Projet programmation fonctionnelle : Tests

Julien Narboux

September 26, 2024

Comparaison des méthodes de vérification

Tests:
- Mise en oeuvre **relativement** facile.
- Nécessaire : on trouve des erreurs dans le système complet : spécification et implantation.

"Beware of bugs in the above code; I have only proved it correct, not tried it." *Donald Knuth*

- Pas suffisant : exhaustivité impossible

Preuves:
- Exhaustif.
- Difficile à mettre en œuvre, nécessite du personnel formé.

Analyse statique par interprétation abstraite :
- Exhaustif pour certaines classes de problèmes (safety).
- N'est pas applicable à tous les programmes.

Vérification et validation :

- environ 30% du développement d'un logiciel standard
- plus de 50% du développement d'un logiciel critique

Validation ?

Les tests permettent de détecter des erreurs, pas de vérifier qu'un programme est correct.

> *"Testing can only reveal the presence of errors but never their absence." - E. W. Dijkstra*

Motivations

1. Trouver des bugs
   - plus les bugs sont trouvés tôt moins ils coûtent cher
   - par les développeurs
2. Validation: convaincre un tiers de la qualité du code.
   - Client
   - Hiérarchie
   - Organisme de certification

$\rightarrow$ Problématique de l'évaluation d'un jeu de tests.

Bibliographie

Sur le test en général:

- Myers G.J., *The Art of Software Testing*, John Wiley Ed, New York, 1979
- Introduction au test logiciel, Philippe Herrmann: `http://sebastien.bardin.free.fr/poly-herrmann.pdf`
- Le cours de Sébastien Bardin: `http://sebastien.bardin.free.fr/cours1.pdf`

Sur le test en OCaml:

- Real world OCaml: `https://dev.realworldocaml.org/testing.html`

Tests statiques/dynamiques

statique sans exécuter le programme sur des données réelles : analyse symbolique ou relecture de code.

dynamique exécuter le programme sur des exemples.

