Cours de Test Logiciel

Leçon 2 : Sélection de Tests

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

Rappels

Vérification et Validation : entre 30% - 50% du coût de développement de logiciels

Test = méthode la plus utilisée de V & V

- une des difficultés principales : sélection des cas de tests
- notion de critères de (sélection de) test

Trois grandes familles de sélection de test

- test boîte noire (BN)
- test boîte blanche (BB)
- test aléatoire

Rappel : Critères de tests (2)

Boîte Noire : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèle formel ou semi-formel

Boîte Blanche : à partir du code

Probabiliste : domaines des entrées + arguments statistiques

Rappel : Critères de test

Sujet central du test

Tente de répondre à la question : “qu’est-ce qu’un bon jeu de test ?”

Plusieurs utilisations des critères :

- guide pour choisir les CT/DT les plus pertinents
- évaluer la qualité d’un jeu de test
- donner un critère objectif pour arrêter la phase de test

Quelques qualités atttendues d’un critère de test :

- bonne corrélation au pouvoir de détection des fautes
  // au moins partiellement
- concis
- automatisable

Test en "boîte noire"

- Ne nécesite pas de connaître la structure interne du système
- Basé sur la spécification de l'interface du système et de ses fonctionnalités : pas trop gros
- Permet d'assurer la conformance spéc - code, mais aveugle aux défauts fins de programmation
- Pas trop de problème d'oracle pour le CT, mais problème de la concrétisation
- Approprié pour le test du système mais également pour le test unitaire
- Méthodes de test BN :
  - Test des domaines d'entrées
    - partition des domaines
    - test combinatoire
    - + test aux limites
  - Couverture de la spécification
  - Test ad hoc (error guessing)

Test en "boîte blanche"

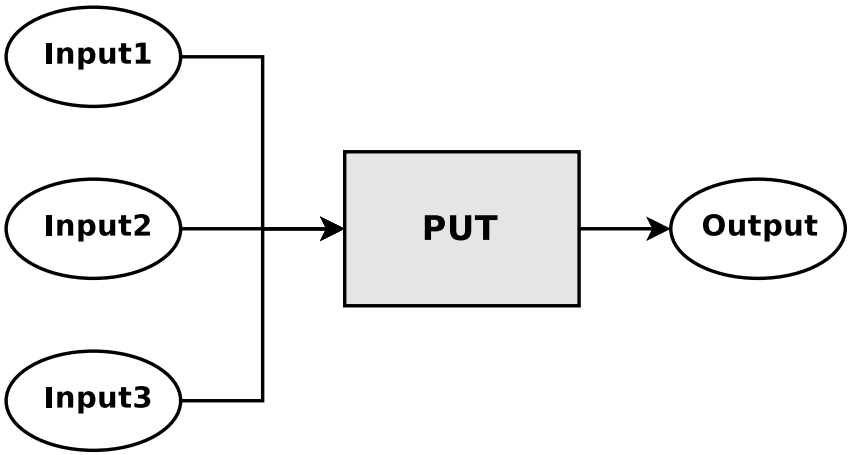
voir plus tard

- La structure interne du système doît être accessible
- Se base sur le code : très précis, mais plus "gros" que spéc
- Conséquences : DT potentiellement plus fines, mais très nombreuses
- Pas de problème de concrétisation, mais problème de l'oracle
- Sensible aux défauts fins de programmation, mais aveugle aux fonctionnalités absentes
- Méthodes de test BB (cf plus tard) :
  - Couverture du code (différents critères)
  - Mutations

Plan

- BN : test combinatoire
- BN : test des partitions
- BN : couverture fonctionnelle
- BN : test ad hoc
- Discussion

Test des domaines d'entrées




```
outType function-under-test(inType x, inType y);
```

Constat : test exhaustif souvent impratiquable

- espace des entrées non borné / paramétré / infini (BD, pointeurs, espace physique, etc.)
- simplement deux entiers 32 bits : 2^{64} possibilités

Test Combinatoire

Test combinatoire = test exhaustif sur une sous-partie (bien identifiée) des combinaisons possibles des valeurs d'entrée

