Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Centre Universitaire de Mila

Département Math & Informatique

Génie logiciel II

CHAPITRE 5
Tests logiciels

HEDJAZ Sabrine

2019-2020

Tests Logiciels

1. Introduction

- Tester un logiciel : Exécuter le logiciel avec un ensemble de données réelles

  « Un programme sans spécifications est toujours correct »

- Il faut confronter résultats de l'exécution et résultats attendus
- Impossibilité de faire des tests exhaustifs
  - Ex : 2 entiers sur 32 bits en entrée : 2^{64} possibilités. Si $1ms$ par test, plus de 10^8 années nécessaires pour tout tester
- Choix des cas de tests :
  - Il faut couvrir *au mieux* l'espace d'entrées avec un nombre réduit d'exemples
  - Les zones sujettes à erreurs nécessitent une attention particulière

2. Tests fonctionnels

- Identification, à partir des spécifications, des sous-domaines à tester
  - Produire des cas de test pour chaque type de sortie du programme
  - Produire des cas de test déclenchant les différents messages d'erreur
- Objectif : Disposer d'un ensemble de cas de tests pour tester le programme complètement lors de son implémentation
- Test « boîte noire » parce qu'on ne préjuge pas de l'implémentation
  - Identification possible des cas de test pertinents avant l'implémentation
  - Utile pour guider l'implémentation

Tests Logiciels

2. Tests fonctionnels

- Principes de la boîte noire
  - Test du logiciel sans prendre connaissance de sa conception et de son implantation
  - Création de jeux de test et évaluation des résultats sur la base de la spécification
  - Test ce que le programme est supposé faire
  - Permet éventuellement de détecter les erreurs commises et les omissions
- Il existe deux approches principales pour la sélection de jeux de test de boîte noire
  - Partition du domaine des entrées en classes d'équivalence
  - Analyse des valeurs frontières

2.1. Partition du domaine des entrées en classe d'équivalence

- Déterminer le domaine D des valeurs d'entrée à partir des interfaces de la spécification (et du programme)
- Partitionner ce domaine en classes d'équivalence D_i de façon à ce que le programme se comporte (à peu près) de la même manière sur toute les valeurs d'entrée appartement à une même classe

$$D = \bigcup_{i=1..n} D_i \text{ où } \forall i \neq j, D_i \cap D_j = \{\ \}$$

Principe de couverture complète

- On construit un jeu de test en sélectionnant un représentant par classe d'équivalence

$$\text{Soit } D = \bigcup_{i=1..n} D_i \text{ où } \forall i \neq j, D_i \cap D_j = \{\ \}$$

$$(d_1 \in D_1, \ldots, d_n \in D_n) \text{ est un jeu de test}$$

2.1. Partition du domaine des entrées en classe d'équivalence

- Comment partitionner un domaine en classes d'équivalence ?
- Quelques lignes directrices
  - Lorsque le domaine d'entrée est un intervalle de valeurs
  - Lorsque le domaine d'entrée est un ensemble fini de valeurs discrètes (type énuméré)

2.1. Partition du domaine des entrées en classe d'équivalence

- Lorsque le domaine d'entrée est un intervalle de valeurs
  - Par exemple : soit D = [0, 5 000]
- On définit trois classes d'équivalence
  - Une classe d'équivalence pour les valeurs appartenant à D1={0 <=d <=5000} l'intervalle de valeurs valides
  - Une classe d'équivalence pour les valeurs d'entrées D2= {d< 0} inférieures à l'intervalle
  - Une classe d'équivalence pour les valeurs d'entrées D3 = {d > 5000} supérieures à l'intervalle
  - Un jeu de test valide serait : T = {10, -1, 6008}

2.1. Partition du domaine des entrées en classe d'équivalence

- Lorsque le domaine d'entrée est un ensemble fini de valeurs discrètes
- Par exemple
  - D = Couleurs = {bleu, blanc, rouge, vert, noir, orange}
- On définit deux classes d'équivalence
  - Une classe d'équivalence pour les valeurs d'entrée valides
  - Une autre classe d'équivalence pour les valeurs d'entrées invalides *devrait* aussi être définie
- Un jeu de test valide serait

D1 = {bleu, blanc, rouge}

$D_2 = \{d \in \text{Couleurs} \mid d \neq \text{bleu} \wedge d \neq \text{blanc} \wedge d \neq \text{rouge}\}$

