UNIVERSITÉ PARIS SUD

L3 informatique et MIAGE

Année 2016-2017

Génie logiciel avancé

Test de mutations

Delphine Longuet

delphine.longuet@lri.fr

http://www.lri.fr/~longuet/Enseignements/16-17/L3-GLA

Qualité d'une suite de tests

Suite de tests valide

Les tests n'échouent que sur des programmes incorrects

Pas de faux positif, les fautes trouvées sont réelles

Suite de tests complète :

Les tests ne réussissent que sur des programmes corrects

Pas de faux négatif, si un programme contient une faute, elle est trouvée

Validité indispensable, complétude impossible en pratique

Comment assurer quand même une certaine qualité des tests ?

Mesures de la qualité des tests

Critères de couverture fonctionnels :

- couverture des exigences fonctionnelles
- couverture des domaines de définition des paramètres et de leurs limites (analyse partitionnelle + test aux limites)
- couverture des couples de paramètres (test pairwise)

Critères de couverture structurels :

- couverture des méthodes, classes, modules
- couverture des instructions, des décisions, des boucles
- couverture des définitions-utilisations de variables

Mesures de la qualité des tests

```
boolean contains(int[] t, int n, int size, int v) {
  int i = 0;
  while (i < size) {
    if (tab[i] == v) {
      return true;
    }
    i++;
  }
  return false;
}
```

Objectif	Données d'entrée				Résultat attendu
	t	n	size	v	
tableau vide	[_,_,_,_]	4	0	32	false
élément absent	[3,1,8,_,_]	5	3	2	false
élément présent	[4,6,2,9,_]	5	4	6	true

Couverture : toutes les instructions, toutes les décisions, passe 0 et plusieurs fois dans la boucle...

Mais ces tests détectent-ils toutes les fautes possibles ?

Test de mutations

Principe

1. Création d'un ensemble de mutants : application d'un ensemble de modifications syntaxiques (mutations) sur le programme original
2. Exécution des tests sur les mutants
3. Analyse des résultat des tests pour chaque mutant :
   - Échec : mutant tué, donc tests adéquats pour trouver la faute simulée par le mutant
   - Succès : mutant survivant

Deux raisons à la survie d'un mutant :

- Tests incomplets : manque un test qui cible cette faute
- Mutant équivalent : même comportement du mutant et du programme original

Exemples d'opérateurs de mutation

	Original	Mutant

Changer un opérateur arithmétique par un autre	`i + j`
	`i++`
	`-i`
		`i - j`
		`i--`
		`i`

Changer un opérateur relationnel par un autre	`i < j`	`i <= j`
Changer un opérateur booléen par un autre	`a && b`	`a || b`
Changer une constante par une constante du même type	`int i = 32`	`int i = 33`
Changer une variable par une variable du même type	`int i,j,k;` `i = j + 1`	`int i,j,k;` `i = k + 1`
Changer une variable par une constante du même type	`int i,j;` `i = j`	`int i,j;` `i = 1`

Mutants

Mutation : Modification syntaxique n'empêchant pas la compilation

Mutant : Programme obtenu par l'application d'un opérateur de mutation sur le programme original

Instruction	Mutants
`if a > 8 then x = y+1`	`if a >= 8 then x = y+1`
	`if a < 8 then x = y+1`
	`if a > 8 then x = x+1`
	`if a > 8 then x = y-1`
	`if a > 8 then x = 1`

Remarque : Très grand nombre de mutants pour un programme

Mutants

Objectifs :

- Simuler des fautes classiques
- Modifier le comportement du programme

Si comportement modifié, un test doit trouver la faute : exécuter une instruction ne suffit pas à la tester, il faut vérifier son impact sur le comportement du programme

But du test : Tuer les mutants

Score = nb mutants tués / nb mutants

Exemple

Programme original P

```
bool contains(int[] t,int n,
              int size,int v) {
  int i = 0;
  while (i < size) {
    if (tab[i] == v) {
      return true;
    }
    i++;
  }
  return false;
}
```

Mutant M

```
bool contains(int[] t,int n,
              int size,int v) {
  int i = 0;
  while (i <= size) {
    if (tab[i] == v) {
      return true;
    }
    i++;
  }
  return false;
}
```

Objectif	Données d'entrée				Résultat attendu	P	M
	t	n	size	v			
tableau vide	[_,_,_,_]	4	0	32	false	✔	✔
élément absent	[3,1,8,_,_]	5	3	2	false	✔	✔
élément présent	[4,6,2,9,_]	5	4	6	true	✔	✔

Exemple

Programme original P

```
bool contains(int[] t,int n,
              int size,int v) {
  int i = 0;
  while (i < size) {
    if (tab[i] == v) {
      return true;
    }
    i++;
  }
  return false;
}
```