Approche pairwise : sélectionner les DT pour couvrir toutes les paires de valeurs

- observation 1 : # paires bcp plus petit que # combinaisons
- observation 2 : un seul test couvre plusieurs paires
- # DT diminue fortement par rapport à test exhaustif

Remarque : on peut étendre à t-uplet, t fixé

- plus de tests, meilleur qualité
- ne semble guère intéressant en pratique

Test Combinatoire (2)

Hypothèse sous-jacente :

- majorité des fautes détectées par des combinaisons de 2 valeurs de variables
  - semble ok en pratique

Utile quand : beaucoup d'entrées, chacune ayant un domaine restreint

- typiquement : GUI (menus déroulants), interface "ligne de commande" avec de nombreux paramètres, tests de configuration (cf exos)
- très utile aussi en addition au test partitionnel (cf. ci-après)

Test Combinatoire (3)

Exemple : 3 variables booléennes A, B ,C

Nombre de combinaisons de valeurs / tests : $2^3 = 8$

Nombre de paires de valeurs (= nb paires de variables $\times$ 4) : 12

- (A=1,B=1), (A=1,B=0), (A=1,C=1), (A=1,C=0)
- (A=0,B=1), (A=0,B=0), (A=0,C=1), (A=0,C=0)
- (B=1,C=1), (B=1,C=0)
- (B=0,C=1), (B=0,C=0)

IMPORTANT : le DT (A=1,B=1,C=1) couvre 3 paires, mais 1 seule combinaison

- (A=1,B=1), (A=1,C=1), (B=1,C=1)

Ici 6 tests pour tout couvrir :

- (0,0,1), (0,1,0), (1,0,1), (1,1,0) couvrent presque tout, sauf (*,0,0) et (*,1,1)
- on ajoute (1,0,0) et (1,1,1)

Test Combinatoire (4)

Sur de plus gros exemples avec N variables à M valeurs :

- nb combinaisons : M^N
- nb paires de valeurs : $\approx M^2 \times N(N-1)/2$
- un test couvre au plus $N(N-1)/2$ paires de valeurs

On peut espérer tout couvrir en M^2 tests plutôt que M^N

- indépendant de N
- plus sensible à la taille des domaines qu'au nombre de variables

Attention : trouver un ensemble de tests de cardinal minimal pour couvrir t-wise est NP-complet

- se contenter de méthodes approchées

Test Combinatoire (5)

Pour aller plus loin 1 : algorithmes usuels [Aditya Mathur, chap. 4]

- covering arrays
- pour $M = 2$: procédure dédiée efficace (polynomiale)

Pour aller plus loin 2 : les DT générées par l'algorithme précédent ne sont pas équilibrées : certaines valeurs sont exercées bien plus que d'autres

- algorithmes à base de carrés latins orthogonaux pour assurer aussi l'équilibrage

Pour aller plus loin 3 : on peut vouloir intégrer certaines contraintes sur les entrées, typiquement exprimer que certaines paires de valeurs sont impossibles

Plan

- BN : test combinatoire
- BN : test partitionel
  - test aux valeurs limites
  - utilisation conjointe avec le test combinatoire
- BN : couverture fonctionnelle
- BN : test ad hoc
- Discussion

Partition des entrées : principe

Principe :

- diviser le domaine des entrées en un nombre fini de classes tel que le programme réagisse pareil (en principe) pour toutes valeurs d'une classe
- conséquence : il ne faut tester qu'une valeur par classe !
- $\Rightarrow$ permet de se ramener à un petit nomlbre de CTs

Exemple : valeur absolue : `abs : int` $\mapsto$ `int`

- 2^{32} entrées
- MAIS seulement 3 classes naturelles : $< 0, = 0, > 0$
- on teste avec un DT par classe, exemple : -5, 0, 19

Partition des entrées : principe (2)

Procédure :

1. Identifier les classes d'équivalence des entrées
   - Sur la base des conditions sur les entrées/sorties
   - En prenant des classes d'entrées valides et invalides
2. Définir des CT couvrant chaque classe

Comment faire les partitions ?

