Vérification et Validation (V&V)

Cours et TP

Dina Irofti (EDF R&D)

Contents

Pourquoi est-il essentiel de tester ?

Exemple sonde spatiale Mars Climate Orbiter

- En septembre 1999, Mars Climate Orbiter, une sonde spatiale conçue et envoyée par la NASA, devait se mettre en orbite autour de la planète Mars et devenir le premier satellite météorologique interplanétaire.
- La sonde s'est écrasée en raison d'un malentendu dans les unités de mesure utilisées par deux modules créés par des groupes de logiciels distincts : un module pour calculer les données du propulseur en unités anglaises (le système impérial) a transmis les données à un autre module qui attendait les données en unités métriques.
- Il s'agit d'un défaut d'intégration très typique (mais en cette affaire extrêmement coûteux, tant en termes d'argent que de prestige).

Pourquoi est-il essentiel de tester ?

Exemple du lancement de la première fusée Ariane 5

- La fusée Ariane 5 a explosé le 37 secondes après le décollage en 1996.
- La cause : un problème non résolu d'une exception de conversion en virgule flottante dans une fonction du système de guidage inertiel.
- Il s'est avéré que le système de guidage ne pourrait jamais rencontrer cette exception lorsqu'elle est utilisée sur la fusée Ariane 4.
- Les développeurs de l'Ariane 5 ont tout à fait raisonnablement voulu réutiliser le système inertiel d'Ariane 4, mais personne n'a réanalysé le logiciel en tenant compte de la trajectoire de vol sensiblement différente de l'Ariane 5.
- De plus, les tests du système qui auraient permis de détecter le problème étaient techniquement difficiles à exécuter et n'ont donc pas été réalisés. Le résultat fut spectaculaire – et cher !

Pourquoi est-il essentiel de tester ?

Panne de courant nord-américaine de 2003

- La panne a commencé lorsqu'une ligne électrique a touché des arbres à cause de la dilatation des câbles causée par la chaleur ; la ligne s'est mise en sécurité en se coupant ; c'est ce qu'on appelle un défaut dans le secteur de l'énergie.
- D'autres lignes se sont également enfoncées dans les arbres et ont fini par s'éteindre, surchargeant d'autres lignes électriques, qui se sont ensuite coupées. Cet effet de cascade a finalement provoqué une panne d'électricité dans tout le sud-est du Canada et huit États du nord-est des États-Unis.
- Ceci est considéré comme la plus grande panne d'électricité dans l'histoire de l'Amérique du Nord, touchant 10 millions de personnes au Canada et 40 millions aux États-Unis, contribuant à au moins 11 décès et coûtant jusqu'à 6 milliards de dollars américains.

Vérification et validation

Vérification

Le processus consistant à déterminer si le produit d'une phase du processus de développement logiciel remplit les exigences établies lors de la phase précédente.

Validation

Le processus d'évaluation du logiciel à la fin de son développement pour garantir le respect des objectifs d'usage du logiciel.

Vérification : est-ce que les exigences sont satisfaites ?
Validation : est-ce que les attentes du client sont respectées ?

Cinq niveaux de tests (cf. Beizer)

- **Niveau 0** : Il n'y a aucune différence entre tester et déboguer.
- **Niveau 1** : Le but des tests est de montrer le comportement correct.
- **Niveau 2** : Le but des tests est de montrer que le logiciel ne fonctionne pas comme il faut.
- **Niveau 3** : Le but des tests n'est pas de prouver quoi que ce soit de spécifique, mais de réduire le risque lié à l'utilisation du logiciel.
- **Niveau 4** : Les tests sont une discipline mentale qui aide tous les professionnels de l'informatique développer des logiciels de meilleure qualité.

Défaut, erreur et défaillance

- **Défaut** (*Fault* en anglais) : un défaut statique dans le logiciel
- **Erreur** (*Error* en anglais) : un état interne incorrect qui est la manifestation d'un défaut
- **Défaillance** (*Failure* en anglais) : Comportement observable incorrect par rapport aux exigences ou par rapport à une autre description du comportement attendu

Comprendre les concepts défaut, erreur et défaillance sur un exemple

On souhaite écrire une fonction qui compte le nombre de zéros dans un *array* de type *int*.
Pour ceci, on écrit la fonction *numZero* avec un paramètre *x* qui renvoie un *int counter* avec le nombre de zéros dans *x*.

