

Test structurel ou boîte blanche

Plan

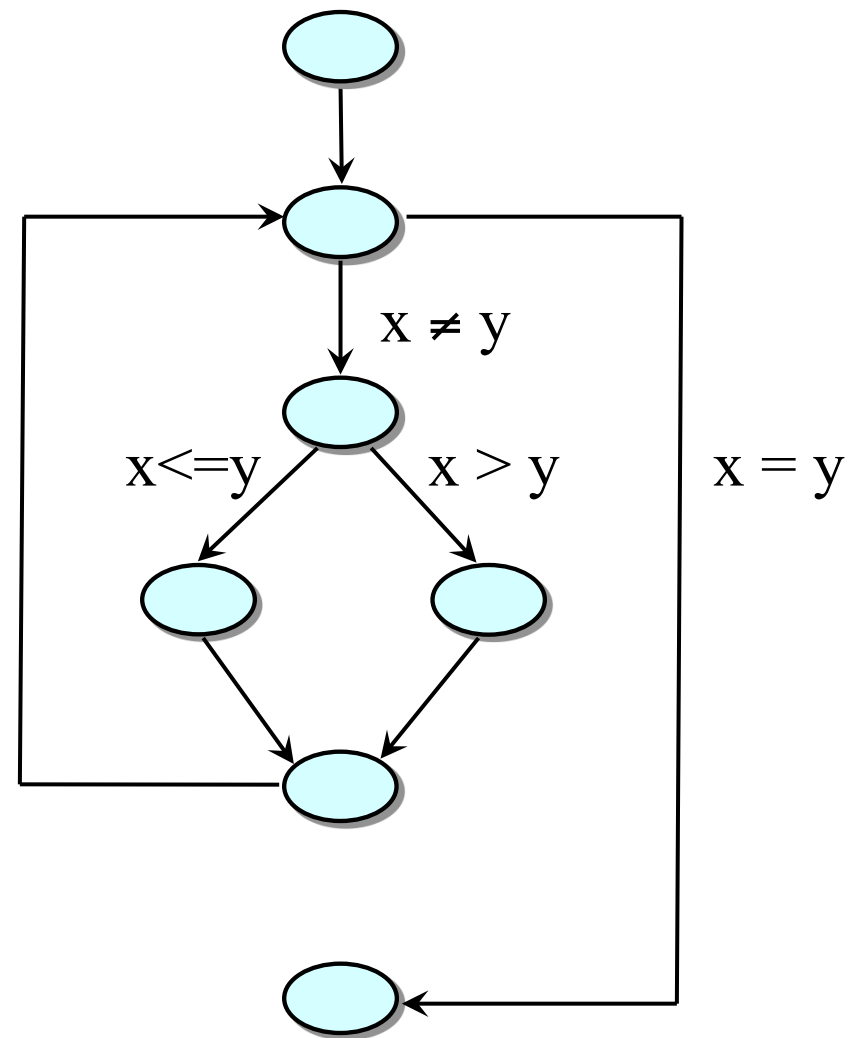
- ❑ Couverture du flot de contrôle :
 - instruction, arête, condition, couverture du chemin.
- ❑ Couverture du flot de données :
 - définitions-usages des données.
- ❑ Analyse de la couverture des données.
- ❑ Test de mutation.

Définitions élémentaires

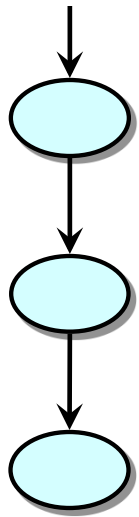
- ❑ Un graphe de flot de contrôle est une représentation sous forme de graphe orienté de tous les chemins possibles dans un programme.
- ❑ Les nœuds sont des blocs d'instructions séquentiels.
- ❑ Les arêtes sont des transferts de contrôle.
- ❑ Les arêtes peuvent être étiquetées avec un attribut représentant la condition du transfert de contrôle.

Graphe de flot de contrôle

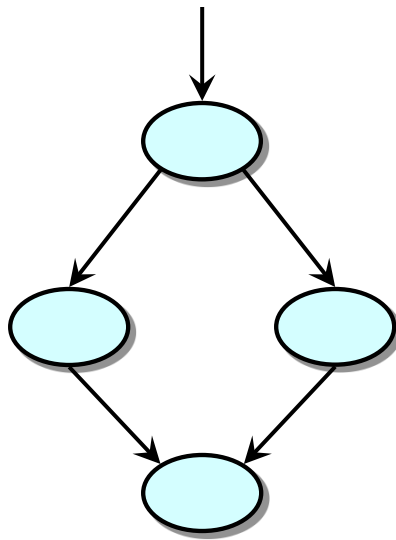
```
read(x);  
read(y);  
while  $x \neq y$  loop  
    if  $x > y$  then  
         $x := x - y;$   
    else  
         $y := y - x;$   
    end if;  
end loop;  
 $\text{gcd} := x;$ 
```



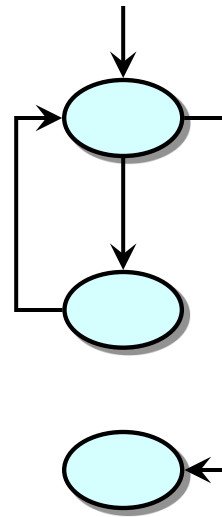
Principes du CFG



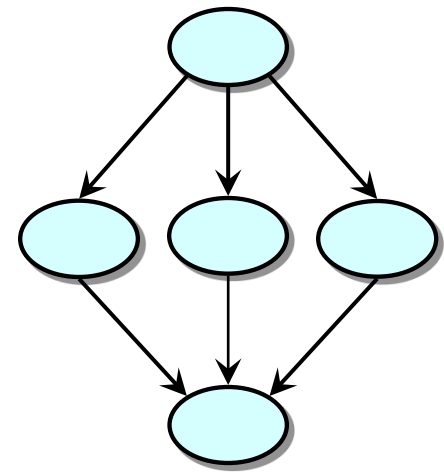
Séquence



Si-Alors-Sinon
(If-Then-Else)