Définir un certain nombre de caractéristiques C_i représentatives des entrées du programme

Pour chaque caractéristique C_i, définir des blocs $b_{i,j} \subseteq C_i$

- (couverture) $\cup_j b_{i,j} = C_i$
- (séparation) idéalement $b_{i,j'} \cap b_{i,j} = \emptyset$

Pourquoi plusieurs caractéristiques ?

- plusieurs variables : `foo(int a, bool b)` :
  $C_1 = \{< 0, = 0, > 0\}$ et $C_2 = \{\top, \bot\}$
- caractéristiques orthogonales : `foo(list<int> l)` :
  $C_1 = \{sorted(l), \neg sorted(l)\}$ et $C_2 = \{size(l) > 10, size(l) \leq 10, \}$

Les partitions obtenues sont le produit cartésien des $b_{i,j}$

- attention à l'explosion !
- on verra une méthode moins coûteuse plus tard

Deux grands types de partition

interface-based

- basée uniquement sur les types des données d'entrée
- facile à automatiser ! (cf. exos)

functionality-based

- prend en compte les relations entre variables d'entrées
- plus pertinent
- peu automatisable

Exemple : `searchList : list<int> × int ↦ bool`

interface-based : $\{empty(l), \neg empty(l)\} \times \{< 0, = 0, > 0\}$

functionality-based : $\{empty(l), e \in l, \neg empty(l) \wedge e \notin l\}$

À propos des entrées invalides du programme

Conseil 1 : attention à en faire !

Conseil 2 : attention à ne pas trop en faire !!

Pour une fonction de calcul de valeur absolue :

- si le programme a une interface textuelle : légitime de tester les cas où l'entrée n'est pas un entier, il n'y a pas d'entrée, il y a plusieurs entiers, etc.
- si on a à faire à un module de calcul avec une interface "propre" (un unique argument entier) : on ne teste pas les valeurs invalides sur le moteur de calcul (la phase de compilation nous assure de la correction), mais sur le front-end (GUI, texte)

Conseil 3 : Ne pas mélanger les valeurs invalides !

Exemple 1 : Valeur absolue

Tester une fonction qui calcule la valeur absolue d'un entier.

type d'entrée : interface textuelle

Classes d'équivalence pour les entrées :

Condition	Classe valide	Classe invalide
nb. entrées	1	0, > 1
type entrée	int	string
valeurs valides	< 0, $>= 0$	

Données de test :
valides : -10, 100, invalides : "XYZ", rien, (10,20)

Exemple 2 : Valeur absolue, bis

Tester une fonction qui calcule la valeur absolue d'un entier.

type d'entrée : un front-end assure qu'on a une paire d'entiers

Classes d'équivalence pour les entrées :

Condition	Classe valide	Classe invalide
nb. entrées	1	
type entrée	int	
valeurs valides	< 0, $>= 0$	

Données de test :
valides : -10, 100, invalides :

Exemple 3 : Calcul somme max

Tester une fonction qui calcule la somme des `v` premiers entiers tant que cette somme reste plus petite que `maxint`. Sinon, une erreur est affichée. Si `v` est négatif, la valeur absolue est considérée.

type d'entrée : un front-end assure que les entrées sont bien une paire d'entiers

Classes d'équivalence pour les entrées :

Condition	Classe valide	Classe invalide
nb. entrées	2	
type entrée	int int	
valeurs valides `v`	< 0, ≥ 0	
valeurs valides `maxint`	$>$ somme , $\leq$ somme	

Données de test et oracle :

maxint	v	return
100	10	55
100	-10	55
10	10	error
10	-10	error

Quand utiliser le test de partitions ?

Pour des spécifications orientés données

- y compris interfaces de fonctions

Pour des interfaces de fonctions / méthodes

- méthode fortement automatisable !

Analyse des valeurs limites

Le test des valeurs limites est une tactique pour améliorer l'efficacité des DT produites par d'autres familles.