2.1. Partition du domaine des entrées en classe d'équivalence

```
If x > y Then
  max := x
Else
  max := x        /* erreur: on voulait max:=y*/
End if;
```

- Partition : $D_1 = \{(x,y) \mid x > y\}$; $D_2 = \{(x,y) \mid x \leq y\}$;
- Jeux de test : {(3,2),(1,2)}; {(5,2),(4,7)}; etc.
  - Ce critère de sélection permet de trouver des jeux de test qui vérifient les deux branches de la condition : il est donc probable qu'on tombe sur l'erreur
  - Toutefois, tout jeu de test sélectionné selon la partition proposée ne garantit pas de révéler l'erreur dans le programme
  - Contre-exemples: {(4,2), (2,2)}; {(4,2), (3,3)}

2.2. Analyse des valeurs frontières

- Les erreurs typiques de programmation
  - Surviennent souvent aux frontières des classes d'équivalence
  - Peuvent être liés à des facteurs psychologiques
  - Les programmeur négligent souvent de traiter avec attention les cas se trouvant aux frontières des classes d'équivalence
  - Mauvais emplois des opérateurs <, >, >=, …

2.2. Analyse des valeurs frontières

- On procède comme dans le cas de la partition du domaine des entrées en classes d'équivalence (étape 1 et 2)
- On choisit les cas de test aux frontières des différentes classes d'équivalence
  - Ex. D = Naturels

$$D_1 = \{0 \leq d \leq 5000\}$$
$$D_2 = \{d < 0\}$$
$$D_3 = \{d > 5000\}$$

- Un jeu de test valide serait : T={-1, 0, 5 000, 5 001}

2.3. Avantages et inconvénients

- Avantages
  - Le jeu de test sélectionné peut garantir une bonne couverture du domaine des entrées du programme
  - Des oublis par rapport à la spécification de l'application peuvent être détectés
  - Lors de modifications du programme ne remettant pas en cause la spécification, il est possible de réutiliser le jeu de tests précédent pour valider la nouvelle version (régressions)
- Inconvénient
  - Ne donne pas d'information sur la localisation des erreurs

3. Tests structurels

Ce test consiste à analyser la structure interne du programme en déterminant les chemins minimaux. Afin d'assurer que:

- Toutes les conditions d'arrêt de boucle ont été vérifiées.
- Toutes les branches d'une instruction conditionnelle ont été testés.
- Les structures de donne interne ont été testées (pour assurer la validité).

La structures de contrôle se présente sous la forme d'un graphe dit graphe de flot.

3. Tests structurels

- Principes de la boîte blanche
  - On teste le programme en tenant compte de sa structure interne
  - On établit des jeux de test en fonction de la conception détaillée du programme
  - On teste ce que le programme fait
  - Permet éventuellement de détecter les erreurs commises… mais pas les omissions !

3. Tests structurels

- Pourquoi se baser sur la structure du programme pour le tester ?
  - Les erreurs ont tendance à se concentrer dans des chemins, instructions, conditions qui sont hors de l'« exécution normale »
  - On a tendance à croire qu'un chemin particulier a peu de chances d'être exécuté alors qu'il l'est très souvent
  - Les erreurs typographiques sont réparties au hasard

3 Tests structurels

- Critères de sélection de jeux de test
  - A. Critère de couverture des instructions
  - B. Critère de couverture des arcs du graphe de flot de contrôle
  - C. Critère de couverture des chemins indépendants du graphe de flot de contrôle

3. 1. Critère de couverture des instructions

- « Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des d ∈ T, chaque instruction de P est exécutée au moins une fois »
- Exemple : Algorithme d'Euclide
  - Trouver un jeu de test qui permet de couvrir toutes les instructions du programme.
  - Pour constituer les jeux de test, on va tenter de grouper dans des classes D_i les éléments du domaine d'entrée D qui activent les mêmes instructions dans P

```
Begin
  Read(x); Read(y);
  While x <> y Do
        If x > y Then x := x - y;
        Else y := y - x;
        End if;
  End while
  pgcd := x;
End
```

- $D_1 = \{(x,y) \mid x > y\}$; $D_2 = \{(x,y) \mid x < y\}$;
- Jeu de test: {(4,2), (2,4)