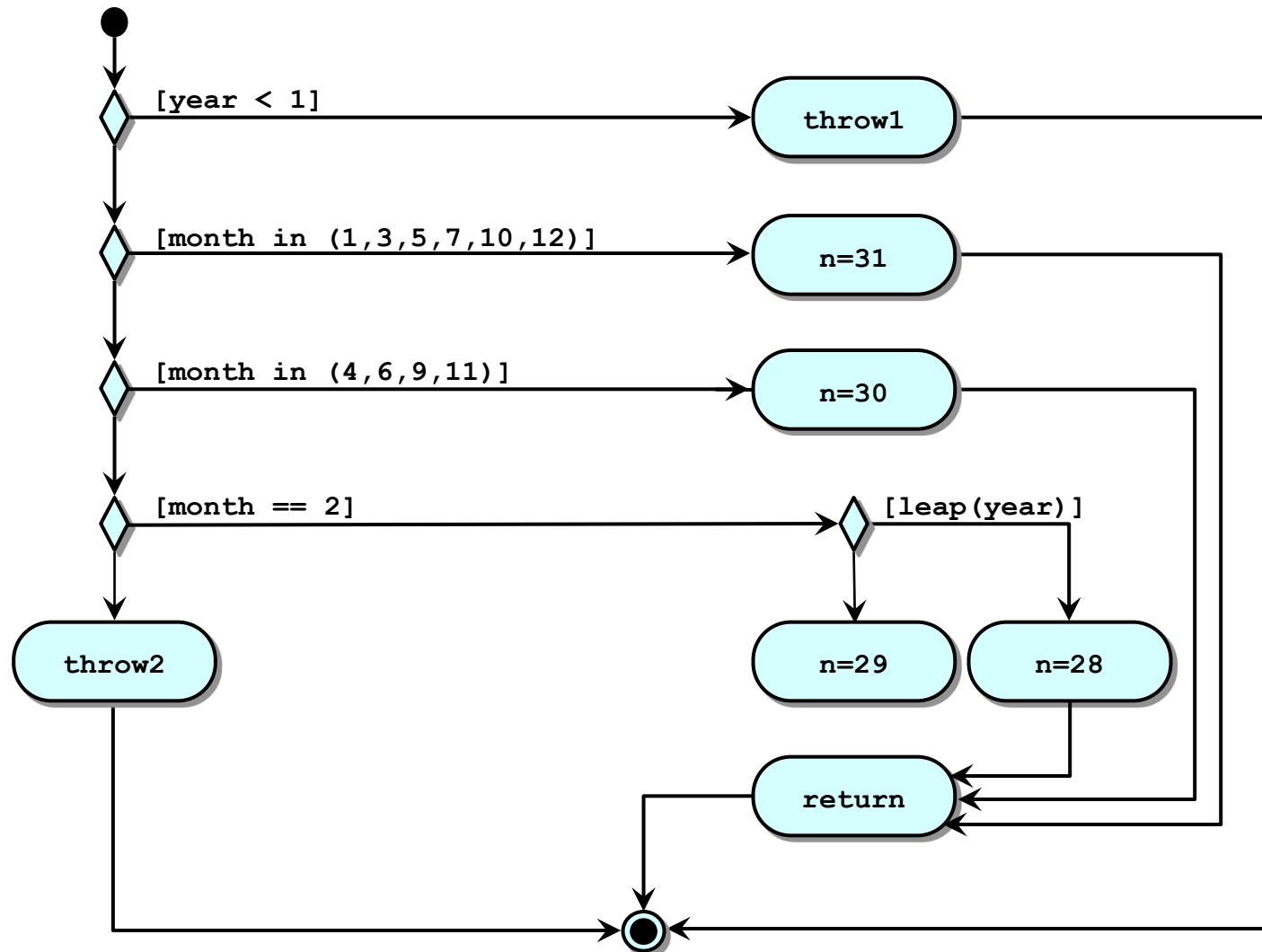


Boucle tant que
(While loop)



Commutateur
(Switch)

Alternative: Diagramme d'activité UML



Instruction/Couverture des nœuds

❑ **Couverture d'instruction :**

les anomalies ne peuvent pas être découvertes si les parties les contenant ne sont pas exécutées.

❑ Équivaut à couvrir tous les nœuds dans le CFG.

❑ En général, plusieurs entrées exécutent les mêmes instructions → une question importante en pratique:

Pouvons-nous minimiser les cas de test ?

Incomplétude

- ❑ La couverture des instructions peut mener à l'incomplétude.

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

-----


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


Un $x < 0$ couvre toutes les instructions.
Mais le cas $x \geq 0$ n'est pas pris en compte.
Le code implicite (en italique et rouge) n'est pas couvert.
Ne rien faire pour le cas $x \geq 0$ peut s'avérer faux, voire dangereux et devrait être testé.

Couverture des arêtes

- ❑ Utiliser la structure de programme, le graphe de flot de contrôle (CFG).
- ❑ **Critère de la couverture des arêtes :**
sélectionner un ensemble de tests T de telle façon qu'en exécutant P pour chaque cas de test présent dans T, chaque arête du graphe du flot de contrôle P ait été traversée au moins une fois.
- ❑ Exercer toutes les conditions qui gouvernent le flot de contrôle du programme avec des valeurs vraies et fausses.

Limite du Méthode: Exemple de code

```
counter:= 0;
found := false;
if number_of_items  $\neq$  0 then counter :=1;
    while (not found) and counter  number_of_items loop
        if table(counter) = desired_element then
            found := true;
        end if;
        counter := counter + 1;
    end loop;
end if;
if found then write ("the desired element exists in
                        the table");
else write ("the desired element does not exists in
            the table");
end if;
```



Jeu de tests

- ❑ Nous choisissons un jeu de tests avec une table de 0 item et une table de 3 items, le second étant celui désiré ($|T| = 2$).
- ❑ Pour le second cas de tests, le corps de la boucle est exécuté deux fois, quand la branche **then** est exécutée.
- ❑ Le critère de la couverture des arêtes est vérifié et l'erreur n'a pas été découverte par le jeu de tests.
- ❑ **Toutes** les valeurs possibles des *éléments de la condition* de la **boucle while loop** n'ont pas été exercées.

Couverture des conditions

- ❑ Plus ample renforcement de la couverture des arêtes.
- ❑ **Critère de la couverture des conditions :**
sélectionner un jeu de tests T de telle façon que, en exécutant P pour chaque élément de T, chaque arête du graphe du flot de contrôle P est traversé, et ***toutes les valeurs des éléments des conditions combinées sont traversés au moins une fois (all possible values!)***.
- ❑ **Conditions combinées :**
C1 **et** C2 **ou** C3 ... où les Ci sont des expressions relationnelles ou des variables booléennes (conditions atomiques).
- ❑ **Couverture modifiée des conditions :**
seulement les combinaisons de valeurs de telle façon que Ci conduit aux deux valeurs de la condition générale (vrai et faux).

Non couverture des arêtes masquées

```
if c1 and c2 then
  st;
else
  sf;
end if;
```

```
if c1 then
  if c2 then
    st;
  else
    sf;
  end if;
else
  sf;
end if;
```

- ❑ Deux programmes équivalents
 - quoique vous écrieriez celui de gauche.
- ❑ Couverture des arêtes
 - ne couvrirait pas obligatoirement les arêtes “masquées”,
 - ex.: C2 = false peut ne pas être couvert.
- ❑ La couverture des conditions peut le faire.

Couverture des conditions

❑ **Couverture des conditions :**

sélectionner un jeu de tests T de telle façon qu'en exécutant P pour chaque élément de T, chaque condition atomique ait été évaluée à Vrai et Faux.

❑ **Couverture des conditions / décisions :**

sélectionner un jeu de tests T de telle façon qu'en exécutant P pour chaque élément de T, chaque arête du graphe du flot de contrôle P ait été traversée, et chaque condition atomique ait été évaluée à Vrai et Faux.

❑ **Couverture modifiée des conditions/décisions (MC/DC)**

Renforce la couverture des conditions décisions. Requiert uniquement les combinaisons de valeurs telles que la condition à tester affecte de manière indépendante la décision finale.

* MC/DC Cause Unique = RACC, MC/DC Masquant = CACC

Exemple

- ❑ Le standard international DO-178B pour la certification des systèmes aéroportés (1992).
- ❑ Exemple : $A \wedge (B \vee C)$

	ABC	Res.	Corr. cas faux
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

MC/DC Cause Unique

Prendre une paire pour chaque élément :

- A : (1,5), or (2,6), or (3,7)
- B : (2,4)
- C : (3,4)

Deux ensembles minimum pour couvrir le critère de condition modifié :

- $\{2,3,4\} \cup x \in \{6, 7\}$

Cela fait 4 cas de tests (au lieu de 8 pour chaque combinaison possible).

Couverture des chemins

- ❑ **Critère de couverture des chemins :**
sélectionner un test T tel qu'en exécutant P pour chaque élément de T, tous les chemins conduisant du nœud initial au nœud final du graphe de flot de contrôle de P soient traversés.
- ❑ En pratique, le nombre de chemins est trop grand, sinon infini.
- ❑ Certains chemins sont infaisables.
- ❑ C'est l'une des clés pour déterminer les "chemins critiques".

Exemple: Toutes les arêtes vs. tous les chemins...

```
if x  $\neq$  0 then  
    y := 5;  
else  
    z := z - x;  
end if;  
if z > 1 then  
    z := z / x;  
else  
    z := 0;  
end if;
```

$T1 = \{ \langle x=0, z=1 \rangle, \langle x=1, z=3 \rangle \}$
exécute toutes les arêtes mais ne
montre pas un risque de division par 0.

$T2 = \{ \langle x=0, z=3 \rangle, \langle x=1, z=1 \rangle \}$
trouverait le problème en exerçant les
flot de contrôle restants possibles à
travers le fragment du programme.

$T1 \cup T2 \rightarrow$ tous les chemins couverts.

