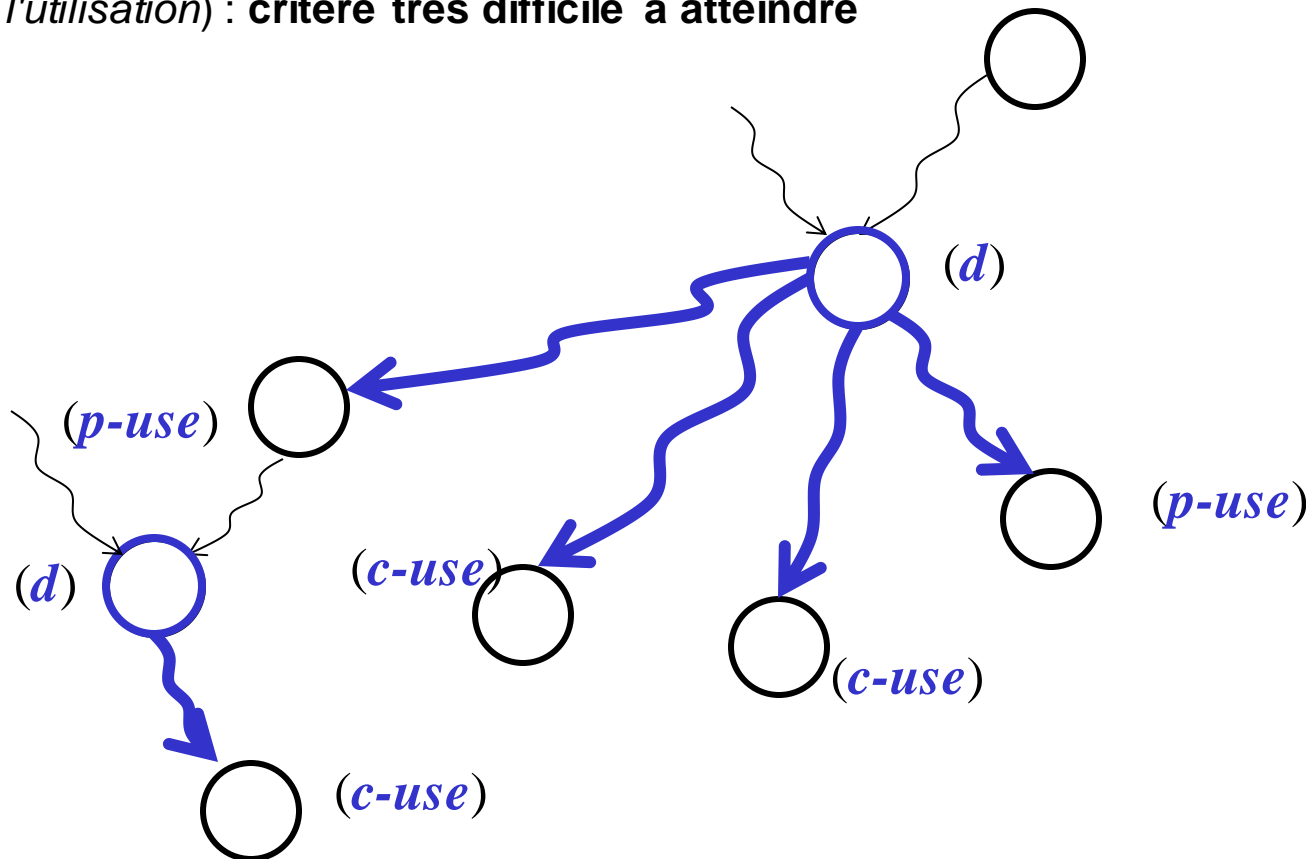




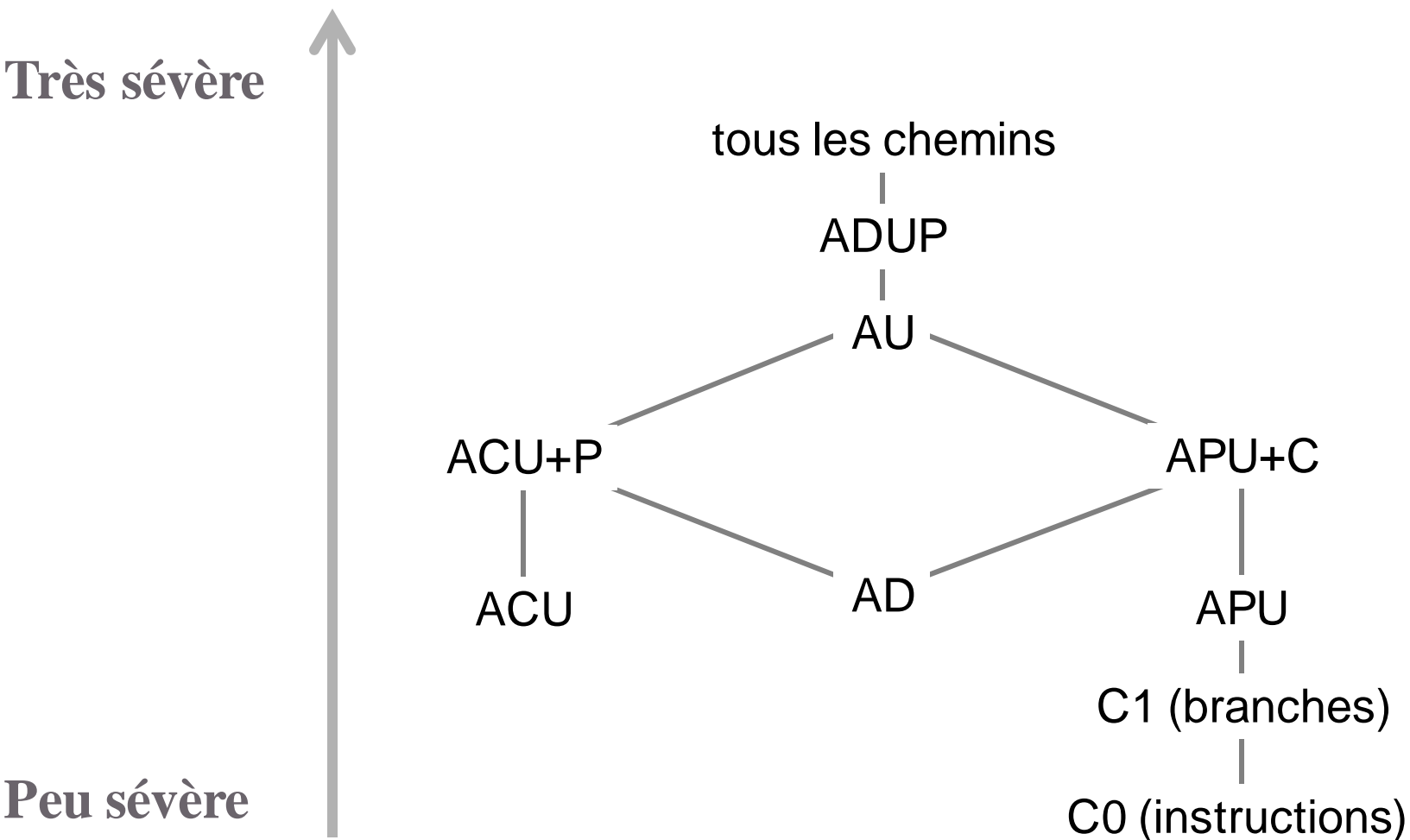
# Choix de chemins « flots de données » : ADUP

## ■ Chemins **ADUP** : All Definitions Uses Paths

- Les chemins qui couvrent, pour chaque variable chaque définition avec **toutes** ses utilisations (P+C) (**sans restrictions** sur une potentielle nouvelle définition avant l'utilisation) : **critère très difficile à atteindre**