Comprendre les concepts défaut, erreur et défaillance

```
1
2  public class Example_numZero {
3
4    public static int numZero (int[] x)
5    {
6      int count = 0;
7      for (int i=1; i<x.length; i++)
8        if (x[i] == 0)
9          count++;
10     return count;
11   }
12   public static void main(String[] args) {
13     int[] x1 = {2,7,0};
14     System.out.println(numZero(x1));
15     int[] x2 = {0,7,2};
16     System.out.println(numZero(x2));
17   }
18
19 }
```

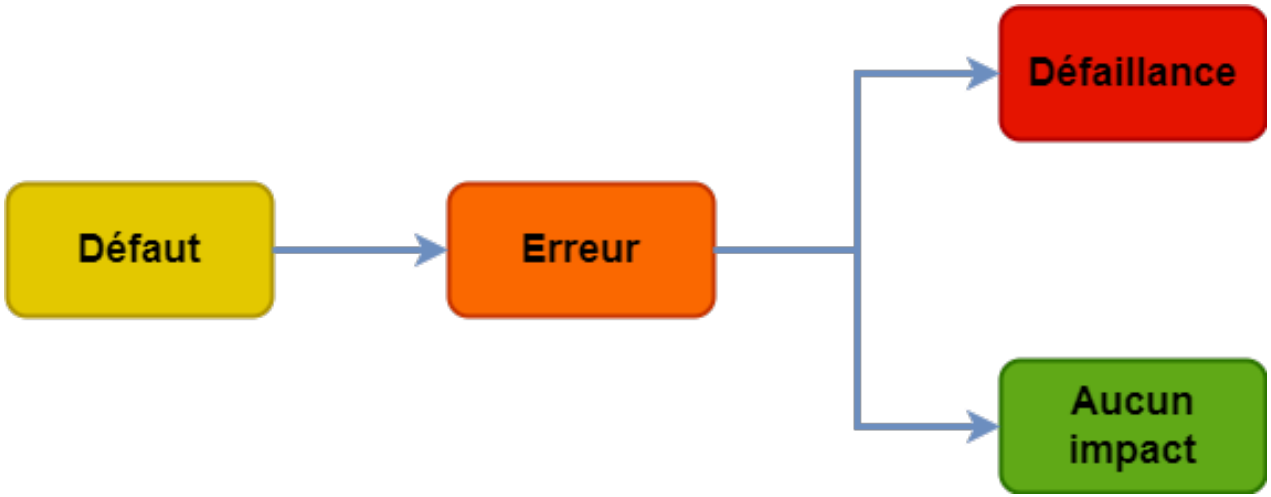
Comprendre les concepts défaut, erreur et défaillance

- Le défaut est dans la méthode *numZero* qui commence à chercher des zéros à l'index 1 au lieu de l'index 0, comme cela est nécessaire pour les tableaux en Java.
- Par exemple, *numZero ([2, 7, 0])* est correctement évalué à 1, tandis que *numZero ([0, 7, 2])* est évalué incorrectement à 0.
- Dans les deux tests, l'instruction qui comporte un **défaut** est exécutée.
- Bien que ces deux tests aboutissent à une **erreur**, seul la second test aboutit à une **défaillance**.

Pourquoi les deux tests aboutissent à une erreur

- L'état de numZero() se compose de valeurs pour les variables x, count, i et le compteur de programme (PC).
- Pour le premier test, l'état de la boucle for dès la toute première itération de la boucle est (x = [2, 7, 0], count = 0, i = 1, PC = « i < x.longueur"").
- Cet état est erroné précisément parce que la valeur de i devrait être nulle à la première itération.
- Cependant, puisque la valeur du *count* est par coïncidence correcte, l'état d'erreur ne se propage pas à la sortie et donc le logiciel ne présente pas de défaillance.
- Dans le deuxième, l'erreur est (x = [0, 7, 2], compte = 0, i = 1, PC = « i < x.longueur »), elle se propage à la variable count et est présente dans la valeur de retour de la méthode.

Take-home message



Bug

Le terme *bug* est souvent utilisé de manière informelle pour désigner les trois (défaut, erreur et défaillance) !

Conditions nécessaires pour pouvoir observer une défaillance

- **Accessibilité** : Le ou les emplacements du programme qui contiennent le défaut doivent être atteints
- **Infection** : L'état du programme doit être incorrect
- **Propagation** : l'état infecté doit provoquer une sortie ou un état final du programme incorrect
- **Révéler** : le testeur doit observer une partie de la partie incorrecte de l'état du programme

Types de test

- **Tests d'acceptation** : le logiciel est-il acceptable pour l'utilisateur ?
- **Tests du système** : tester la fonctionnalité globale du système
- **Tests d'intégration** : tester la façon dont les modules interagissent les uns avec les autres
- **Test de module (test développeur)** : Testez chaque classe, fichier, module, composant
- **Tests unitaires (tests développeurs)** : tester chaque unité (méthode) individuellement

