Fiabilité Logicielle

Tests

Arnaud Labourel

2025-26

amU Faculté des sciences Aix Marseille Université

Section 1

Généralités sur le test logiciel

Définitions du test

Norme IEEE (Standard Glossary of Software Engineering Terminology)

« 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. »

- Validation dynamique (exécution du système)
- Comparaison entre système et spécification

Périmètre du test

E. W. Dijkstra (prix Turing 1972)

"Program testing can be used to show the presence of bugs, but never to show their absence !"

$\Rightarrow$ le test ne permet pas de garantir la conformité d'un système (une infinité de tests serait nécessaire)

G. Myers (The Art of Software testing)

"Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts."

$\Rightarrow$ le test a pour but la détection de bugs.

Bug ?

- erreur : instruction erronée (l'erreur est humaine)
- infection : propagation des conséquences de l'erreur dans la suite du programme
- défaut : constatation d'une faute/ d'un défaut dans le programme

Le bon test met en lumière un défaut qui permettra de retrouver l'erreur.

Le vocabulaire du test (1/2)

- **Objectif de test** : comportement du système à tester
- **Données de test (DT)** : données à fournir en entrée au système afin de tester le comportement souhaité.
- **Résultat de test** : "sorties" de l'exécution d'un test (affichage, valeur des variables, envoi de messages, ...)

Objectif :	première sol. (+ Delta) d'une équation du second degré
Données :	coefficients a, b, c flottants
Résultat :	valeur retournée par la fonction

- **Cas de test (CT)** : données d'entrée et résultats attendus associés à un objectif de test

Objectif :	première sol. (+ Delta) d'une équation du second degré
Données :	1,-3,-10
Résultat :	5

Le vocabulaire du test (2/2)

- **Suite de test** : ensemble de cas de test permettant d'atteindre un ou plusieurs objectifs de test.
- **Scénario de test** : pour un cas de test ou/et une suite de test, suite des opérations nécessaires à sa réalisation
- **Plan de test** : document décrivant le périmètre, l'approche, les ressources (humaines, logicielles et matérielles) ainsi que le planning des activités de test prévues.

Le processus de test

1. Choisir les comportements à tester (objectifs de test)
2. Choisir des données de test permettant d'initier ces comportements + décrire le résultat attendu pour ces données
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus pour établir un verdict (via un oracle)

Oracle

Oracle : Donne le verdict quant-à la réussite du test en comparant le résultat produit avec la valeur attendue

Problème : rendre cette décision peut être complexe notamment si

- égalité entre objets calculés est difficilement définissable
- le système n'est pas déterministe ou fait intervenir l'aléatoire
- le programme est basé sur des heuristiques et ne rend qu'une approximation de l'optimal

Alternative :

- la valeur exacte attendue
- une liste de valeurs possibles
- une propriété que doit vérifier la valeur calculée

Oracle : exemple

- Spécification : Trouver le minimum d'une liste d'entiers

  Entrée : [4 ; 2 ; 3 ; 6] Sortie attendue : 2

  Oracle : Égalité sur les entiers **OK**

- Spécification : Calculer l'itinéraire le plus rapide entre deux villes

  Entrée : Paris – Lyon Sortie attendue : ... A6...

  Oracle : Égalité des chemins ? **KO**

  Trajet de 4h17 (quel que soit l'itinéraire choisi) **OK**

- Spécification : Problème du sac à dos (résolu avec une heuristique)

  Oracle : Résultat raisonnablement éloigné du résultat optimal ? **KO**
  Résultat = résultat optimal + 5% **OK**

Oracles complexes

Oracle : en général, résultat attendu = ensemble de conditions si plusieurs résultats possibles et énumération impossible

Concevoir l'oracle est dans ce cas une tâche complexe avec les risques d'avoir des conditions

- trop restrictives et de voir des programmes corrects échouer au test (Faux-négatifs)
- trop larges et de voir des programmes incorrects passer le test (Faux-positifs)

La validité des tests est importante : les tests se doivent de réussir sur les programmes conformes à la spécification.