# Sévérité des critères de couverture et flot de données



## Générer des cas de test avec le flot de données

- Une fois un ensemble de chemins choisi il faut générer des cas de tests pour parcourir ces chemins : **problème difficile** !
- Peut être fait manuellement en essayant de « télécommander » les branchements ce qui demande de grands efforts
- Peut s'automatiser avec la notion d'exécution symbolique : technique statique très puissante mais complexe à mettre en œuvre

## Tests basés sur les flots de données : conclusion

- Les tests "flot de données" permettent une recherche efficace des erreurs
- Ils permettent d'identifier le "cône d'influence" (la tranche) du programme relatif à une instance de la variable x (toutes les instructions qui affectent la variable choisie)
- Dans le développement, cette technique permet d'enlever de ce cône les instructions dont on est sûr du comportement. Et ceci jusqu'à trouver le défaut !
- Les symboles d, k, u...et d'autres peuvent être associés à d'autres éléments (i/o files, ressources, etc.)
- *MAIS ils sont complexes à mettre en œuvre → Utilisés principalement dans les méthodes formelles*

- Le passé dans un cheminement a une forte importance sur le futur
- Les défauts complexes proviennent d'enchainements particuliers d'actions
- On peut suivre ces enchainements en se focalisant sur l'accès aux données avec la construction d'un graphe de flot de données
- On peut suivre ces enchainements en se focalisant sur les décisions et les branchements en définissant des critères de couverture spécifiques

## Couverture « Toutes les décisions » (C1)

```
int f1(int a, int b, int c) {  
  
    if ((a>0) && ( (b>0) || (c>0) )) {  
        // I1  
    }  
    // I2 erreur si (c ≤ 0) ou (b ≤ 0)  
}
```

a = 1, b = 1, c = 1

C0 OK MAIS défaut NON détecté

Viser C1 permet de détecter le défaut

a = 1, b = -1, c = 0

## Couverture « Toutes les conditions » (C1-p)

```
int f1(int a, int b, int c) {  
    if ((a>0) && ( (b>0) || (c>0) )) {  
        // I1  
    }  
    // I2 erreur si (a ≤ 0)  
}
```

a = 1, b = 1, c = 1

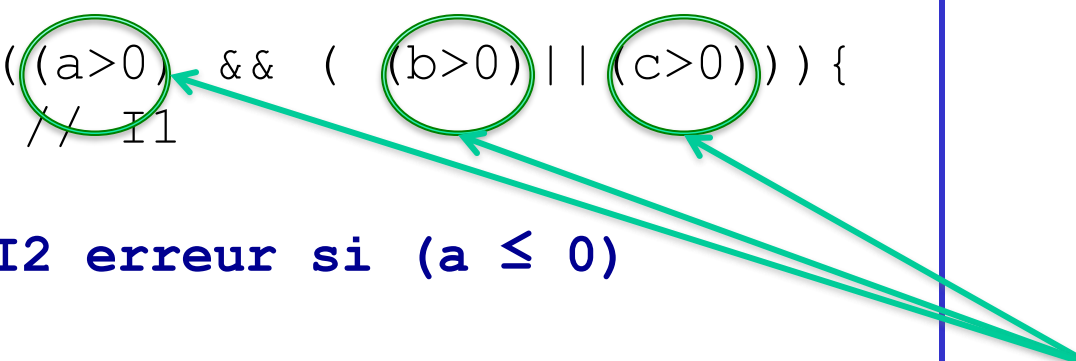
a = 1, b = -1, c = 0

C1 OK MAIS défaut NON détecté

Viser C1 n'est pas suffisant → Viser toutes les Conditions

## Couverture « Toutes les conditions » (C1-p) (suite)

```
int f1(int a, int b, int c) {  
    if ((a>0) && ((b>0) || (c>0))) {  
        // I1  
    }  
    // I2 erreur si (a ≤ 0)  
}
```