Types de test : le cas POO

- **Tests inter-classes** : tester plusieurs classes ensemble
- **Tests intra-classe** : tester une classe entière sous forme de séquences d'appels
- **Tests inter-méthodes** : Tester des paires de méthodes dans la même classe
- **Tests intra-méthodes** : tester chaque méthode individuellement

Types de test dans la literature classique

- **Tests en boîte noire** : tests dérivés à partir de descriptions externes du logiciel, y compris les spécifications, les exigences et la conception.
- **Tests en boîte blanche** : tests dérivés à partir du code source interne du logiciel, y compris spécifiquement les branches, les conditions individuelles et les instructions.
- **Tests basés sur un modèle** : tests dérivés à partir d'un modèle du logiciel (tel qu'un diagramme UML)

Conception de tests basés sur un modèle

La conception de tests (en anglais *Test Design*) est le processus de conception de valeurs d'entrée qui testeront efficacement les logiciels.
Etapes :

1. **Conception des tests**
   - **Basé sur des critères** : Concevoir des valeurs de test pour satisfaire les critères de couverture ou autres objectifs d'ingénierie
   - **Basé sur l'humain** : Concevoir des valeurs de test basées sur la connaissance humaine du domaine
2. **Automatisation des tests** : Intégrer les valeurs de test dans des scripts exécutables
3. **Exécution des tests** : Exécutez des tests sur le logiciel et enregistrez les résultats
4. **Évaluation des tests** : Évaluer les résultats des tests

Critères de couverture

Même les petits programmes ont trop d'entrées pour tous les tester complètement !
Exemple : **private static double computeAverage (int A, int B, int C)**

- Sur une machine 32 bits, chaque variable a plus de 4 milliards de valeurs possibles
- Plus de 80 octillions (10^{48}) de tests possibles !!

Solution : Les critères de couverture offrent des moyens structurés et pratiques de rechercher l'espace d'entrée

Critères de couverture

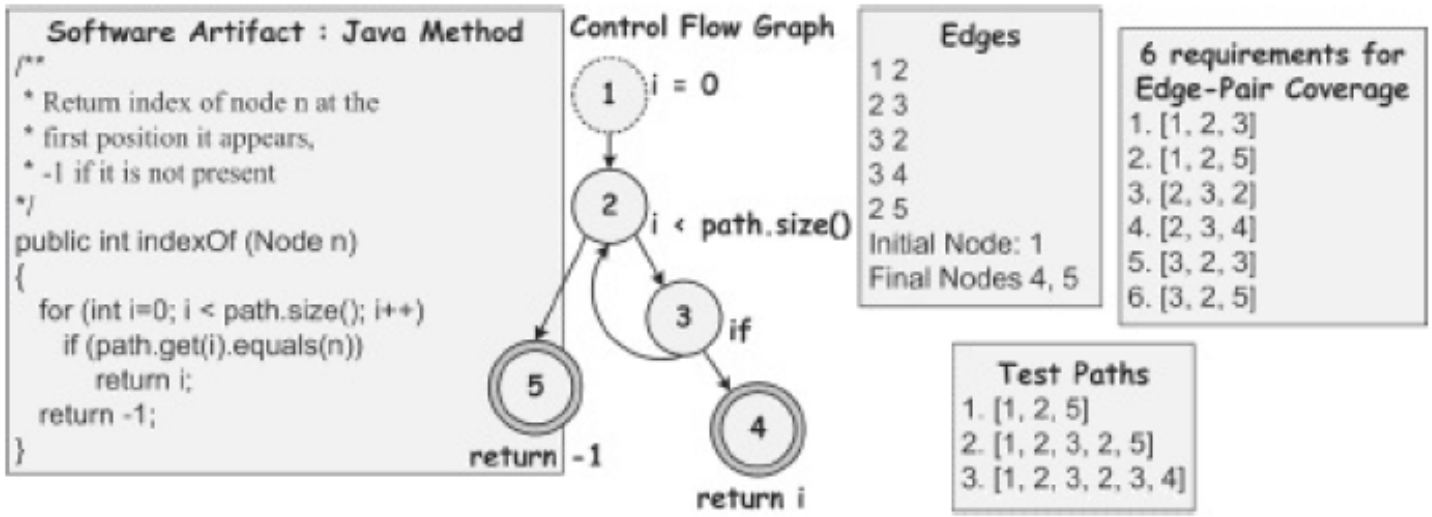
Critère de test : un ensemble de règles et un processus qui définissent les exigences du test

- Couvrir chaque instruction
- Couvrir toutes les exigences fonctionnelles

Exigences du test : éléments spécifiques qui doivent être satisfaits ou couverts lors du test

- Chaque instruction peut être une exigence de test
- Chaque exigence fonctionnelle peut être une exigence de test

Exemple test unitaire pour une methode Java



Automatisation des tests : quelques notions

L'**Automatisation des tests** : l'utilisation de logiciels pour

- contrôler l'exécution des tests,
- la comparaison des résultats réels aux résultats prévus,
- la configuration des conditions préalables aux tests.