Cas des boucles

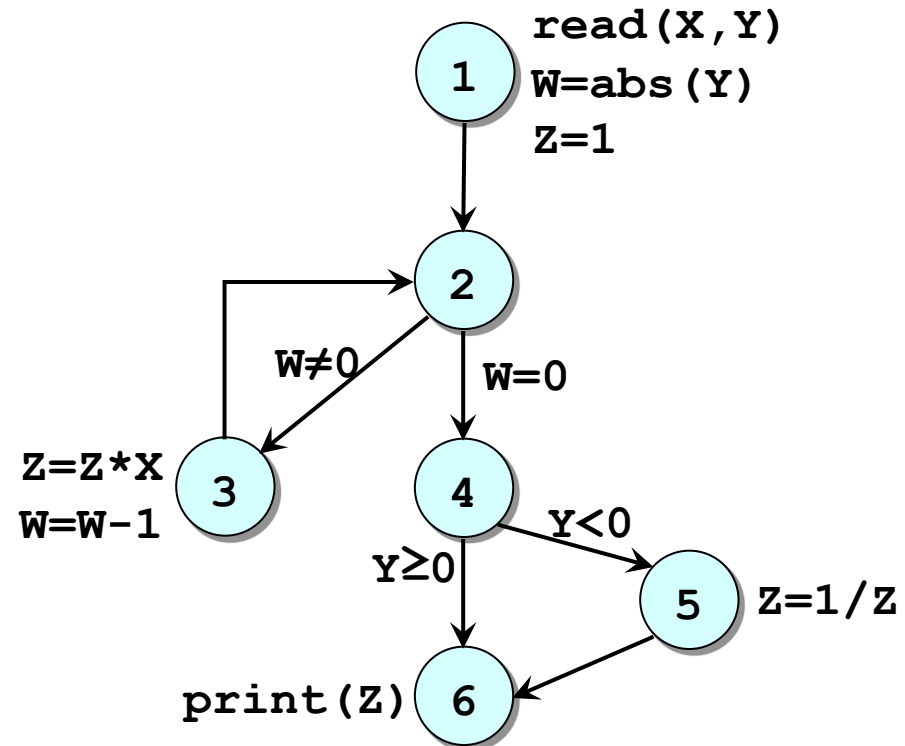
- ❑ Chercher les conditions qui exécutent les boucles :
 - zéro fois,
 - un nombre maximum de fois,
 - un nombre moyen de fois (critère statistique).

- ❑ Par exemple, dans un algorithme de recherche d'un élément dans une table, on peut :
 - sauter une boucle (la table est vide),
 - exécuter la boucle une ou deux fois et par la suite trouver l'élément,
 - chercher dans toute la table sans trouver l'élément désiré.

Autre exemple : la fonction puissance

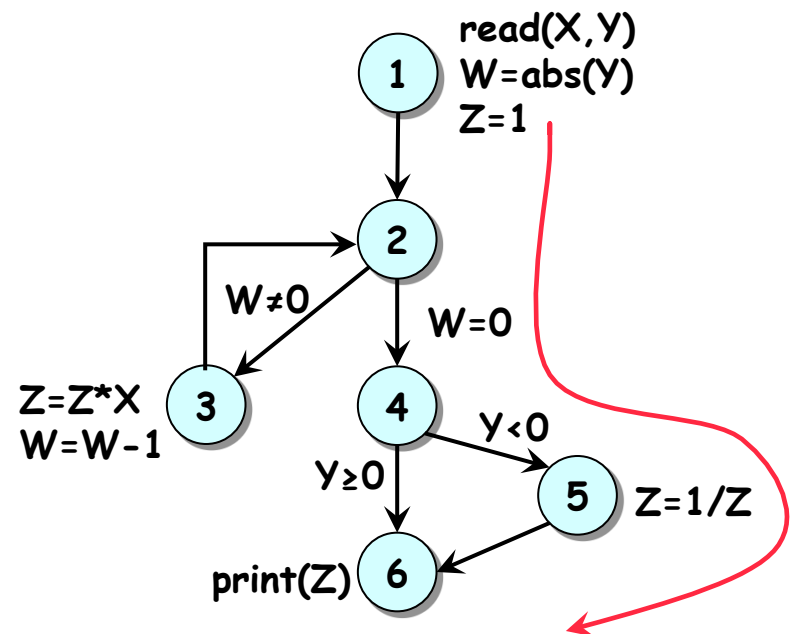
Programme estimé $Z=X^Y$

```
BEGIN
  read (X, Y) ;
  W = abs(Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END
```



Exemple : Test du flot de contrôle

- Tous les chemins.
 - Chemin infaisable :
 - ✓ $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$
 - Nombre infini de chemins :
 - ✓ Autant de chemins à réitérer :
 $2 \rightarrow (3 \rightarrow 2)^*$ que la valeur de $\text{Abs}(Y)$ $[W]$
- Toutes les branches.
 - Deux cas de tests suffisent :
 - ✓ $Y < 0$: $1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$
 - ✓ $Y \geq 0$: $1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^* \rightarrow 4 \rightarrow 6$
- Toutes les instructions.
 - Un cas de test suffit :
 - ✓ $Y < 0$: $1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$



Dérivée des valeurs d'entrée

- ❑ Il arrive, dans les programmes du monde réel, que certaines instructions ne soient pas atteignables
- ❑ Ce n'est pas toujours possible de décider automatiquement si une instruction est atteignable, ni même le pourcentage des instructions atteignables.
- ❑ Quand on n'atteint pas une couverture de 100%, il est parfois très difficile d'en déterminer la raison.
- ❑ Des outils sont nécessaires pour supporter cette activité et la recherche est très active quant à la proposition d'algorithmes efficaces pour dériver automatiquement des jeux de tests présentant des bons niveaux de couverture.
- ❑ Le test de flot de contrôle est, en général, plus adapté pour tester les petits cas.

Plan

- ❑ Couverture du flot de contrôle :
 - instruction, arête, condition, couverture du chemin.
- ❑ Couverture du flot de données :
 - définitions-usages des données.
- ❑ Analyse de la couverture des données.
- ❑ Test de mutation.
- ❑ Test d'intégration :
 - Stratégies,
 - critères.
- ❑ Conclusions :
 - génération des données de test, outils, recommandations de Marick.

Analyse du flot de données

- ❑ S'intéresse aux chemins du CFG qui sont significatifs pour le flot de données dans le programme.
- ❑ Se concentre sur l'affectation des valeurs aux variables et à leurs usages.
- ❑ Analyse des occurrences des variables.
 - L'occurrence de définition : la valeur est liée à la variable.
 - Les occurrences d'utilisation : la valeur de la variable est référée.
 - ✓ L'utilisation comme prédicat : la variable est utilisée pour décider si le prédicat est vrai.
 - ✓ L'utilisation de calcul : la variable est utilisée pour définir d'autres variables ou calculer une valeur (possiblement de sortie).

Exemple factoriel

```
1. public int factorial(int n) {  
2.     int i, result = 1;  
3.     for (i=2; i<=n; i++) {  
4.         result = result * i;  
5.     }  
6.     return result;  
7. }
```

Variable	Ligne de définition	Ligne d'utilisation
n	1	3
résultat	2	4
résultat	2	6
résultat	4	4
résultat	4	6