Conditions

a = 1, b = -1, c = 0

a = 0, b = 1, c = 1

Viser toutes les Conditions permet de détecter le défaut



## Couverture « Multiples Conditions » (MCC)

```
int f1(int a, int b, int c) {  
  
    if ((a>0) && ( (b>0) || (c>0) )) {  
        // I1  
    }  
    // I2 erreur si (b≤0) et (a>0) et (c>0)  
}
```

a = 1, b = 1, c = 1

a = 1, b = -1, c = 0

a = 0, b = 1, c = 1

C0, C1, Toutes les conditions OK  
**MAIS** défaut non détecté

→ Viser toutes les combinaisons de conditions ( $2^n$  combinaisons !)

## Couverture « Modified Conditions / Decision Coverage » (MC / DC)

```
int f1(int a, int b, int c) {  
  
    if ((a>0) && ( (b>0) || (c>0) )) {  
        // I1  
    }  
    // I2 erreur si ...  
}
```

Dès que (a=0) la décision est fausse !

Réduite la combinatoire MCC en se focalisant sur les conditions « importantes », i.e. celle qui ont un impact sur les décisions

## Couverture MC / DC (norme DO 178 C)

- every statement in the program has been invoked at least once. C0
- every point of entry and exit in the program has been invoked at least once.
- every control statement (i.e., branchpoint) in the program has taken all possible outcomes (i.e., branches) at least once. C1
- every nonconstant Boolean expression in the program has evaluated to both a true and a false result. C1-p
- every nonconstant condition in a Boolean expression in the program has evaluated to both a true and a false result. C1-p
- every nonconstant condition in a Boolean expression in the program has been shown to **independently** affect that expression's outcome. MC / DC

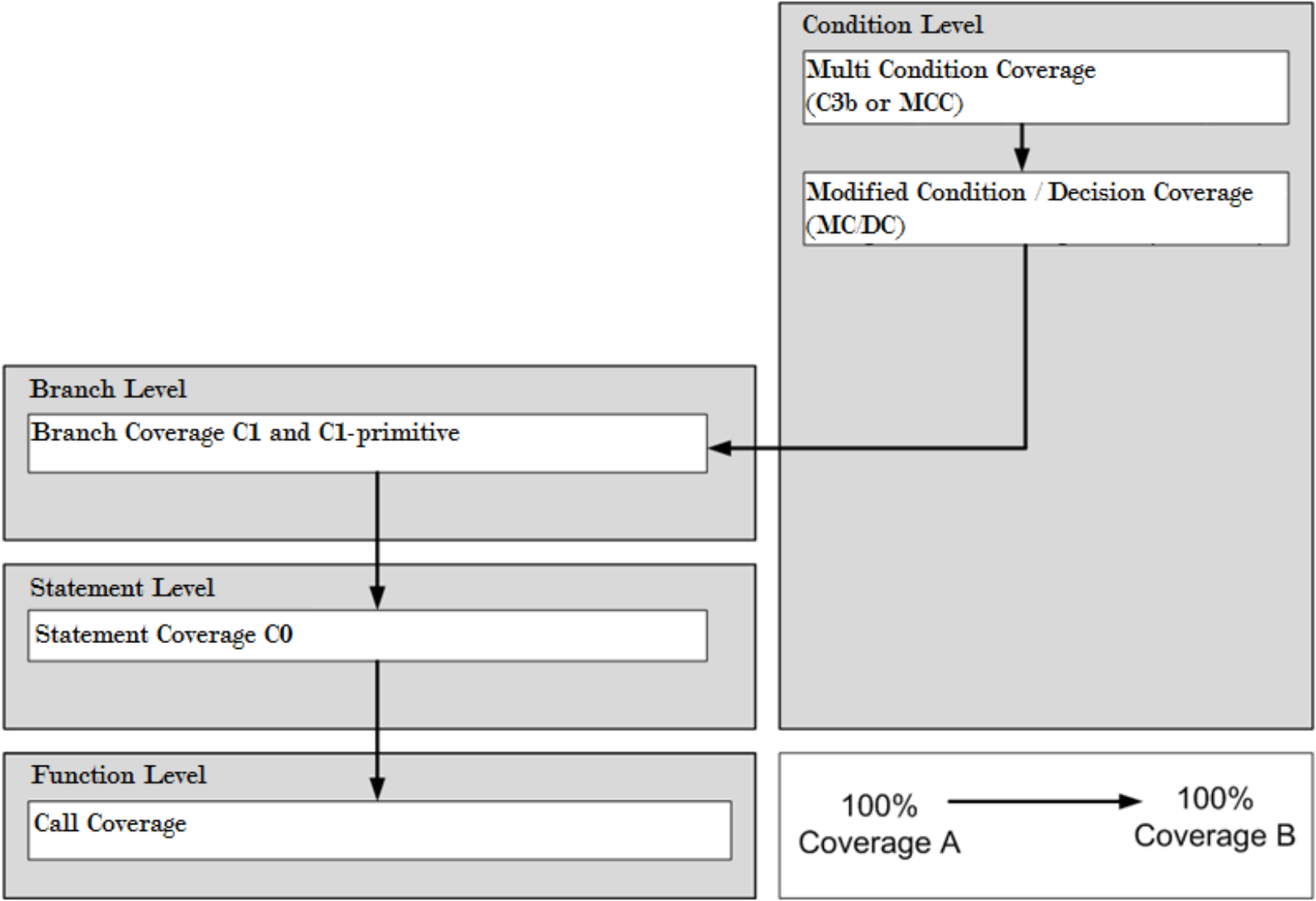
*Différentes interprétations (ex: (A and B) or (A and C) )*