La **Testabilité du logiciel** : la mesure dans laquelle un système ou un composant facilite l'établissement de critères de test et la réalisation de tests pour déterminer si ces critères ont été remplis.
L'**Observabilité** : dans quelle mesure il est facile d'observer le comportement d'un programme en termes de sorties, d'effets sur l'environnement et d'autres composants matériels et logiciels
La **Controlabilité** : dans quelle mesure il est facile de fournir à un programme les entrées nécessaires, en termes de valeurs, d'opérations et de comportements

Composants d'une séquence de test

- **Les valeurs d'entrée** (nécessaires pour terminer une exécution du logiciel testé)
- **Les résultats attendus** : le résultat qui sera produit par le test si le logiciel se comporte comme prévu

Un **oracle** de test utilise les résultats attendus pour décider si un test a réussi ou échoué.

Automatisation des tests : plusieurs séquences de test dans un script capable de s'exécuter automatiquement.
(JUnit pour Java)

Définir des tests basés sur des critères

Principe : définir un modèle du logiciel, puis trouver des moyens de le couvrir

Il existe plusieurs types de critères :

1. Caractérisation du domaine d'entrée :

$$A : \{0, 1, > 1\}, B : \{600, 700, 800\}, C : \{swe, cs, isa, infs\}$$

2. Graphes
3. Expressions logiques : $(not\ X\ or\ not\ Y)\ and\ A\ and\ B$
4. Structures syntaxiques (grammaires) :

$$if\,(x > y)\ then\ z = x - y;$$

$$else\ z = 2 * x;$$

Couverture

Couverture par rapport à un critère

Étant donné un ensemble d'exigences de test TR pour le critère de couverture C, un ensemble de tests T satisfait la couverture C si et seulement si pour chaque exigence de test tr dans TR, il existe au moins un test t dans T tel que t satisfait tr.

Exigences de test irréalisables : exigences de test qui ne peuvent pas être satisfaites :

- Il n'existe aucune valeur de scénario de test répondant aux exigences de test.
- Exemple : code mort
- La détection d'exigences de test irréalisables est formellement indécidable pour la plupart des critères de test

Une couverture à 100% est impossible en pratique !

Contents

Graphes

- Les graphes sont la structure la plus couramment utilisée pour les tests
- Quand on conçoit un test, l'idée est de couvrir au maximum le graphe
- Systèmes qui peuvent être transformés dans un graphe :
  - Graphe de flot de contrôle
  - Automate fini
  - Diagramme états-transitions
  - ...

Définition d'un graphe

- Un ensemble N de nœuds, N n'est pas vide
- Un ensemble N_0 de nœuds initiaux, N_0 n'est pas vide
- Un ensemble N_f de nœuds finaux, N_f n'est pas vide
- Un ensemble E d'arcs, chaque arc connecte un nœud à un autre

Chemin : Une séquence de nœuds (chaque paire de nœuds est un arc).
Longueur : Le nombre d'arcs (Un seul nœud est un chemin de longueur 0).
Sous-chemin : Une sous-séquence de nœuds dans un chemin p est un sous-chemin de p.

Chemins de test et graphes SESE

Chemin de test

- Un chemin de test est un chemin qui commence à un nœud initial et se termine à un nœud final.
- Les chemins de test représentent l'exécution des cas de test.
- Certains chemins de test peuvent être exécutés par de plusieurs tests
- Certains chemins de test ne peuvent être exécutés par aucun test

Graphes SESE (Single-entry single-exit)

N_0 et N_f ont seulement un nœud !