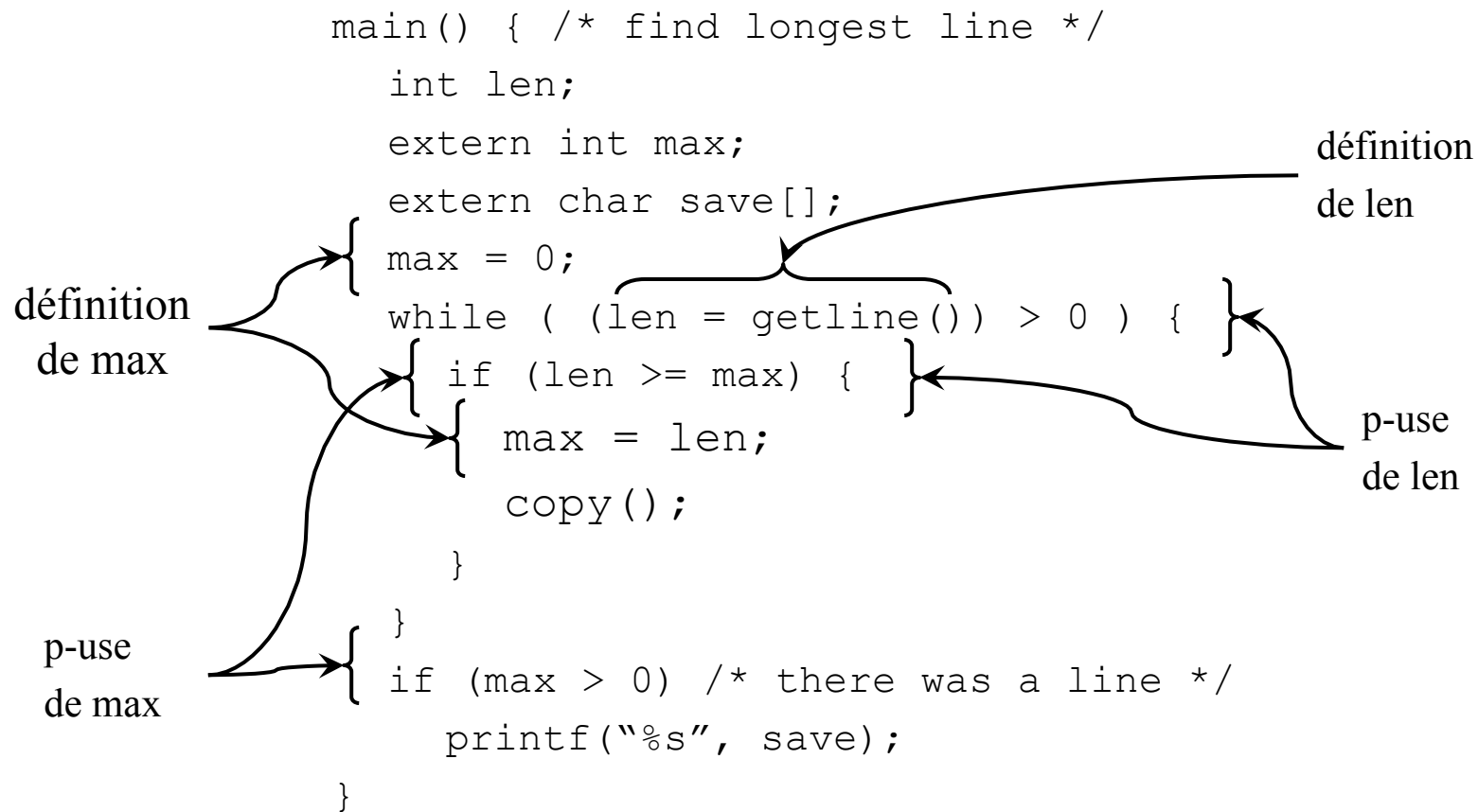
Définitions élémentaires

- ❑ Le nœud $n \in \text{CFG}(P)$ est un **nœud de définition** de la variable $v \in V$, écrit comme $\text{DEF}(v,n)$, ssi la valeur de la variable v est définie dans l'instruction correspondant au nœud n .
- ❑ Le nœud $n \in \text{CFG}(P)$ est un **nœud d'utilisation** de la variable $v \in V$, écrit comme $\text{USE}(v,n)$, ssi la valeur de la variable v est utilisée dans l'instruction correspondant au nœud n .
- ❑ Un nœud utilisé $\text{USE}(v,n)$ est une **utilisation comme prédicat** (représenté comme P-Use) ssi l'instruction n est une instruction prédicat, sinon $\text{USE}(v,n)$ est une **utilisation de calcul** (représentée comme C-use).

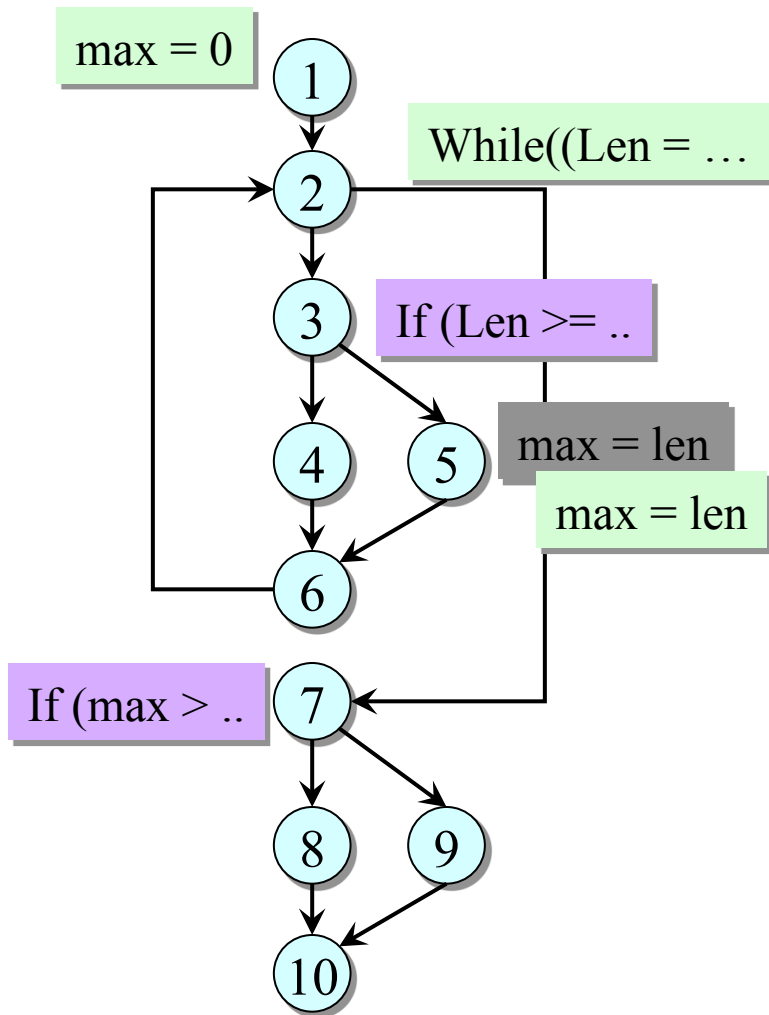
Définitions élémentaires II

- Un **(sous)chemin definition-use** d'une variable v (notée du-path) est un (sous)chemin dans $\text{PATHS}(P)$ tel que, pour $v \in V$, il y a un nœud de définition $\text{DEF}(v, m)$ et un nœud d'utilisation $\text{USE}(v, n)$ avec m et n sont les nœuds initial et final du (sous) chemin.
- Un **(sous)chemin definition-clear** d'une variable v (notée dc-path) est un chemin definition-use de $\text{PATHS}(P)$ avec un nœud initial $\text{DEF}(v, m)$ et un nœud final $\text{USE}(v, n)$ de telle manière qu'aucun autre nœud du chemin ne soit un nœud de définition de v .

Exemple simple



Exemple simple (CFG)



<code>v = ...</code>	Définition DEF(v, n)
<code>v >= ...</code>	P-use USE(v,n)
<code>... = ...v ...</code>	C-use USE(v,n)

- DEF(max, 1)
- DEF(len, 2)
- DEF(max, 5)
- C-USE(len, 5)
- P-USE(len, 2)
- P-USE(len, 3)
- P-USE(max, 3)
- P-USE(max, 7)

Définitions formelles de critères I

- ❑ L'ensemble T satisfait le critère **all-Definitions** pour le programme P ssi, pour chaque variable $v \in V$, T contient au moins un chemin definition-clear à partir de chaque nœud de définition de v à un nœud d'utilisation de v .
- ❑ L'ensemble T satisfait le critère **all-Uses** pour le programme P ssi, pour chaque variable $v \in V$, T contient au moins un chemin definition-clear partant de chaque nœud de définition de v à chaque nœud d'utilisation atteignable de v .
- ❑ L'ensemble T satisfait le critère **all-P-Uses/Some C-Uses** pour le programme P, ssi pour chaque variable $v \in V$, T contient au moins un chemin definition-clear partant de chaque nœud de définition de v à chaque utilisation prédicat de v , et si une définition de v n'a pas de P-Uses, il y a un chemin definition-clear à au moins une utilisation calcul.

Définitions formelles de critères II

- ❑ L'ensemble T satisfait le critère **all-C-Uses/Some P-Uses** pour le programme P ssi, pour chaque variable $v \in V$, T contient au moins un chemin definition-clear partant de chaque nœud définition de v à chaque nœud d'utilisation de calcul de v , et si une définition de v n'a pas de C-Uses, il y a un chemin definition-clear à au moins une utilisation prédicat de v .
- ❑ L'ensemble T satisfait le critère **all-DU-Paths** pour un programme P ssi, pour chaque variable $v \in V$, T contient tous les chemins definition-clear partant de chaque nœud de v à chaque nœud d'utilisation de v atteignable, et que ces chemins sont soit des traverses simples de boucles, soit ils ne contiennent pas de cycles.