- s'intègre très naturellement au test partitionnel

Idée : les erreurs se nichent dans les cas limites, donc tester aussi les valeurs aux limites des domaines ou des classes d'équivalence.

- test partitionnel en plus agressif
- plus de blocs, donc plus de DT donc plus cher

Stratégie de test :

- Tester les bornes des classes d'équivalence, et juste à côté des bornes
- Tester les bornes des entrées et des sorties

Analyse des valeurs limites (2)

Exemples :

- soit N le plus petit/grand entier admissible : tester $N-1$, N, $N+1$
- ensemble vide, ensemble à un élément
- fichier vide, fichier de taille maximale, fichier juste trop gros
- string avec une requête sql intégrée
- ...

Exemple : Valeur absolue, ter

Tester une fonction qui calcule la valeur absolue d'un entier.

type d'entrée : un front-end assure qu'on a une paire d'entiers

Classes d'équivalence pour les entrées :

Condition	Classe valide	Classe invalide
nb. entrées	1	
type entrée	int	
valeurs valides	< 0, $>= 0$	
limites	0, -2^{31}	

Données de test :
valides : -10, 100, 0, -2^{31} invalides :

Caractéristiques multiples

Si on a plusieurs entrées :

dans le cas où il y a trop de $b_{i,j}$, le nombre de partitions $\Pi b_{i,j}$ explose et la technique devient inutilisable

Comment faire : l'approche combinatoire peut être appliquée aux $b_{i,j}$

- on ne cherche plus à couvrir tout $\Pi b_{i,j}$
- mais par exemple toutes les paires $(b_{i,j}, b_{i',j'})$
- on retrouve l'approche pair-wise

Notion de cas de base

La notion de cas de base permet encore de réduire la combinatoire des tests de manière intéressante

Pour le moment on a vu comme types de blocs :

- valides (dont limites)
- invalides

Et un conseil : ne pas combiner les cas invalides entre eux

On ajoute la notion de cas de base (BC)

- forcément un cas valide
- cas le plus représentatif (donc pas un cas limite)

- exemple : `int` : BC $= \{> 0\}$

Notion de cas de base

Utilisation : la notion de BC permet de casser la combinatoire en variant le critère de couverture selon le type de caractéristique (invalide, valide - BC, valide - limite)

Couverture des blocs invalides :

- pour chaque bloc invalide $b^{\star}_{i,j}$ de C_i, faire un CT de la forme $BC_1 \times BC_2 \times \ldots \times BC_{i-1} \times b^{\star}_{i,j} \times BC_{i+1} \times \ldots \times BC_n$
- couverture 1-wise sur les blocs invalides, en utilisant systématiquement les cas de base
- nb tests = nb blocs invalides

Couverture des blocs valides :

- idéalement 2-wise
- mais si encore trop de CTs / paires : distinguer pour chaque C_i plusieurs BC, des cas invalides et des cas limites. Puis couvrir les cas limites combinés avec des BC (cf cas invalides), et couvrir les BC en 2-wise