Mutant M

```
bool contains(int[] t,int n,
              int size,int v) {
  int i = 0;
  while (i <= size) {
    if (tab[i] == v) {
      return true;
    }
    i++;
  }
  return false;
}
```

Objectif	Données d'entrée				Résultat attendu	P	M
	t	n	size	v			
tableau vide	[_,_,_,_]	4	0	32	false	✔	✔
élément absent	[3,1,8,_,_]	5	3	2	false	✔	✔
élément présent	[4,6,2,9,_]	5	4	6	true	✔	✔
élément absent et tableau plein	[2,9,0,1]	4	4	8	false	✔	✘

Comparaison avec les critères de couverture

Opérateurs de mutation instables : mutants obtenus facilement tués par suite de tests couvrant toutes les instructions ou toutes les décisions

Remplacer `return true` par `return false`

↳ Critère toutes les instructions suffit

Remplacer `if (a < 3 && b = 5)` par `if (true)`

↳ Critère toutes les décisions suffit

Opérateurs de mutation stables : mutants obtenus survivent aux tests couvrant les instructions et les décisions

Mutants intéressants car représentant fautes potentielles non détectées

Comparaison avec les critères de couverture

Opérateurs de mutation instables : mutants obtenus facilement tués par suite de tests couvrant toutes les instructions ou toutes les décisions

Opérateurs de mutation stables : mutants obtenus survivent aux tests couvrant les instructions et les décisions
Mutants intéressants car représentant fautes potentielles non détectées

En général, couverture atteinte par élimination des mutants supérieure à celle atteinte par critères structurels classiques (instructions, décisions, conditions multiples, définitions, utilisations...)

En pratique

Mutants très nombreux : du même ordre de grandeur que le nombre de lignes du programme original

Générer + compiler + tester les mutants : très coûteux en temps et en espace mémoire

Exemples d'optimisations :

- Muter le code compilé
- Exécuter seulement les tests qui couvrent la ligne mutée
- Ne pas jouer les tests restants si mutant tué

Mutants équivalents

Mutant ne pouvant pas être détecté par un test :

- mutation non accessible (code mort)
- mutation équivalente (`a < b` et `a <= b` avec `a` toujours différents de `b`)
- mutation d'une ligne sans conséquence (variable jamais utilisée)

Mutants équivalents empêchent score 100%

Détecter les mutants équivalents : problème indécidable

Mais certaines méthodes d'analyse de code permettent d'en repérer certains

Utilisation de la méthode

Seule : tuer les mutants devient l'objectif des tests, chaque test est conçu pour tuer un mutant donné.

- Très coûteux en exécution, et données de test plutôt difficiles à trouver puisque pas liées à un critère strictement structurel

Pour mesurer la qualité :

- Si aucun critère de couverture n'a été vérifié au préalable, peut être très coûteux et demander l'écriture de nombreux tests complémentaires
- Si suite de tests couvre déjà toutes les instructions au moins, beaucoup de mutants tués facilement et mutants restants intéressants

Conclusion

Avantages :

- Plus puissant que nombre de critères structurels
- Très bien corrélé à la découverte de fautes

Inconvénients : Nécessité d'un très grand nombre de mutants

- Beaucoup sont tués facilement (environ 90%)
- Intérêt de la méthode : élimination des 5-15% restants

Difficile et coûteux, mais grande qualité des tests générés

Utilisation préconisée pour la mesure de qualité

Démo avec CodeCover

CodeCover : outil de mesure de couverture de code (`codecover.org`)

Disponible comme plug-in Eclipse[1] : `http://update.codecover.org/`

Critères de couverture mesurés `codecover.org/features/coverage.html`

- Statement coverage : toutes les instructions
- Branch coverage : toutes les décisions
- Term coverage : toutes les conditions multiples
- Loop coverage : tous les chemins passant 0 fois, 1 fois, et plusieurs fois dans les boucles

[1] Plus maintenu mais encore fonctionnel sous Eclipse Kepler

Démo avec PIT

PIT : outil de test de mutation pour Java (`pitest.org`)

Disponible comme plug-in Eclipse :

`http://eclipse.pitest.org/release/`

Opérateurs de mutation disponibles `pitest.org/quickstart/mutators/`

- Conditionals Boundary Mutator (<, <=, >, >=)
- Negate Conditionals Mutator (== par !=, < par >=)
- Math Mutator (+ par -, * par /, % par *)
- Increments Mutator (`i++` par `i--`)
- Invert Negatives Mutator (`-i` par `i`)
- Return Values Mutator (mute les valeurs renvoyées dans `return`)
- Void Method Calls Mutator (enlève les appels aux méthodes de type `void`)