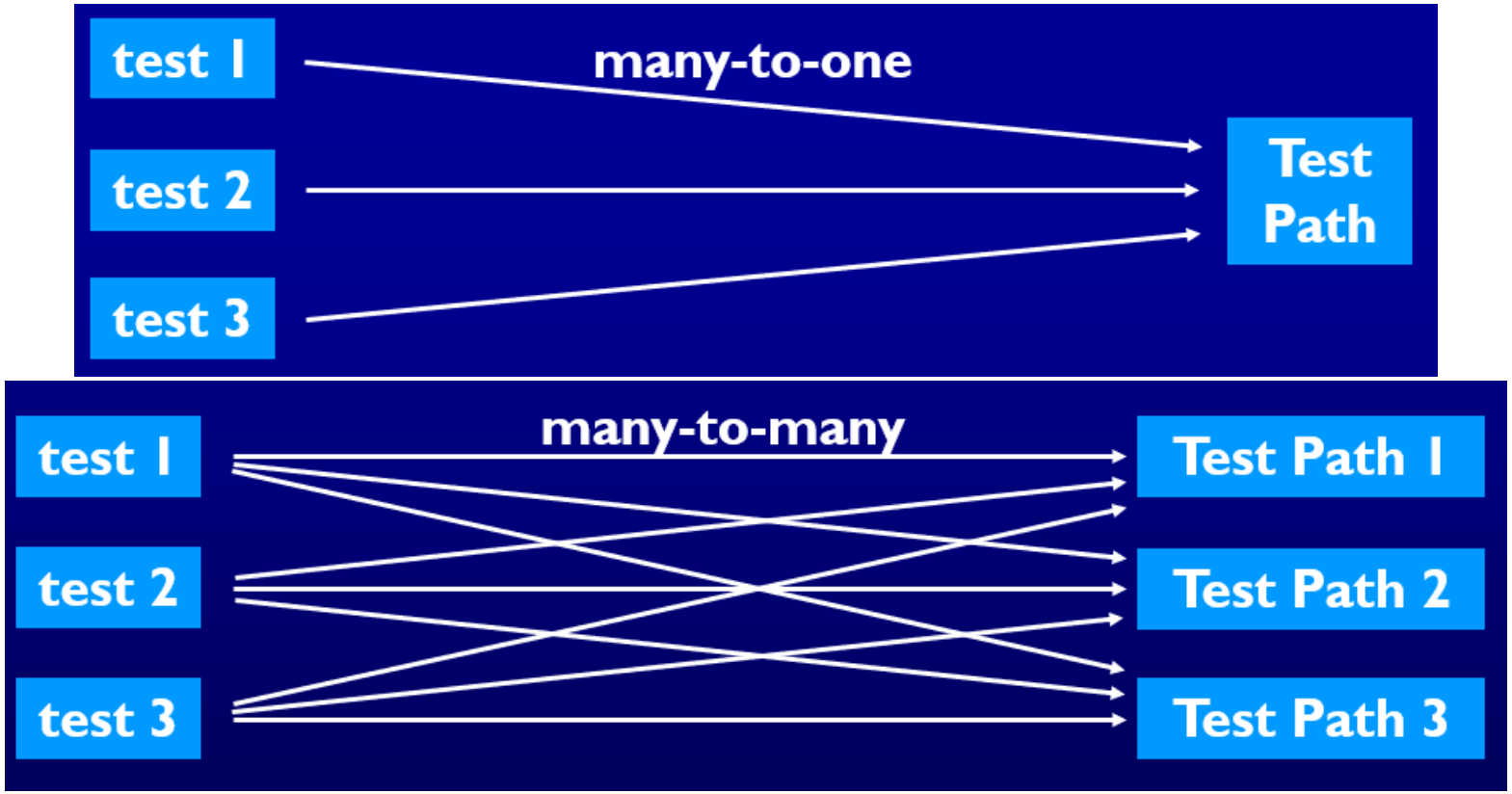
Parcours de graphe

Parcourir un graphe consiste à visiter ses arcs, en suivant les nœuds qui les relient.

Tests et chemins de test

- path (t) : Le chemin de test exécuté par le test t
- path (T) : L'ensemble des chemins de tests exécutés par l'ensemble de tests T
- Chaque test exécute seulement un seul chemin de test (execution à partir d'un nœud de départ à un nœud final)
- Un emplacement dans un graphe (nœud ou arc) peut être atteint depuis un autre emplacement s'il existe une séquence d'arcs du premier emplacement au deuxième.

Tests et chemins de test



Conception de tests pour la couverture des graphes

Comment utiliser des graphes pour le test :

- Développer un modèle du code automate ou du logiciel sous forme de graphe
- Concevoir des tests pour parcourir des ensembles spécifiques de nœuds, d'arcs ou de sous-chemins

Conception de tests pour la couverture des graphes

- **Exigences de test (TR)** : décrire les propriétés des chemins de test
- **Critère de test (C)** : règles qui définissent les exigences du test
- **Satisfaction** : Étant donné un ensemble TR d'exigences de test pour un critère C, un ensemble de tests T satisfait C sur un graphe si et seulement si pour chaque exigence de test tr dans TR, il existe un chemin de test dans path(T) qui répond à l'exigence de test tr du TR

Couverture des nœuds

Un critère simple : chaque nœud du graphe est parcouru.

Couverture des nœuds

L'ensemble de tests T satisfait la couverture de nœuds sur le graphe G ssi pour chaque nœud atteignable n dans N, il existe un chemin p dans path(T) tel que p visite n.

Autrement dit : TR contient chaque nœud atteignable dans G.

Couverture des arcs

La couverture des arcs est légèrement plus forte que la couverture des nœuds.