3. 1. Critère de couverture des instructions

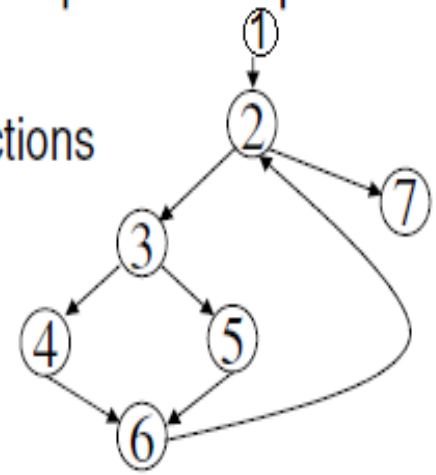
- Quelques limites…

```
Begin
    If x < 0 Then x := -x;
    End if;
    z := x;
End;
```

- Le jeu de test {-4} permet de couvrir toutes les instructions
  - Mais manque de complétude
  - Pas de test sur les entiers positifs !!!

3. 2. Critère de couverture des arcs du graphe de flot de contrôle

- Au lieu de s'intéresser aux instructions, on s'intéresse ici aux branchements de contrôle conditionnels dans un programme
- Un programme structuré peut être représenté par un graphe de flot de contrôle
  - Les sommets représentent les instructions
  - Les arcs représentent le flot de contrôle entre les instruction

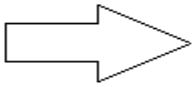


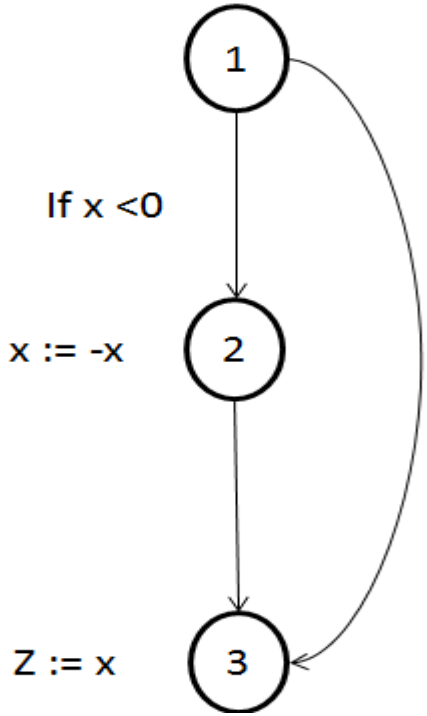
3. 2. Critère de couverture des arcs du graphe de flot de contrôle

- « Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des d ∈ T, chaque arc du graphe de flot de contrôle de P est traversé au moins une fois »
- Pour chaque instruction conditionnelle (If, While), tester le cas où la condition est vraie et celui où elle est fausse
  - Critère de sélection plus fort que la couverture des instructions

3. 2. Critère de couverture des arcs du graphe de flot de contrôle

```
Begin
    If x < 0 Then x := -x;
    End if;
    z := x;
End;
```





Le jeu de test {(-4,4)} permet de couvrir touts les arcs du graphe

3. 3. Critère de couverture des chemins indépendants

- « Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des d ∈ T, tous les 1-chemins du graphe de flot de P sont parcourus au moins une fois »
- Chemin = Séquence de nœuds et d'arcs dans le graphe de flot de contrôle, initiée depuis le nœud de départ jusqu'à un nœud terminal
  - Il peut y avoir plusieurs nœuds terminaux dans un programme
  - 1-chemin : chemin parcourant les boucles 0 ou 1 fois
  - Chemin indépendant : (1-)chemin qui parcourt au moins un nouvel arc par rapport aux autres chemins définis d'une base B (*i.e.*, introduit au moins une nouvelle instruction non parcourue)

3. 3. Critère de couverture des chemins indépendants

- Méthode de sélection des jeux de test
  - Construire le graphe de flot de contrôle de P
  - Déterminer l'indice de complexité cyclomatique du graphe de flot
  - Définir un ensemble de base B de chemins indépendants dans le graphe
  - Construire un jeu de test qui permettra l'exécution de chaque chemin de l'ensemble B

3. 3. Critère de couverture des chemins indépendants

- Chemin indépendant : chemin du graphe de flot de contrôle qui parcourt au moins un nouvel arc par rapport aux autres chemins définis
  - Une base comportant V(G) chemins nous assure de couvrir tous chemins indépendants du graphe de flot
  - Mais on ne couvre pas nécessairement tous les 1-chemins du graphe