Scénario de test

Le test est découpé en plusieurs parties :

- **le préambule** :
  - dans le cas de test structurel, création et initialisation des objets, fichiers, sockets, connexion à une base de données ... nécessaires à l'exécution du test.
  - dans le cas de tests fonctionnels, représente la séquence d'appels d'opérations depuis l'état initial jusqu'à l'état qui permet d'activer le comportement associé à l'objectif du test
- **le corps du test** : exécution du cas de test (exécution du programme sur les données de test et verdict de l'oracle)
- **le postambule** :
  - dans le cas de test structurel, désallocations et/ou suppressions des objets, fichiers, sockets, connexion à une base de données ... nécessaires à l'exécution du test.
  - dans le cas de tests fonctionnels, une séquence d'opérations qui permet de ramener le système à l'état initial (ou à celui précédent le test)

Section 2

Tests boite blanche

Rappel : le processus de test

1. Choisir les comportements à tester (objectifs de test)
2. Choisir des données de test permettant d'initier ces comportements + décrire le résultat attendu pour ces données
3. Exécuter les cas de test sur le système + collecter les résultats
4. Comparer les résultats obtenus aux résultats attendus pour établir un verdict

Test boîte blanche - test structurel

Test boite blanche :

- tester avec l'implémentation visible
- tester l'implémentation (vis-à-vis d'une spécification détaillée)

Test structurel (basé sur la structure du programme)

⊕ au plus près du programme et non de la spécification

⊕ proche du comportement réel du produit final

⊖ au plus près du programme et non de la spécification

⊖ basé sur une nécessaire abstraction du comportement

Abstraction pour le test boîte blanche

- abstraction du programme par un modèle (possiblement guidé par les tests à mener)
- utilisation de cette abstraction pour définir les cas de test et les suites de tests

Deux modèles usuels :

- le graphe de flot de contrôle (CFG)
- le graphe de flot de données (DFG)

Ces modèles sont également :

- construits lors de la compilation par la plupart des compilateurs (gcc, javac, ..) afin d'optimiser le code produit
- les représentations de base des outils d'analyse statique du code

Graphe de flot de contrôle

Deux types de graphes :

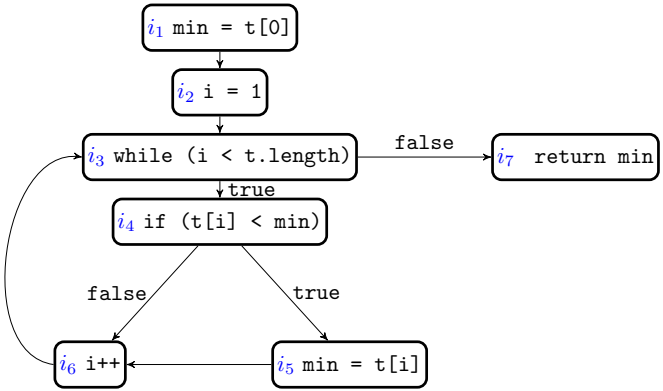
- intra-procédural :
  - restreint à ce qui se passe au sein d'une procédure/fonction/méthode
  - la granularité est donnée par les instructions
- inter-procédural :
  - considère le programme dans sa globalité
  - la granularité est donnée par les appels de procédure/fonction/méthode

CFG intra-procédural (1/2)

Graphe orienté :

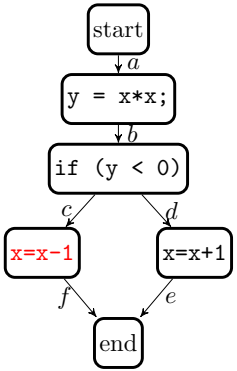
- nœuds = instructions
- arc reliant les instructions pouvant se suivre directement.

```
min = t[0];
i = 1;
while (i < t.length) {
  if (t[i] < min) {
    min = t[i];
  }
  i++;
}
return min;
```



CFG intra-procédural (2/2)

```
y = x*x;
if (y < 0) {
  x = x - 1;
} else {
  x = x + 1;
}
```