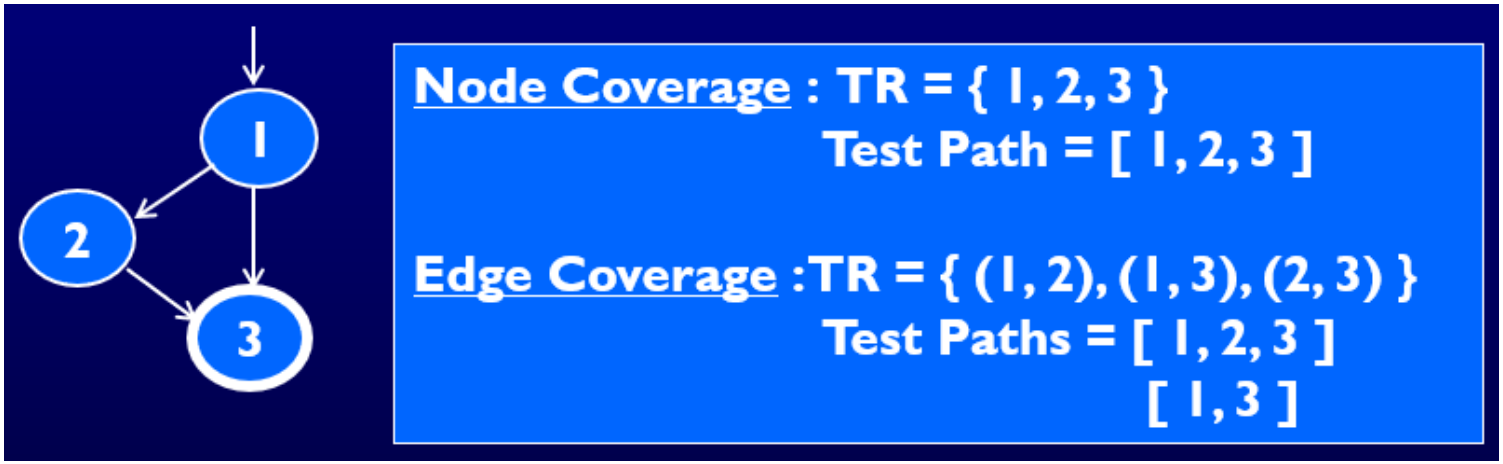
Couverture des arcs

TR contient chaque chemin atteignable de longueur 1 inclus dans G.

Couverture des nœuds et couverture des arcs

Remarque

La couverture des nœuds et la couverture des arcs ne sont différents que lorsqu'il y a un arc et un autre sous-chemin entre une paire de nœuds (comme dans une instruction « if-else »).

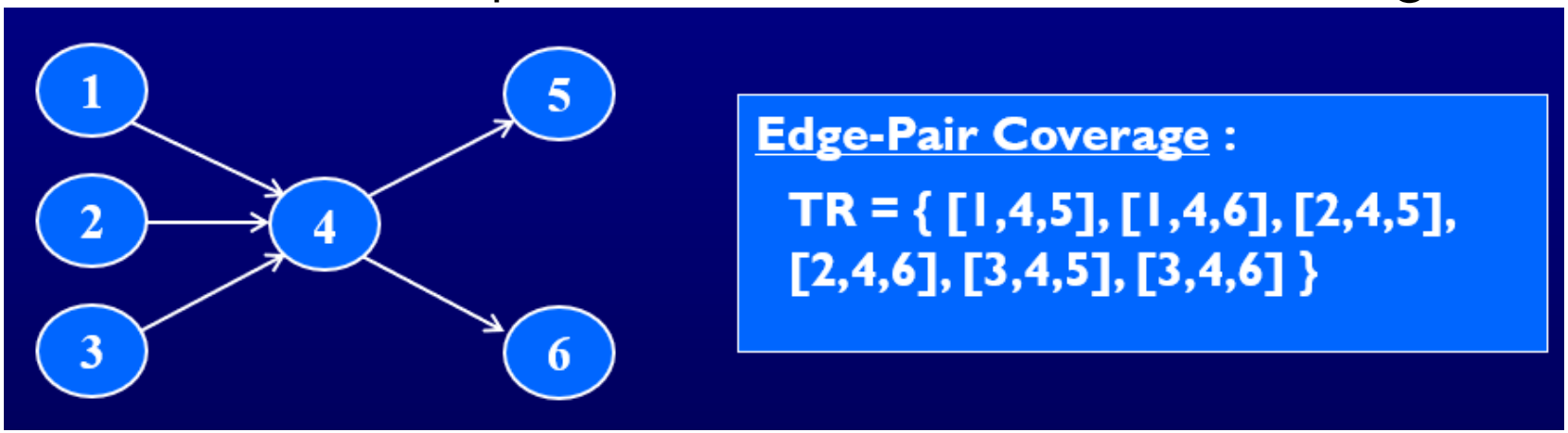


Couverture de plusieurs arcs

Couverture des paires des arcs

TR contient chaque chemin accessible d'une longueur 2 inclus dans G.