Petit programme

Programme estimé $Z=X^Y$

```
BEGIN
  read (X, Y) ;
  W = abs(Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END
```

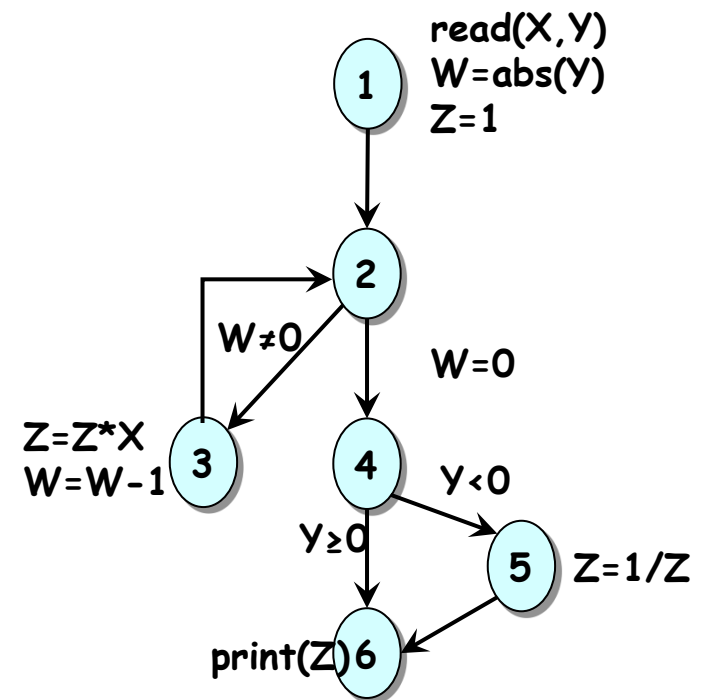
Exemple de puissance (w,1-2,1-3)

Dcu = Definition c-use (clear)

Dpu = Definition p-use (clear)

nœud i	def(i)	c-use(i)	p-use(i,j)
1	X, Y, W, Z	Y	
2			W
3	W, Z	X, W, Z	
4			Y
5	Z	Z	
6		Z	

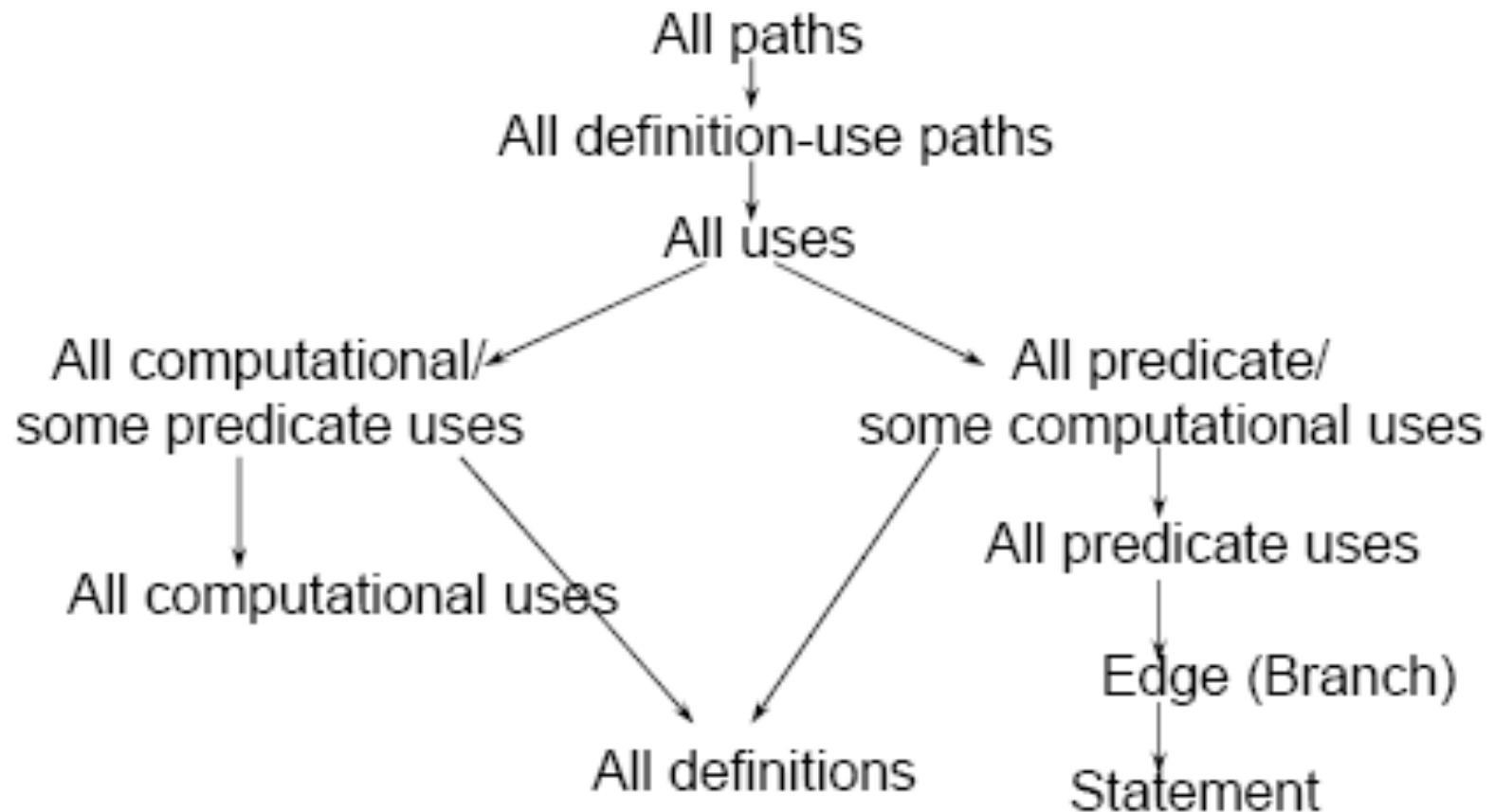
nœud i	dcu(i,v)	dpu(i,v)	du(i,v)
1	dcu(1,X) = {3} dcu(1,Z) = {3,6} dcu(1,W) = {3}	dpu(1,Y) = {4} dpu(1,W) = {2}	du(1,W)={2,3} du(1,X)={3} du(1,Y)={4} du(1,Z)={3,6}
3	dcu(3,W) = {3} dcu(3,Z) = {3,6}	dpu(3,W) = {2}	du(3,W)={2,3} du(3,Z)={3,6}
5	dcu(5,Z) = {6}		du(5,Z) = {6}



Discussion

- ❑ Aide à générer des données de test en suivant la manière dont la donnée est manipulée dans le programme.
- ❑ Aide à définir les critères intermédiaires entre le test de toutes les arêtes (possiblement trop faible) et le test de tous les chemins (souvent impossible).
- ❑ Mais nécessite le support d'un outil efficace (voir la fin du chapitre).

Hiérarchie des critères de couverture



Mesure de la couverture du code

- ❑ Un avantage des critères structurels est que leur couverture peut être mesurée *automatiquement* :
 - Pour contrôler le progrès des tests,
 - Pour estimer l'achèvement des tests en terme d'anomalies restantes et de fiabilité,
 - Pour aider à fixer des objectifs pour les testeurs.
- ❑ Une haute couverture n'est pas une garantie d'un logiciel sans anomalie, seulement un élément d'information pour augmenter notre confiance → modèles statistiques.

Plan

- ❑ Couverture du flot de contrôle :
 - Instruction, arête, condition, couverture du chemin.
- ❑ Couverture du flot de données :
 - définitions-usages des données.
- ❑ Analyse de la couverture des données.
- ❑ Test de mutation.
- ❑ Test d'intégration :
 - Stratégies,
 - critères.
- ❑ Conclusions :
 - production des données de test, outils, recommandations de Marick.