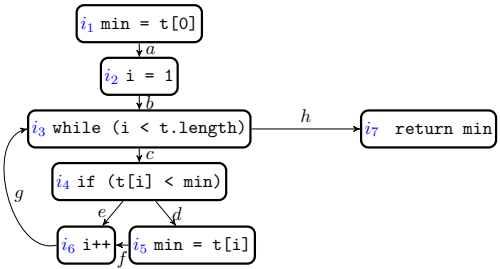
- ajout possible d'un nœud d'entrée (start) et de sortie (end) (éviter les cycles sur ces nœuds)
- ajout de nœud pour rendre le graphe plus lisible
- définition des nœuds par des blocs et non plus des instructions :
  - bloc = suite d'instructions avec un seul successeur et n'ayant qu'un seul antécédent, lui-même membre du bloc
  - si une exécution contient une instruction du bloc alors elle les contient toutes

Chemins dans un CFG (1/2)

Un chemin dans un CFG est :

- soit une suite de nœuds
- soit une suite d'arêtes

allant de l'entrée à la/une sortie



suite de nœuds :

- $i_1 i_2 i_3 i_7$
- $i_1 i_2 i_3 i_4 i_5 i_6 i_3 i_7$
- $i_1 i_2 i_3 i_4 i_6 i_3 i_4 i_6 i_3 i_7$

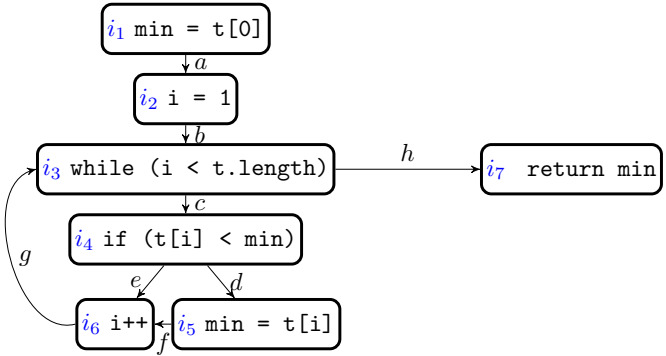
suite d'arêtes :

- abh
- $abcdfgcegcegh$

Si le graphe contient une boucle alors le nombre de chemins est infini.

Chemins dans un CFG (2/2)

On peut décrire l'ensemble des chemins par une expression régulière

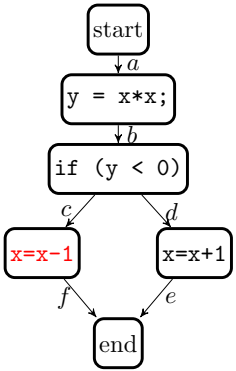


- $i_1 i_2 (i_3 i_4 (\epsilon + i_5) i_6)^* i_3 i_7$
- $ab(c(df + e)g)^* h$

Chemins réalisables dans un CFG

Définition

Un chemin du graphe est *réalisable* s'il existe une exécution du programme suivant exactement ce chemin.



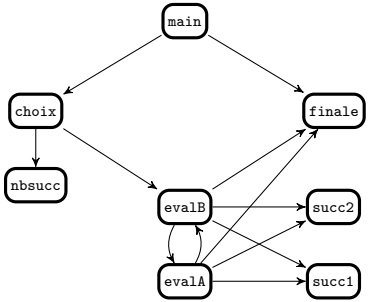
- le chemin $abce$ est réalisable
- le chemin $abcf$ n'est pas réalisable

Il est indécidable de savoir si un chemin est réalisable (puisque l'atteignabilité d'une instruction ne l'est pas).

CFG inter-procédural

- les noeuds sont les procédures/fonctions/méthodes.
- une arête orientée entre la procédure appelante (calling method) et la méthode appelée (callee method).