Ce critère nécessite : paires des arcs ou sous-chemins de longueur 2.



Extension naturelle : tous les chemins de tous longueur

Couverture de plusieurs arcs

Couverture complète des chemins

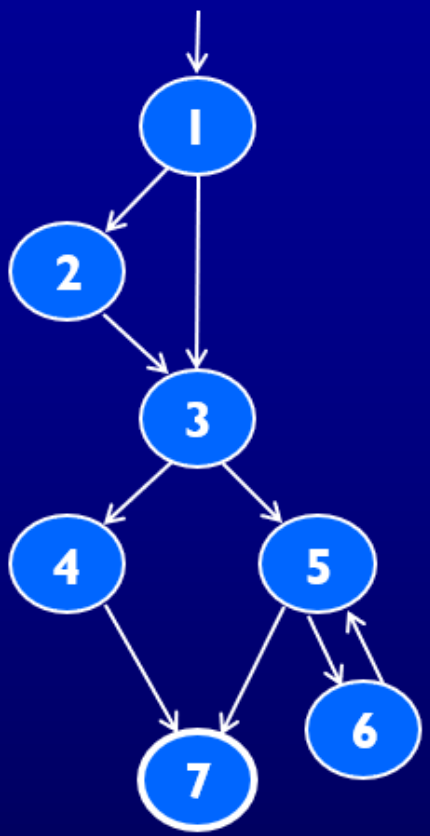
TR contient tous les chemins dans G.

Malheureusement, cela est impossible si le graphe comporte une boucle, donc un compromis faible oblige le testeur à décider quels chemins tester.

Couverture des chemins spécifiés

TR contient un ensemble S de chemins de test, où S est fourni en tant que paramètre.

Exemple



Node Coverage

TR = { 1, 2, 3, 4, 5, 6, 7 }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 6, 5, 7]

Edge Coverage

TR = { (1,2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 7), (5, 6), (5, 7), (6, 5) }

Test Paths: [1, 2, 3, 4, 7] [1, 3, 5, 6, 5, 7]

Edge-Pair Coverage

TR = {[1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6], [3,5,7], [5,6,5], [6,5,6], [6,5,7] }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 3, 4, 7] [1, 3, 5, 6, 5, 6, 5, 7]

Complete Path Coverage

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 2, 3, 5, 6, 5, 6] [1, 2, 3, 5, 6, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 6, 5, 7] ...

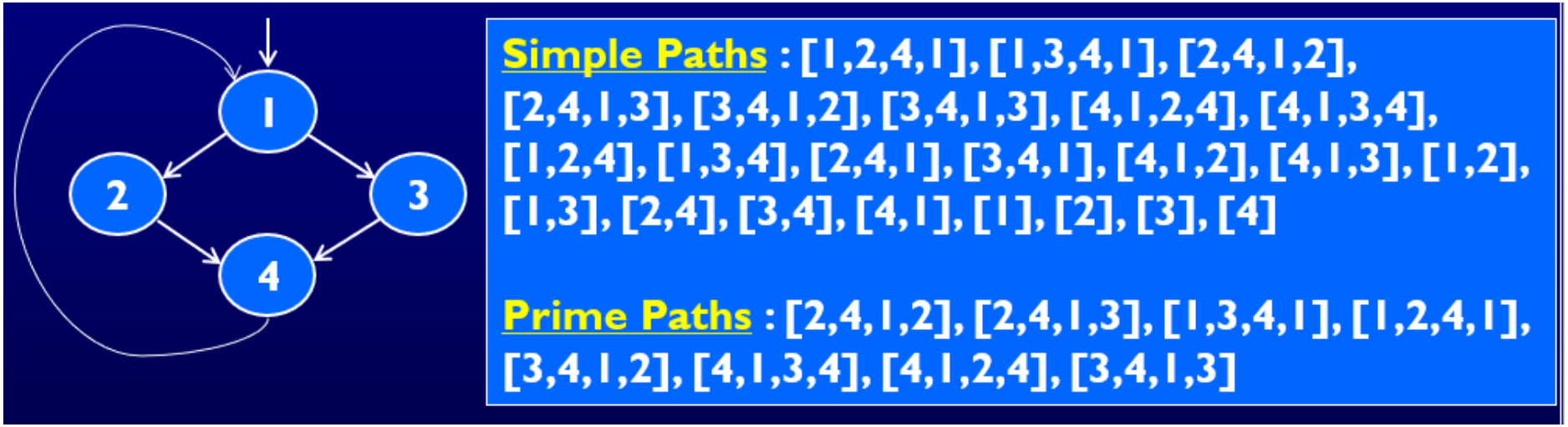
Comment faire quand le graphe comporte des boucles