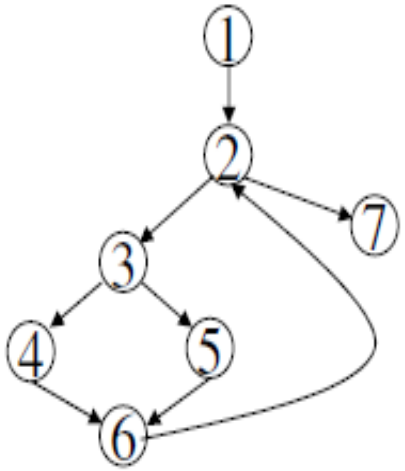
3. 3. Critère de couverture des chemins indépendants

- La représentation vectorielle d'un chemin est un vecteur qui compte le nombre d'occurrence de chaque arc
  - Ex. $Chemin_1 = (1, 0, 0, 0, 0, 0, 0, 1)$

- Un ensemble de chemins est linéairement indépendants si aucun ne peut être représenté par une combinaison linéaire des autres (en représentation vectorielle)

3. 3. Critère de couverture des chemins indépendants

Exemple



- Algorithme d' Euclide:
  - Par exemple
    - 1-2-7
    - 1-2-3-4-6-2-7 (nouveaux arcs: 2-3, 3-4, 4-6, 6-2)
    - 1-2-3-5-6-2-7 (nouveaux arcs: 3-5, 5-6)

3. 3. Critère de couverture des chemins indépendants

Exemple

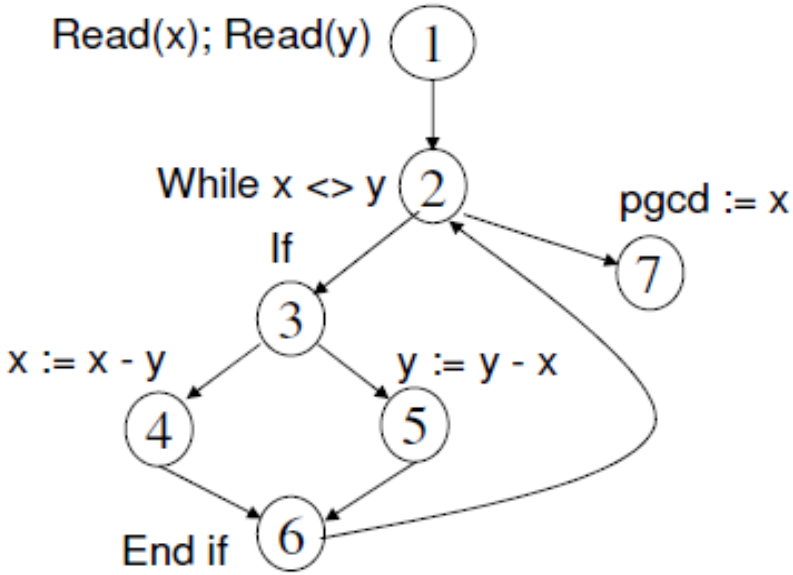
Sélection des jeux de tests:

- Pour chaque chemin indépendant de la base, on doit trouver un jeu de test qui permette de le traverser (en itérant possiblement sur certains segments du chemin)
  - Sélection difficile dans le cas de gros programmes
  - Ceci équivaut à
    - Résoudre un système de contraintes composé des nœuds prédicats qui se trouvent sur le chemin à parcourir
    - Attention : tous les chemins ne sont pas nécessairement satisfiables ! (Problème indécidable…)

3. 3. Critère de couverture des chemins indépendants

Exemple

Sélection des jeux de tests:



- Partitionnement
  - Chemin 1-2-7
    - $D_1 = \{(x,y) \mid x = y\}$
    - Cas de test : <x=3, y=3>
  - Chemin 1-2-3-4-6-2-7
    - $D_2 = \{(x,y) \mid x > y, x - y = y\}$
    - Cas de test : <x=8, y=4>
  - Chemin 1-2-3-5-6-2-7
    - $D_3 = \{(x,y) \mid x < y, x = y - x\}$
    - Cas de test : <x=3, y=6>
- Jeu de test : {<x=3, y=3>, <x=8, y=4>, <x=3, y=6>}

Tests Logiciels

Planification

Chapitre6