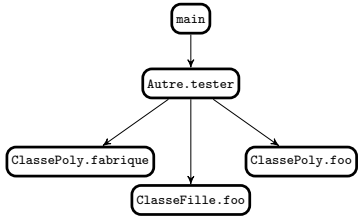
```
int evalA(position p) {
 if (finale(p)) { evalA=1;}
  else {if (nbsuc(p)==1){evalA=evalB(suc1(p));}
           else {evalA=max(evalB(suc1(p)),evalB(suc2(p)));}}
 return evalA;}

int evalB(position p) {
 if (finale(p)) {evalB=1;}
  else {if (nbsuc(p)==1){evalB=evalA(suc1(p));}
          else {evalB=max(evalA(suc1(p)),evalA(suc2(p)));}}
 return evalB;}

int choix(position p) {
  if (nbsuc(p)==1){choix=1;}
    else {if (evalB(suc1(p)) > evalB(suc2(p))) { choix=1;}
             else {choix=2;}}
return choix;}

int main { ... while (!finale(p)){
  if (..) {n=choix(p);...}
    else {...} }
return 0;}
```



CFG inter-procédural

L'héritage et le polymorphisme rendent le graphe moins précis : les arguments effectifs lors de l'appel peuvent être nécessaires pour identifier précisément la méthode appelée (late binding).

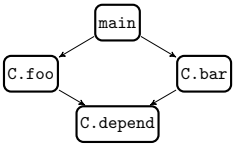
```
public class ClassePoly {
 public static ClassePoly fabrique(String sorte ) {
    if (sorte=="ClassePoly"){ return new ClassePoly();}
    if (sorte=="ClasseFille"){ return new ClasseFille();}
    return null;
   }

 public static void main(String [] args) {
    (new Autre()).tester();}

  void foo() {
     System.out.println("methode foo () appelee: pere");}
}


public class ClasseFille extends ClassePoly{
  void foo() {
     System.out.println("methode foo () appelee: fille");}
}

public class Autre extends ClassePoly{
  void tester (){
     ClassePoly monClassePoly=ClassePoly.fabrique("ClasseFille");
     monClassePoly.foo();
   }
}
```

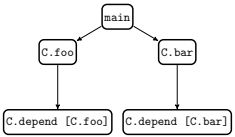


CFG inter-procédural

Context-insensitive vs Context-sensitive

- Context-insensitive : chaque appel est indépendant de son contexte : les noeuds du graphe correspondent aux noms des méthodes (annotés par la classe)
- Context-sensitive : deux appels (occurrences d'appel) à une même méthode (d'une même classe) peuvent être distingués :
  - selon la classe appelante
  - selon les paramètres formels
  - selon la position de l'appel dans une même méthode

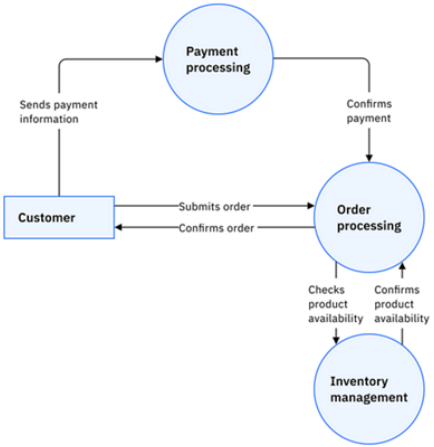




Graphe de flot de données (DFG)

Graphe de flot de données

- modéliser les dépendances entre les données
- dépendance : relier la définition d'une donnée et son/ses utilisation(s).



DFG : définition / utilisation

Définition : endroit dans le programme où une variable est initialisée ou où sa valeur est (re)définie

- partie gauche d'une affectation
- paramètres effectifs des méthodes
- valeurs lues
- valeur par défaut (si existante dans le langage)

Utilisation : endroit dans le programme où la valeur de la variable est utilisée

- partie droite d'une affectation
- conditionnelle
- indice de tableau
- passage de paramètres
- valeur retournée, valeur affichée

DFG : le graphe

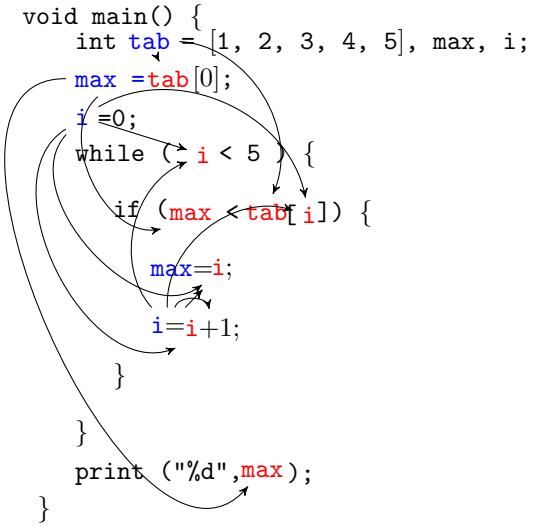
Une arête existe entre deux nœuds du graphe