- Si un graphe contient une boucle, il a un nombre infini de chemins
- Ainsi, la couverture complète des chemins n'est pas réalisable
- La couverture des chemins spécifiés n'est pas satisfaisante car les résultats sont subjectifs et varient selon le testeur
- Approches pour gérer les boucles :
  - Années 1970 : Exécuter la boucle une fois
  - Années 1980 : Exécuter chaque boucle exactement une fois
  - Années 1990 : Exécuter des boucles 0 fois, une fois, plus d'une fois
  - Années 2000 : Introduction d'autres types de chemin

Chemin simple et chemin prime

- **Chemin simple** : Un chemin du nœud n_i à n_j est simple si aucun nœud n'apparaît plus d'une fois, sauf le premier et le dernier nœuds qui peuvent être identiques.
  - pas de boucles internes
  - une boucle est un chemin simple
- **Chemin prime** : un chemin simple qui n'apparaît pas comme un sous-chemin d'un autre chemin simple

Chemin simple et chemin prime



Couverture des chemins primes

Couverture des chemins primes

TR contient chaque chemin prime dans G.

- Un critère simple, élégant et fini
- Les boucles peuvent être exécutées ou ignorées
- Comporte des chemins de longueur 0, 1 etc. $\Rightarrow$ englobe la couverture des nœuds et des arcs
- Englobe la couverture des paires des arcs

Références

- IEEE Standard for Software and System Test Documentation. Institute of Electrical and Electronic Engineers, New York. IEEE Std 829-2008.
- Beizer, B. (1990). Software Testing Techniques. Van Nostrand Reinhold, Inc, New York, NY, 2nd edition.
- Paul Ammann and Jeff Offutt, Introduction to Software Testing, 2nd edition.

Contents

Exercice 1

lastZero()

Tester la fonction suivante. Répondre aux questions :

1. Expliquez ce qui ne va pas avec le code donné. Décrivez le défaut précisément en proposant une modification du code.
2. Si possible, donnez un scénario de test qui n'exécute pas le défaut. Sinon, expliquez brièvement pourquoi.
3. Si possible, donnez un scénario de test qui exécute le défaut, sans l'apparition d'une erreur. Sinon, expliquez brièvement pourquoi.
4. Si possible, donnez un scénario de test qui entraîne une erreur, mais sans une défaillance visible. Sinon, expliquez brièvement pourquoi.
5. Pour le scénario de test donné, décrivez le premier état d'erreur. Assurez-vous de décrire l'état complet (avec le program counter).

Exercice 1

lastZero()

- Effectuez votre réparation et vérifiez que le test donné maintenant produit le résultat attendu. Soumettre une capture d'écran ou autre preuve que votre nouveau programme fonctionne.

Exercice 1

```
1
2  public class TP_lastZero {
3
4    public static int lastZero (int[] x)
5    {
6      for(int i=0; i<x.length; i++)
7        if(x[i]==0)
8          return i;
9      return -1;
10   }
11
12   public static void main(String[] args) {
13     int[] x1 = {0,1,0};
14     System.out.println(lastZero(x1)); // Expected 2
15
16   }
17
18 }
```

Exercice 2

oddOrPos()

Tester la fonction suivante. Répondre aux questions :

1. Expliquez ce qui ne va pas avec le code donné. Décrivez le défaut précisément en proposant une modification du code.
2. Si possible, donnez un scénario de test qui n'exécute pas le défaut. Sinon, expliquez brièvement pourquoi.
3. Si possible, donnez un scénario de test qui exécute le défaut, sans l'apparition d'une erreur. Sinon, expliquez brièvement pourquoi.
4. Si possible, donnez un scénario de test qui entraîne une erreur, mais sans une défaillance visible. Sinon, expliquez brièvement pourquoi.
5. Pour le scénario de test donné, décrivez le premier état d'erreur. Assurez-vous de décrire l'état complet (avec le program counter).

Exercice 2

oddOrPos()

- Effectuez votre réparation et vérifiez que le test donné maintenant produit le résultat attendu. Soumettre une capture d'écran ou autre preuve que votre nouveau programme fonctionne.

Exercice 2

```java
public class TP_oddOrPos {

  public static int oddOrPos(int[] x)
  {
    int count = 0;
    for(int i=0; i<x.length; i++)
      if(x[i]%2==1 || x[i]>0)
        count++;
    return count;
  }

  public static void main(String[] args) {
    int[] x1 = {-3, -2, 0, 1, 4};
    System.out.println(oddOrPos(x1)); // Expected 3
  }

}
```