Productivité de test

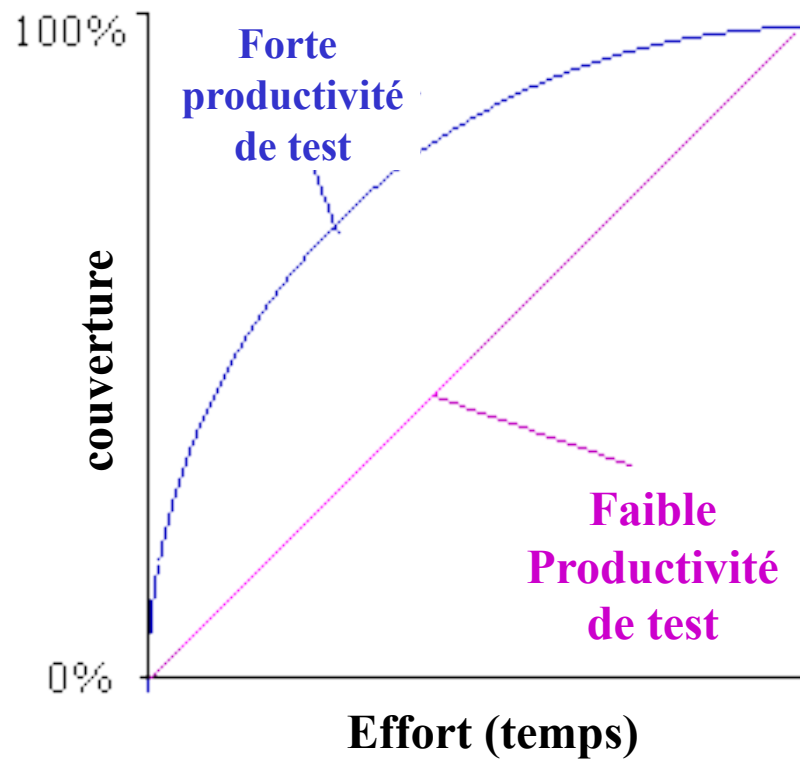


Figure 1 : Taux de la couverture

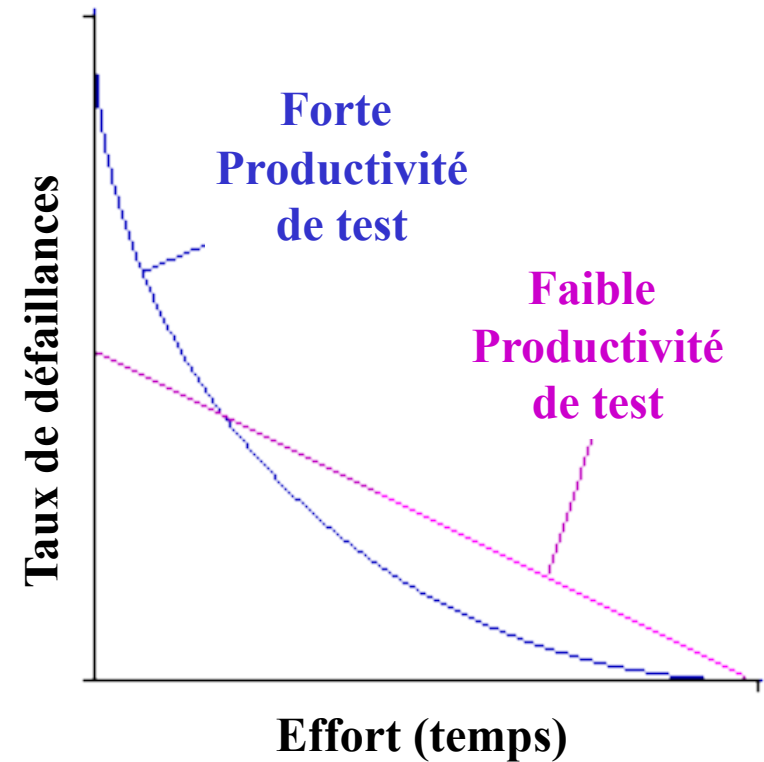
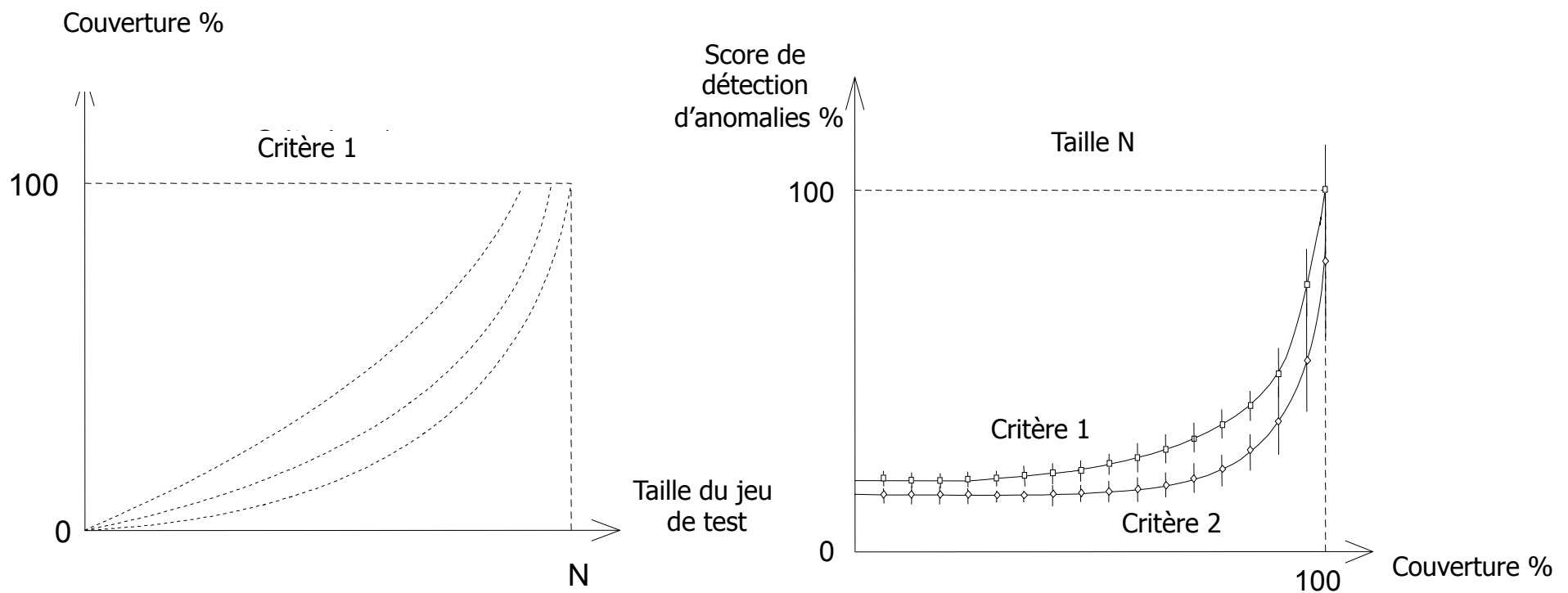


Figure 2 : Taux de détection de défaillances

Analyses typiques



Expérimentation de Hutchins et al.

- ❑ Les critères de couverture All-Edges (tous les arêtes) et All-DU ont été appliqués à 130 versions de programmes défectueux provenant de 7 programmes C (141-512 LOC) en propageant des anomalies réalistes.
- ❑ Les 130 anomalies ont été créées par 10 personnes différentes, chacune ne connaissant pas le travail de l'autre; leur but étant d'être aussi réaliste que possible.
- ❑ La procédure de génération des tests était conçue pour produire une large gamme de tests i.e., au moins 30 jeux de tests pour chaque 2% d'intervalle de couverture pour chaque programme.
- ❑ Ils examinèrent la relation entre la détection des anomalies et la couverture du jeu de test / taille.

Un exemple de programme

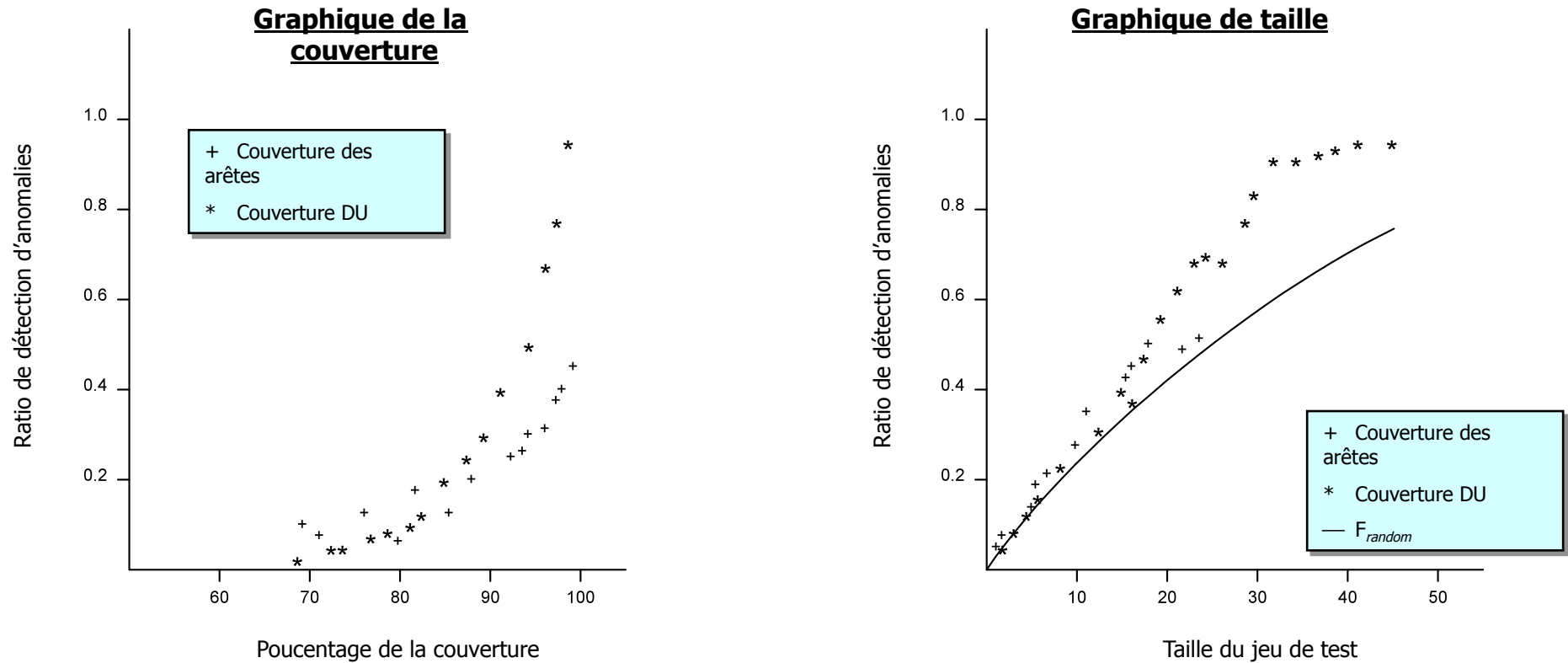


Figure : Ratios de détection d'anomalies pour un programme défectueux.