- entre la définition et l'utilisation d'une même variable (dans un scope de définition)
- une exécution potentielle mène de cette définition à cette utilisation ...
- ... sans aucune redéfinition de la variable le long de cette exécution

DFG : le graphe (II)

```
void main() {
    int tab = [1, 2, 3, 4, 5], max, i;
    max =tab[0];
    i =0;
    while (  i < 5 ) {

       if (max <tab[ i]) {

          max=i;

          i=i+1;

       }

    }
    print ("%d",max);
 }
```

DFG : le graphe (II)



DFG : applications

- Compilateurs :
  - variables non-initialisées (mais utilisées)
  - variables non utilisées (mais définies)
  - optimisation de code (ré-arrangement des lignes de code, ...)
- Analyseurs statiques

Qualité des (suites de) tests

Le CFG et le DFG peuvent être utilisés pour définir des suites de tests. Les cas de test sont définis à partir de ces graphes.

La qualité d'une suite de tests est mesurée par une métrique définie à partir du CFG ou du DFG

La métrique mesure la qualité d'une **suite de cas de test** et non d'un **test isolé**

Le taux de couverture des différentes méthodes de test est définie à partir de cette métrique

Couverture de tests

- identifier la partie du code non exécutée lors d'un test (partie non couverte)
- mesurer le taux de couverture de la suite de test
- créer de nouveaux cas de test pour augmenter la couverture
- identifier les tests redondants

Sur quoi mesurer la couverture ?

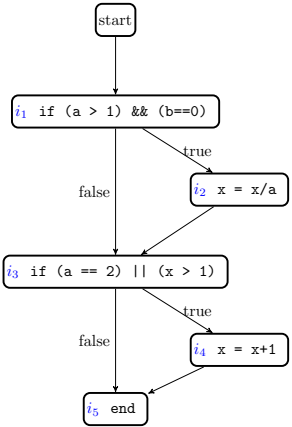
- code source
- code objet (bytecode)
- modèle abstrait
  - graphe de flot de contrôle
  - graphe de flot de données
- listes de point à vérifier (check points list)

Quelques métriques

- couverture des instructions (sommet du CFG) : mesure si chaque instruction est exécutée par au moins un cas de test
- couverture des arêtes du CFG : mesure si chacune des arêtes du CFG est couverte par l'exécution d'un cas de test au moins.
- couverture des chemins du CFG potentiellement impossible en cas de boucle : tester 0,1,2, ... itérations de la boucle (ou un nombre de fois fixé mais dépendant d'autres données du cas de test)
- couverture des chemins "simples" du CFG
- couverture des conditions : chaque condition de base (ne comportant pas de connecteurs logiques) prend toutes les valeurs possibles pour la suite de cas.
- couverture des décisions : chaque valeur booléenne des structures de contrôle est évaluée à chacune des valeurs possibles (V et F) dans la suite de cas de tests.
- couverture des conditions multiples : chaque combinaison des conditions d'une décision est évaluée.

Couvertures des instructions

Chaque instruction est exécutée au moins une fois



les instructions sont couvertes par l'unique cas de test : a $= 2$, b $= 0$, x $= 4$

Les deux arêtes étiquetées par $false$ ne sont pas couvertes par ce test.