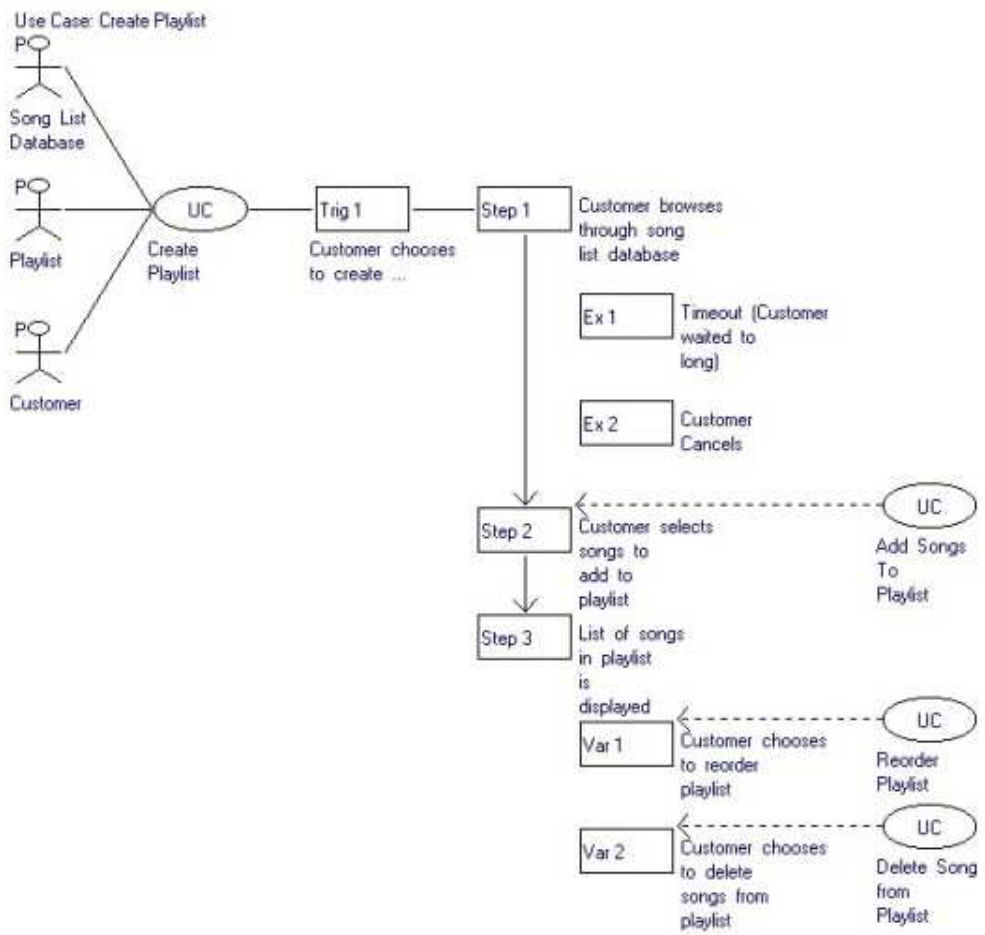
Plan

- BN : test combinatoire
- BN : test des partitions
- BN : couverture fonctionnelle
- BN : test ad hoc
- Discussion

Couverture Fonctionnelle

Descriptions du comportement du programme, sous formes plus ou moins graphiques, plus ou moins formelles

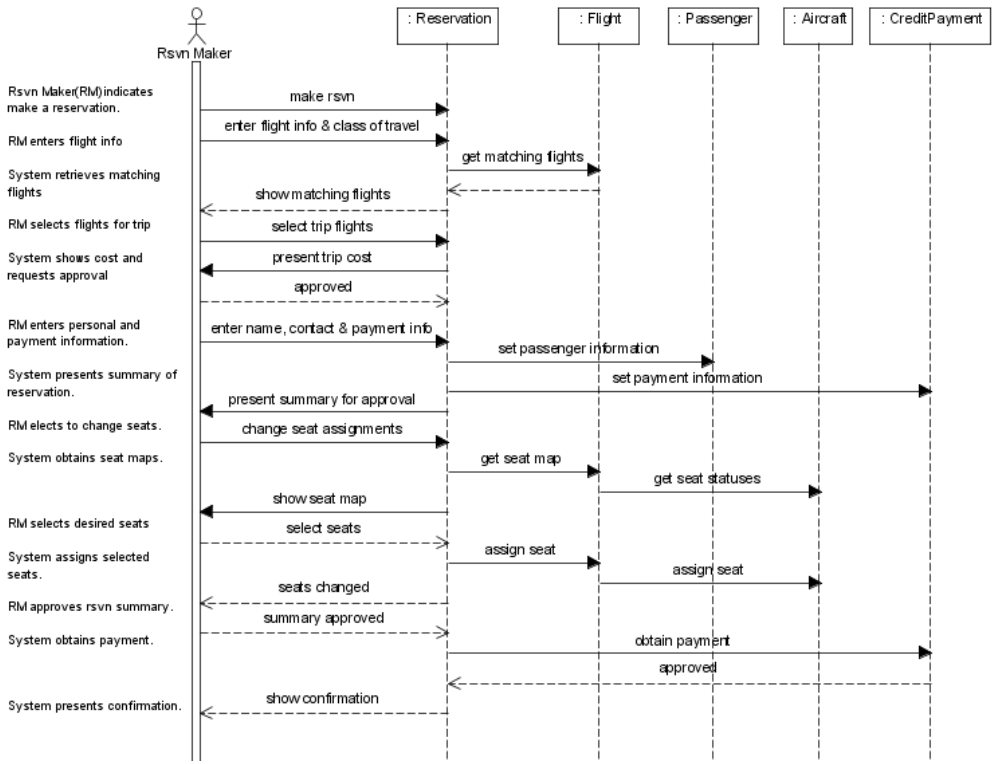
- use-cases, scénarios, message sequence charts, diagrammes d'activité, etc.