Résultats

- ❑ Les deux critères de la couverture ont performé mieux que la sélection aléatoire des tests– spécialement DU-coverage.
- ❑ Des améliorations significatives se sont produites au moment où la couverture a augmenté de 90 à 100%.
- ❑ Une couverture de 100% seule n'est pas un indicateur fiable de l'efficacité d'un jeu de tests – spécialement la couverture des arêtes.
- ❑ Une large variation de l'efficacité de test pour une couverture donnée.
- ❑ Confirme qu'en moyenne, réaliser une couverture all-DU nécessite des jeux de tests plus grands que ceux requis par la couverture all-Edges (tous les arêtes).

Plan

- ❑ Couverture du flot de contrôle :
 - instruction, arête, condition, couverture du chemin.
- ❑ Couverture du flot de données :
 - définitions-usages des données.
- ❑ Analyse de la couverture des données.
- ❑ **Test de mutation.**
- ❑ Test d'intégration :
 - Stratégies,
 - critères.
- ❑ Conclusions :
 - génération des données de tests, outils, recommandations de Marick.

Test de mutation : définitions – DE Millo

- ❑ *Test Fault-based* : orienté vers la découverte des anomalies “typiques” qui peuvent se produire dans un programme.
- ❑ Les variations syntaxiques sont appliquées d’une manière systématique au programme pour créer des versions défectueuses qui montrent un comportement différent (mutant).
- ❑ E.g., $x = a + b$ vs $x = a - b$
- ❑ **Le test de mutation aide un utilisateur à créer des données de test effectives d’une manière interactive.**
- ❑ L’objectif est de pousser chaque version défectueuse (mutant) à montrer un comportement différent.

Mutants différents

- ❑ Les mutants Stillborn : syntaxiquement incorrects, supprimés par le compilateur.
- ❑ Les mutants Triviaux: supprimés par presque n'importe quel cas de test.
- ❑ Les mutants équivalents : produisent toujours la même sortie que le programme original.
- ❑ Aucun des mutants décrits ci-dessus ne sont intéressants d'un point de vue de test de mutation.

Idée de base

- ❑ Prendre un programme et les données de test générées pour ce programme (utilisant une autre technique de test).
- ❑ Créer un nombre de programmes *similaires* (mutants), chacun différant légèrement de l'original, i.e., chacun possédant une anomalie.
- ❑ Exemple: remplacer l'opérateur d'addition par l'opérateur de multiplication.
- ❑ Les données originales du test sont alors utilisées par les *mutants*.
- ❑ Si les données de test détectent des différences dans les mutants, alors on dit que les mutants sont *morts* et le jeu de tests est *adéquat*.

Idée de base II

- ❑ Un mutant reste *Vivant* soit parce qu'il est équivalent au programme original (fonctionnellement identique bien que différent syntaxiquement – mutant équivalent) ou parce que le test est inadéquat pour supprimer le mutant.
- ❑ Dans le dernier cas, les données de test ont besoin d'être réexaminées, possiblement augmentées pour supprimer le mutant *vivant*.
- ❑ Pour la génération automatique des mutants, nous utilisons les *opérateurs de mutation*, qui prédéfinissent les règles de modification de programme (i.e., correspondant à un modèle d'anomalie).

Exemple d'opérateurs de mutation

- ❑ Remplacement de constante.
- ❑ Remplacement d'une variable scalaire.
- ❑ Remplacement d'une Variable scalaire par une constante.
- ❑ Remplacement d'une constante par une Variable scalaire.
- ❑ Remplacement d'une référence dans un tableau par une constante.
- ❑ Remplacement d'une référence dans un tableau par une variable scalaire.
- ❑ Remplacement d'une constante par une référence dans un tableau.
- ❑ Remplacement d'une variable scalaire par une référence dans un tableau.
- ❑ Remplacement d'une référence dans un tableau par une référence dans le tableau.
- ❑ Remplacement de la source de constante.
- ❑ Altération d'une instruction de données.
- ❑ Remplacement du nom d'un tableau par un nom comparable.
- ❑ Remplacement d'un opérateur arithmétique.
- ❑ Remplacement d'un opérateur relationnel.
- ❑ Remplacement d'un opérateur logique de base.
- ❑ Insertion de la valeur absolue.
- ❑ Insertion d'un opérateur unaire.
- ❑ Effacement d'une instruction.
- ❑ Remplacement de l'instruction de retour (return).

Exemple d'opérateurs de mutation II

Spécifique aux langages de programmation orientée objets :

- ❑ Remplacer un type avec un sous-type compatible (héritage).
- ❑ Changer le modificateur d'accès d'un attribut, d'une méthode.
- ❑ Changer l'expression de création d'instance (héritage).
- ❑ Changer l'ordre des paramètres dans la définition de la méthode.
- ❑ Changer l'ordre des paramètres dans un appel.
- ❑ Enlever une méthode de surcharge.
- ❑ Réduire le nombre de paramètres.
- ❑ Enlever la redéfinition de méthode.
- ❑ Enlever un champ caché.
- ❑ Ajouter un champ caché.

Spécifier des opérateurs de mutation

- ❑ Idéalement, nous aimerions que les opérateurs de mutation représentent (et engendrent) tous les types réels d'anomalies qui pourraient se produire en pratique.
- ❑ Les opérateurs de mutation changent avec les langages de programmation, la conception et les paradigmes de spécification, quoiqu'il y ait beaucoup de chevauchements.
- ❑ En général, le nombre d'opérateurs de mutation est étendu puisqu'ils sont supposés capter toutes les variations syntaxiques possibles dans un programme.
- ❑ Quelques études récentes semblent suggérer que les mutants sont de bons indicateurs d'efficacité de test (Andrews et al 2004).

Couverture de mutation

- ❑ La couverture complète équivaut à la suppression des mutants non-équivalents.
- ❑ Le taux de la couverture est appelé “le score de mutation”.
- ❑ Nous pouvons voir chaque mutant comme un test requis.
- ❑ Le nombre de mutants dépend de la définition des opérateurs de mutation et de la syntaxe/structure du logiciel.
- ❑ Le nombre de mutants tend à être élevé, même pour de petits programmes (échantillonnage aléatoire?).

Exemple simple

Fonction originale

```
int Min (int A, int B)
{
    int minVal;
    minVal = A;
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

Avec mutants incorporés

	<pre>int Min (int A, int B) { int minVal; minVal = A; minVal = B; if (B < A) if (B > A) if (B < minVal) { minVal = B; Bomb(); minVal = A; minVal = failOnZero (B); } return (minVal); } // end Min</pre>
Δ1	
Δ2	
Δ3	
Δ4	
Δ5	
Δ6	

Discussion de l'exemple - Offutt

- ❑ Le mutant 3 est équivalent, à ce point, `minVal` et `A` ont la même valeur.
- ❑ Le mutant 1 : afin de trouver un cas de test approprié, nous devons
 - atteindre l'anomalie injectée durant l'exécution (atteignabilité) :
 - ✓ toujours vrai ;
 - causer l'état du programme à être incorrect (infection) :
 - ✓ $A \neq B$;
 - causer la sortie du programme à être incorrecte (propagation) :
 - ✓ $(B < A) = \text{faux}$.

Hypothèses

- ❑ Que pensez-vous d'erreurs plus complexes, impliquant plusieurs instructions ?
- ❑ *Supposition de programmeurs compétents*: ils écrivent des programmes qui sont presque corrects.
- ❑ *Supposition de l'effet de couplage* : la donnée de test qui permet de distinguer les programmes qui diffèrent d'un programme correct par de simples erreurs est tellement sensible qu'elle distingue aussi implicitement des erreurs plus complexes.
- ❑ Il y a quelques évidences empiriques de ces hypothèses.

Exemple simple

Spécification :

Le programme demande à l'utilisateur un nombre entier positif compris entre 1 et 20 puis, une chaîne de cette longueur.

Le programme demande par la suite un caractère et retourne la position dans la chaîne où le caractère a été trouvé la première fois ou un message indiquant que le caractère n'était pas présent dans la chaîne.

L'utilisateur a la possibilité de chercher d'autres caractères.

Morceau de code