Insuffisance de la couvertures des noeuds

```
int *p=NULL;
if (cond) {p = &x;}
*p=1;
```

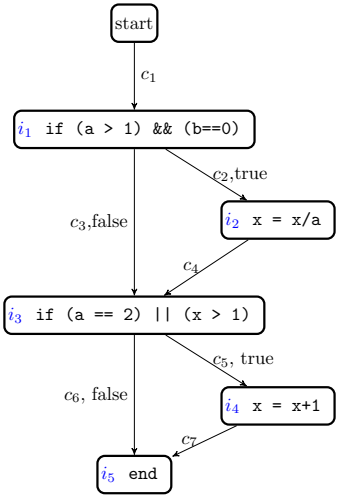
Pour un cas de test dont les données rendent `cond` vrai alors

- métrique : **OK**
- correction du programme : **KO**

La couverture des instructions est souvent **insuffisante**

- des chemins sont non parcourus
- peut ne pas tester la sortie d'une boucle
- insensibilité aux opérateurs constituant les conditions complexes des tests.

Couvertures des arêtes



On souhaite avoir une suite de tests dont les exécutions permettent de passer globalement par toutes les arêtes

les arêtes sont couvertes par les deux cas de test suivants :

- a $= 2$, b $= 0$, x $= 4$
- a $= 1$, b $= 1$, x $= 0$

exécutant les deux chemins

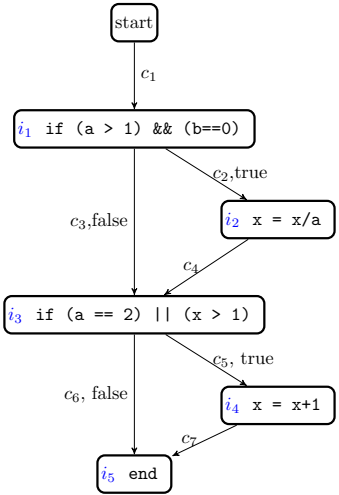
- c_1, c_2, c_4, c_5, c_7
- c_1, c_3, c_6

Remarque :

Couverture des arêtes $\Rightarrow$ couverture des sommets

Tous les chemins (simples) ne sont pas couverts

Couvertures des chemins simples



les chemins simples sont couverts par les quatres cas de test suivants :

- a = 2, b = 0, x = 4
- a = 1, b = 1, x = 0
- a = 3, b = 0, x = 0
- a = 1, b = 0, x = 4

exécutant les quatres chemins

- c_1, c_2, c_4, c_5, c_7
- c_1, c_3, c_6
- c_1, c_2, c_4, c_6
- c_1, c_3, c_5, c_7

Remarque :

Couverture des chemins simples $\Rightarrow$ couverture des arêtes

Couvertures des chemins simples (II)

```
if (x > 1) {
  if ((x < 0) && (y == 0)) {....}
  else {.....}
}
```

Si aucune valeur de donnée de test ne permet de couvrir une arête alors pb de développement

Couverture du DFG

La suite de test définit un ensemble de chemins C dans le graphe de flot de contrôle.

L'ensemble C couvre :

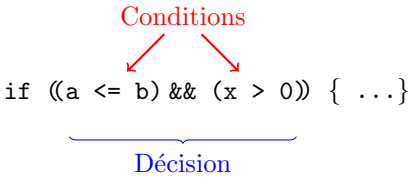
- toutes les définitions si **pour toute variable** x, **toute définition** de x d_x, **il existe** une utilisation de x u_x tel que (d_x, u_x) est une arête du graphe de flot de données et un chemin de C passant par d_x puis par u_x sans redéfinition de x.

  Intuitivement, toutes les définitions sont utilisées au moins une fois.

- toutes les utilisations si **pour toute variable** x, **toute définition** de x d_x, **toute utilisation** de x u_x tel que (d_x, u_x) est une arête du graphe de flot de données, **il existe** un chemin de C passant par d_x puis par u_x sans redéfinition de x.

  Intuitivement, toutes les utilisations accessibles par chaque définition.