Couverture Fonctionnelle

Descriptions du comportement du programme, sous formes plus ou moins graphiques, plus ou moins formelles

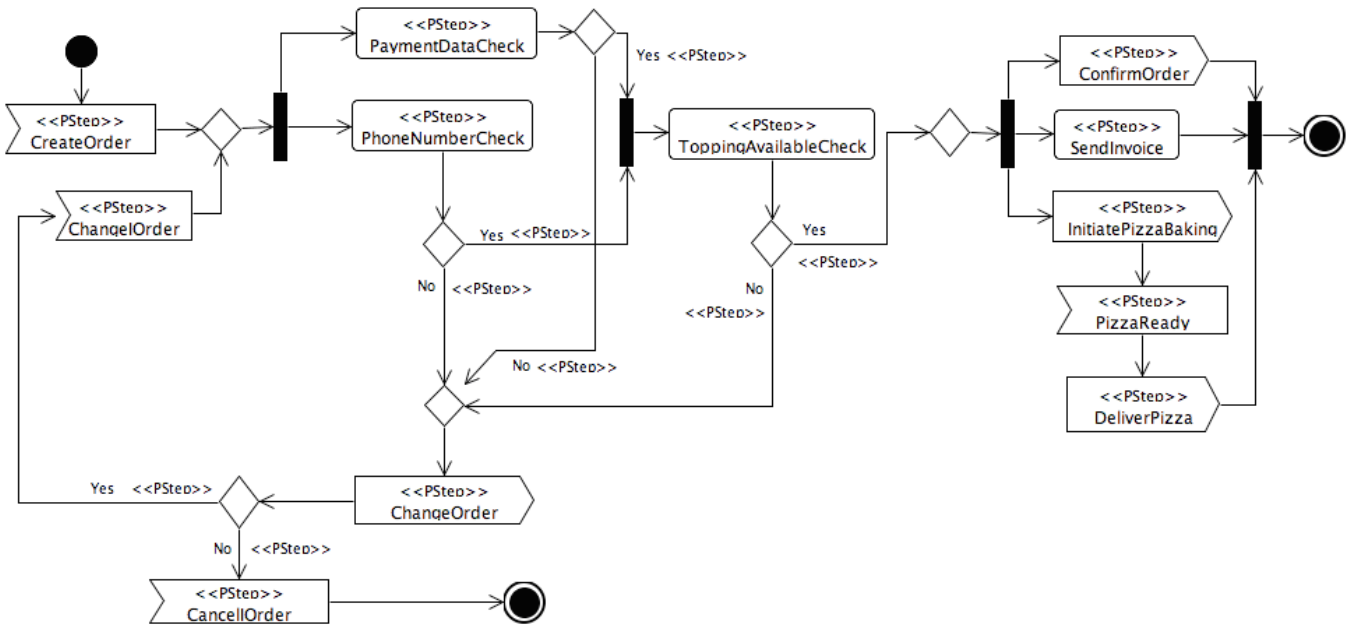
- use-cases, scénarios, message sequence charts, diagrammes d'activité, etc.



Couverture Fonctionnelle

Descriptions du comportement du programme, sous formes plus ou moins graphiques, plus ou moins formelles

- use-cases, scénarios, message sequence charts, diagrammes d'activité, etc.