```
...  
i := 1;  
...  
found := FALSE;  
i := 1;  
while (not (found)) and (i <= x) do  
begin  
    if a[i] = c then  
        found := TRUE  
    else  
        i := i + 1  
end  
...
```

Cas de test 1

Entrée				Sortie prévue
x	a	c	Réponse (Continuer?)	
25				Entrez un nombre entier entre 1 et 20
1	x	x		Caractère x apparaît à la position 1
			y	
		a		Caractère ne se trouve pas dans la chaîne
			n	

Mutant 1

- ❑ Remplacer `Found := FALSE;` avec `Found := TRUE;`
- ❑ Reprendre l'ensemble original des données de test.
- ❑ Un seul petit changement à la fois, pour éviter le danger des anomalies introduites avec des effets interférents.
- ❑ Échec : le "caractère « a » apparaît à la position 1" au lieu de dire que ce caractère ne se trouve pas dans la chaîne.
- ❑ Le mutant 1 a été supprimé.

Mutant 2

- ❑ Remplacer $i := 1$ (la 2^{ème} occurrence) par $x := 1$;
- ❑ Notre ensemble original de données de test a échoué dans la détection de l'anomalie.
- ❑ Comme résultat du bogue, seule la position 1 dans la chaîne sera examinée.
- ❑ Nous devons augmenter notre longueur de chaîne et rechercher un caractère plus loin le long de celle-ci.
- ❑ Nous modifions les données de test afin de
 - préserver l'effet des premiers tests,
 - pour être certain que le mutant vivant sera supprimé.

Cas de test 2

Entrée				Sortie prévue
x	a	c	Réponse (Continuer?)	
25				Entrez un nombre entier entre 1 et 20
3	xCv	x		Caractère x apparaît à la position 1
			y	
		a		Caractère ne se trouve pas dans la chaîne
			y	
		v		Caractère v apparaît à la position 3
			n	

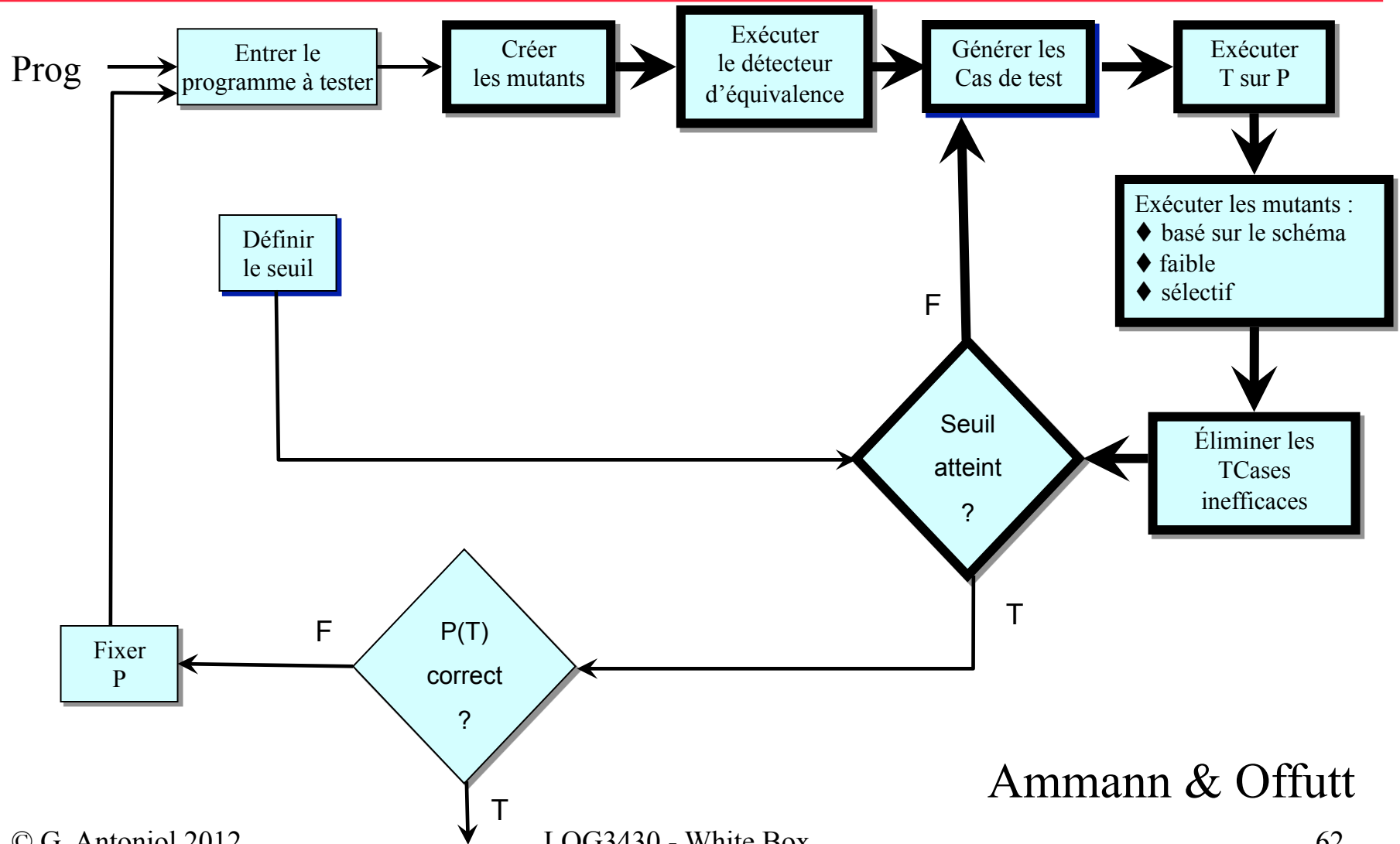
Mutant 3

- ❑ $i := i + 1;$ est remplacé par $i := i + 2;$
- ❑ Encore une fois, nos données de test ont échoué à supprimer le mutant.
- ❑ Nous devons augmenter les données de test afin de rechercher un caractère dans le milieu de la chaîne.
- ❑ Avec les nouvelles données de test, le mutant 3 est mort.
- ❑ Plusieurs autres changements peuvent être faits avec ce petit morceau de code, e.g., changer la matrice de la référence, changer l'opérateur relationnel.

Cas de test 3

Entrée				Sortie Prévue
x	a	c	Réponse	
25				Entrez le nombre entier entre 1 et 20
1	xCv	x		Caractère x apparaît à la position 1
			y	
		a		Caractère ne se trouve pas dans la chaîne
			y	
		v		Caractère v apparaît à la position 3
			y	
		C		Caractère C apparaît à la position 2
			n	

Processus de test de mutation



Ammann & Offutt

Discussion (test de mutation)

- ❑ Il mesure la qualité des cas de test.
- ❑ Il fournit au testeur un objectif clair (des mutants à supprimer).
- ❑ Le test de mutation montre que certaines sortes d'anomalies (spécifiées par le modèle d'anomalies) sont improbables.
- ❑ Il force le programmeur à scruter le code et à penser aux données de test qui exposeront certaines sortes d'anomalies.
- ❑ Un très grand nombre possible de mutants est généré : échantillonnage aléatoire, opérateurs sélectif de mutation (Offutt).
- ❑ Les mutants équivalents sont un problème pratique : c'est, en général, un problème indécidable.
- ❑ Probablement plus utile pour les tests unitaires.
- ❑ Quelques systèmes (outils) ont été conçus pour aider mais ils consomment encore beaucoup de temps.

Autres applications

- ❑ Les opérateurs et les systèmes de mutation sont aussi très utiles pour l'évaluation de l'efficacité des stratégies de tests—ils ont été utilisés dans un nombre d'études de cas :
 - Définir un ensemble d'opérateurs de mutation réalistes ;
 - générer des mutants (automatiquement) ;
 - générer des cas de test selon des stratégies alternatives ;
 - estimer le *score de mutation* (pourcentage de mutants supprimés).

Autres applications II

- ❑ Nous avons vu des opérateurs de mutation en code.
- ❑ Mais il y a du travail fait sur
 - les opérateurs de mutation pour les interfaces de module (visés dans le test d'intégration) ;
 - les opérateurs de mutation sur des spécifications : les réseaux de Petri, les machines à états, ... (visés dans le test du système).