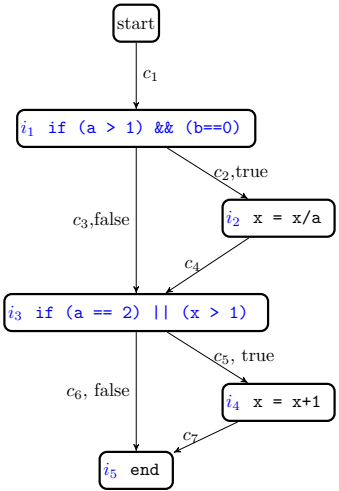
Conditions et décisions



- pour des données de test d'un cas de test, chaque condition s'évalue à vrai ou faux
- avec la valeur de chaque condition, une décision s'évalue à vrai ou faux

Couvertures des décisions

Chaque décision est évaluée à vrai et à faux dans la suite de tests



les décisions sont couvertes par les deux cas de test suivants :

- a = 2, b = 0, x = 4
  - la décision i_1 est évaluée à vrai
  - la décision i_3 est évaluée à vrai
- a = 1, b = 1, x = 0
  - la décision i_1 est évaluée à faux
  - la décision i_3 est évaluée à faux

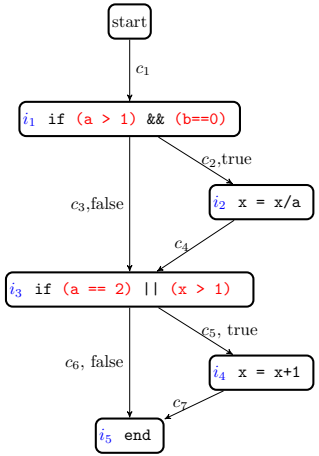
Couvertures des décisions (II)

- avantage : couvre toutes les instructions
- inconvénient : ne couvre pas tous les chemins
- évaluation globale de la décision mais pas des conditions la constituant

Equivalent à la couverture des arêtes

Couvertures des conditions (1/2)

Chaque condition d'une décision est évaluée à vrai et à faux dans la suite de tests



les décisions de chacune des conditions sont par les deux cas de test suivants :

	a>1	b==0	a==2	x>1
a=2 b=0 x=4	V	V	V	V
a=1 b=1 x=0	F	F	F	F

Couvertures des conditions (2/2)

`((cond1 && cond2) && cond3)`

Couverture des conditions par

- `cond1` = V, `cond2` = F, `cond3` = V
- `cond1` = F, `cond2` = V, `cond3` = F

Mais la décision V n'est pas couverte par cette suite de test.

Inversement `(cond1||cond2)`

Couverture des décisions par

- `cond1` = V, `cond2` = F
- `cond1` = F, `cond2` = F

Mais la condition `cond2` = F n'est pas couverte.

Couvertures des chemins

Impossible puisque

- le nombre de chemins peut être infini
- certains chemins peuvent ne pas être réalisables

Chemins simples (en arêtes) : chemins réalisables qui ne passent par chaque arête au plus une fois (dépliage des boucles 0 ou 1 fois)

Résumé pour les couvertures

Couvertures :

- Décisions *implique* Instructions
- Chemins simples *implique* Décisions
- Décisions *n'implique pas* Conditions
- Conditions *n'implique pas* Décisions
- Chemins *n'implique pas* Conditions

Couvertures des conditions multiples

Pour chaque décision, chaque combinaison booléenne des conditions est testée

- 2^n tests nécessaires pour n conditions, ce qui est beaucoup.
- possiblement moins en cas de court-circuit sur les opérateurs booléens
- mais difficile de trouver une suite de cas de test minimale dans ce cas.

Si les décisions ne sont pas couvertes alors problème dans le code.

Couvertures MC/DC (1/2)

Norme DO-178C de l'avionique

Modified Condition / Decision Coverage : couverture condition modifiée / décision

Objectifs : augmenter la couverture de tests par rapport à la couverture des décisions et à celle des conditions mais de manière plus économique que les conditions multiples.

Principe : on ne considère une combinaison de valeurs faisant varier une condition que si cette condition influence la décision

Couvertures MC/DC (2/2)

- Toutes les décisions sont couvertes par la suite de test
- Toutes les conditions sont couvertes par la suite de test
- Pour chaque condition, il existe deux cas de test qui
  - donnent des valeurs différentes à cette condition,
  - laissent les valeurs des autres conditions inchangées,
  - et produisent une décision différente

Couvertures MC/DC : exemple

`C1 && (C2 || C3)`

Décision

C1	C2	C3	D
V	V	V	V
F	V	V	F

Conditions multiples

C1	C2	C3	D
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	F
V	F	F	F
V	F	V	V
V	V	F	V
V	V	V	V