Couverture Fonctionnelle (2)

On peut toujours définir une notion de couverture des spécifications (besoins, modèles, etc.)

Dans ce cas, on choisit les DT pour couvrir les éléments de la spécification

- noeuds, transitions, etc.
- réutilisation des concepts de test boite blanche

Remarques :

- la couverture des use-cases est naturelle
- le véritable intérêt = quand il y a un modèle formel (Model-Based Testing)
- dans ce cas, on peut utiliser techniques de couverture (BB) sur le modèle : couverture fonctionnelle = couverture structurelle du modèle
- utiliser un graphe causes-effets revient à se créer un modèle et le couvrir

Plan

- BN : test combinatoire
- BN : test des partitions
- BN : couverture fonctionnelle
- BN : test ad hoc
- Discussion

Test ad hoc

Les méthodes précédentes ne permettent pas de trouver à coup sûr les erreurs

Le test laisse de la place à la créativité

- intuition
- expérience

Idée : essayer de placer le programme dans des situations à risque

- dépend du type de programme, type de langage, etc.

Test ad hoc (2)

Exemples pour un programme de tri de listes en C

- liste dont tous les éléments sont identiques
- liste déjà triée
- liste circulaire

Exemples pour un programme de fusion de listes en C

- une des listes est circulaire
- les deux listes ne sont pas disjointes

Plan

- BN : test combinatoire
- BN : test des partitions
- BN : test ad hoc
- BN : couverture fonctionnelle
- Discussion

Comment combiner les différentes stratégies ?

Principe pour ne pas générer trop de tests : incrémental et prioritisation

- ordonner les méthodes de sélection selon #tests générés
- prendre celle qui en génère le moins habituellement
- choisir les DT, lancer les tests
- prendre la prochaine famille de tests, regarder "couverture" obtenue et ne rajouter que CT manquants
- itérer

Ordre typique :

- couverture des use-cases / domaines (functionality-based)
- couverture du modèle formel / domaines (interface based)
- couverture du code [cf cours suivant]

Conseils

Combiner les critères, du plus simple au plus difficile

Penser à la distinction cas de base - valeur limite - valeur invalide

- penser au test de robustesse
- ne pas faire que du test de robustesse

Inclure un peu d'aléatoire (par ex, pour choisir $v \in b_{i,j}$)

Pas de critère parfait : adaptation et innovation

Automatisation ?

Test pairwise : facile (outils commerciaux)

Test aléatoire : facile a priori

- problèmes des préconditions et de l'oracle
- outils ?

interface-based testing pour du code : facile a priori

- problèmes des préconditions et de l'oracle
- outils ?

Couverture fonctionnelle : travaux autour du MBT (cf après)

- méthodes avancées (état de l'art)
- quelques outils +/- commerciaux (GaTeL, BZTools)
- nécessitent un utilisateur très expérimenté

Couverture structurelle : (cf après)

- très gros progrès récents, nombreux outils académiques
- Pex pour C# (Microsoft)
- faciles à utiliser mais problème de l'oracle !!!

Critères et applications

Test combinatoire :

- GUI, configuration
- +aide à méthodes des partitions

Méthodes des partitions : domain-based / interface-based

- systèmes orientés données
- fonctions / méthodes d'un programme

Couverture de modèles (de type système de transitions)

- systèmes réactifs / event-oriented

A propos de la classification

La classification usuelle (BB,BN) n'est pas toujours lisible

- la couverture de modèle (BN) est plus proche la couverture de code (BB) que de l'approche par partitions (BN)
- les mutations (cf plus tard) peuvent s'appliquer aussi bien au code qu'aux spécifications
- etc.

Ammann et Offutt proposent une nouvelle classification, qui peut être déclinée à chaque niveau de développement :

- critères de graphe (cfg / cg / ddg du programme, système de transitions)
- critères de partition du domaine d'entrée (interfaces des fonctions ou des modèles)
- critères logiques
- critères syntaxiques / mutation