# Couverture MC / DC : exemple

```
int f1(int a, int b, int c){  
  
    if ((a>0) && ( (b>0) || (c>0) )) {  
        // I1  
    }  
    // I2 erreur si ...  
}
```

N	<b>a&gt;0</b>	<b>b&gt;0</b>	<b>c&gt;0</b>	(a>0) && [ (b>0)    (c>0) ]
1	T	T	T	T
<b>2</b>	<b>T</b>	<b>F</b>	<b>T</b>	<b>T</b>
<b>3</b>	<b>T</b>	<b>T</b>	<b>F</b>	<b>T</b>
<b>4</b>	<b>T</b>	<b>F</b>	<b>F</b>	<b>F</b>
5	F	T	T	F
<b>6</b>	<b>F</b>	<b>T</b>	<b>F</b>	<b>F</b>
7	F	F	T	F
8	F	F	F	F

# Hiérarchie des couvertures basées sur les décisions / conditions



## Couvertures basées sur les décisions / conditions : bilan

- La couverture C0 est simple à obtenir mais ne garantit pas grand chose
- Les couvertures C1 et C1-p apportent plus d'assurance mais restent faibles
- La couverture MCC est la plupart du temps inenvisageable
- La couverture MC / DC offre un bon compromis entre qualité de la couverture et simplicité de mise en œuvre
- AUCUNE couverture ne remplacera la réflexion

## Tests boîte blanche : conclusion

- Technique efficace lorsque l'on dispose du code (ou des dépendances entre modules ou classes)
- Permet de détecter des erreurs non détectées par les tests boîte noire
- Les techniques permettant de réduire la combinatoire des tests boîte noire (partitionnement, limites, tableaux) peuvent être employées également dans les tests boîtes blanches
- **Les critères de couverture peuvent être (r)éutilisés pour mesurer la qualité et l'avancée de tests boîte noire**
- **Les tests statiques du code permettent d'obtenir une très bonne garantie pour des logiciels « sensibles » (proche de la preuve)**