Couvertures MC/DC : exemple (II)

MC/DC

`C1 && (C2 || C3)`

C1	C2	C3	D
V	F	V	V
V	V	F	V
F	F	V	F
V	F	F	F

- couverture des décisions
- couverture des conditions

Couvertures MC/DC : exemple (II)

MC/DC

`C1 && (C2 || C3)`

C1	C2	C3	D
V	**F**	**V**	**V**
V	V	F	V
F	**F**	**V**	**F**
V	F	F	F

- couverture des décisions
- couverture des conditions

Couvertures MC/DC : exemple (II)

MC/DC

`C1 && ( C2 || C3)`

C1	C2	C3	D
V	F	V	V
V	**V**	**F**	**V**
F	F	V	F
V	**F**	**F**	**F**

- couverture des décisions
- couverture des conditions

Couvertures MC/DC : exemple (II)

MC/DC

`C1 && (C2 || C3 )`

C1	C2	C3	D
V	**F**	**V**	**V**
V	V	F	V
F	F	V	F
V	**F**	**F**	**F**

- couverture des décisions
- couverture des conditions

Couvertures MC/DC : exemple (III)

Cette solution n'est pas unique

C1	C2	C3	D
F	V	V	F
V	F	F	F
V	F	V	V
V	V	F	V
V	V	V	V

Minimalité d'une suite de test

Une suite de tests est **minimale** (pour un critère donné) s'il n'existe pas de suite comportant moins de tests et assurant la même couverture

Pour des raisons d'efficacité, on recherche des suites de test les plus courtes possible et pourquoi pas minimale.

- Si les conditions ne sont que des comparaisons entre variables et/ou constantes, alors trouver une suite de tests pour les métriques considérées est décidable (mais le problème sera sans doute NP-complet si la minimalité est requise)
- Si les conditions sont complexes et impliquent par exemple l'arithmétique (`x * y > 2`) alors ce problème peut être indécidable.

Boucles et infinité de chemins

Itération :

- tester 0 itération,
- tester une itération
- tester deux itérations
- tester un nombre "typique" d'itérations.

Tester les chemins **indépendants**.

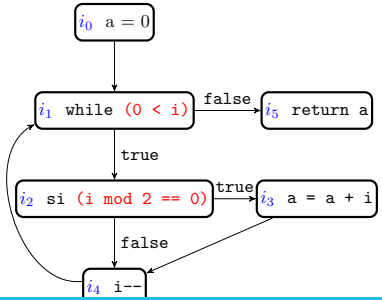
Chemins indépendants : soit $\mathcal{C}$ un ensemble de chemins. Un chemin c est indépendant vis-à-vis de $\mathcal{C}$ s'il existe au moins une arête c non couverte (non présente) dans les chemins de $\mathcal{C}$.

Chemins indépendants et nombre cyclomatique

Soit G un graphe de flot de contrôle avec V sommets et E arêtes. Le **nombre cyclomatique** de G est

$$\#G = |E| - |V| + 2 = \text{nb de décisions} + 1$$

Nombre maximum de chemins nécessaires pour couvrir l'ensemble des arêtes du graphes.



Chemins indépendants

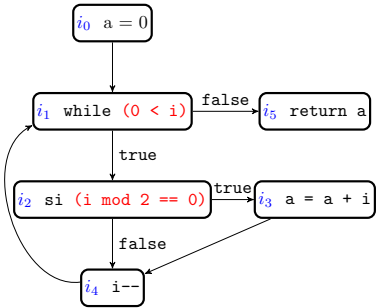
- $i_0 i_1 i_5$
- $i_0 i_1 i_2 i_3 i_4 i_1 i_5$
- $i_0 i_1 i_2 i_4 i_1 i_5$

Chemins indépendants et tests

Couvrir tous les chemins indépendants permet de couvrir toutes les décisions.

Construire un jeu de test de taille au minimum $\#G$ permettant de couvrir les chemins indépendants.

Produire un cas de test pour chaque chemin indépendant.



Cas de tests

- $i = 0$
- $i = 1$
- $i = 2$