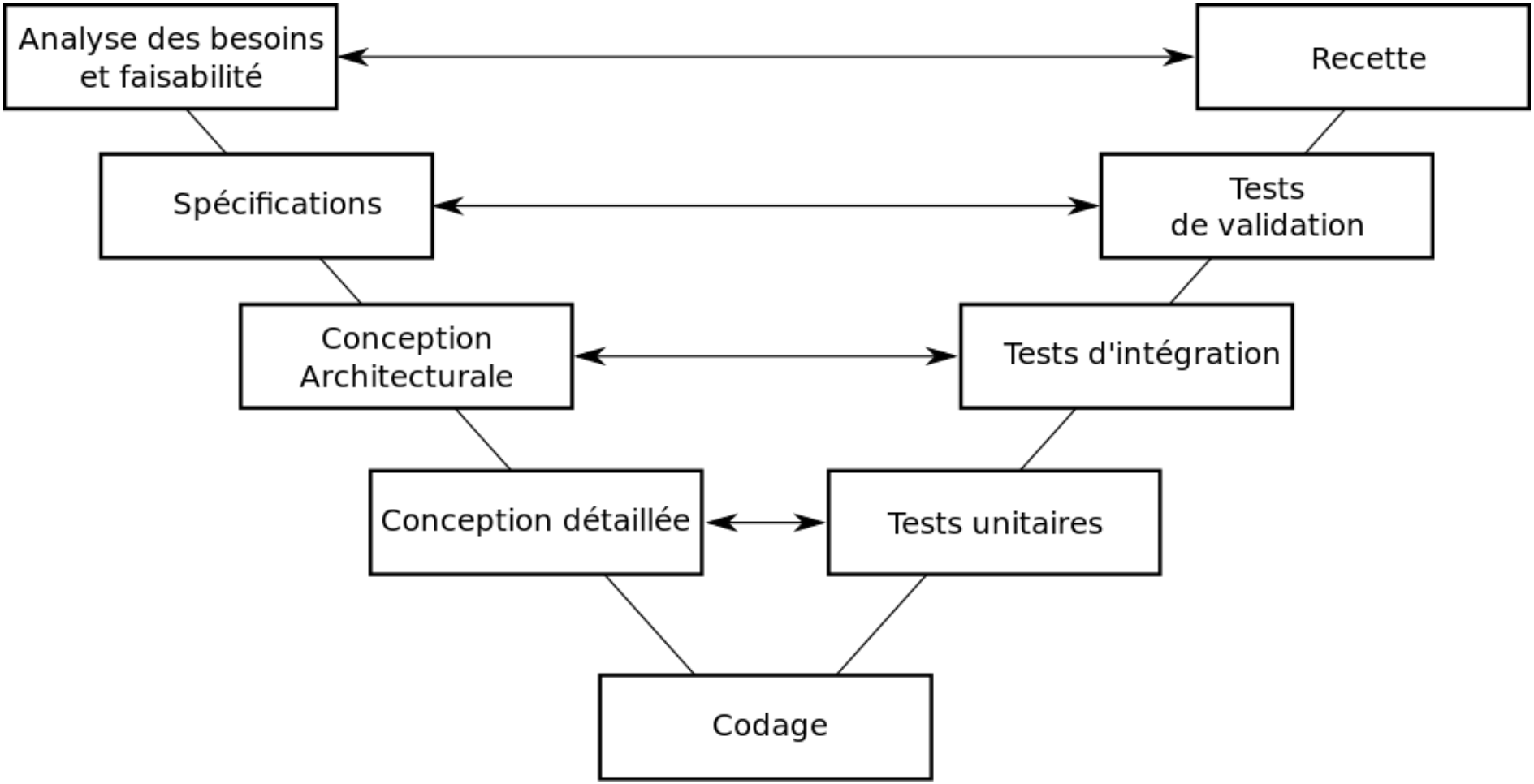
Tests dynamiques

- code écrit à la main (tests unitaires)
- code généré automatiquement
- action manuelle
- action automatisée

Tests: ce qui est testé

- Test de conformité
- Test de robustesse
- Test de sécurité
- Test de performance

Cycle en V



Licence CC, auteur:Christophe Moustier

Tests: classement par niveau de détail

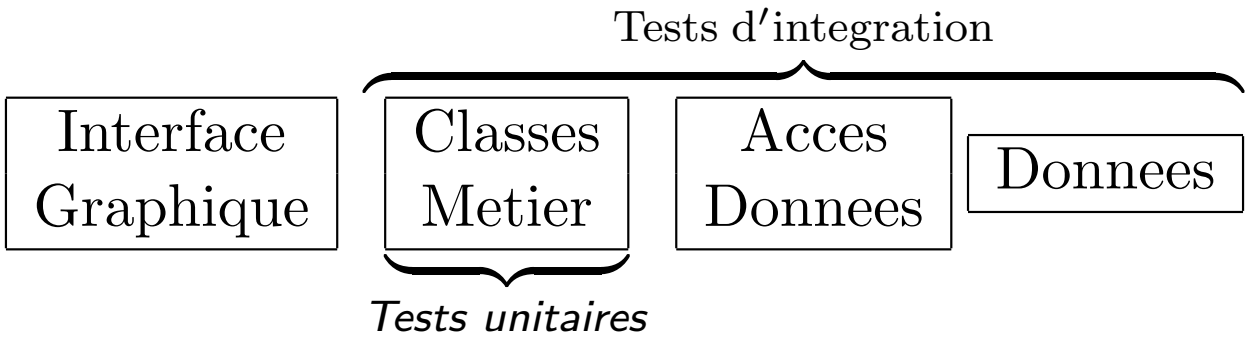
Tests unitaires Test des unités de programme de façon isolée : uniquement correction fonctionnelle.

Tests d'intégration Test de la composition des modules via leur interface : principalement correction fonctionnelle.

Tests de validation Test du produit fini par rapport au cahier des charges : conformité, robustesse, sécurité, performance.

Recette Test du produit fini par le client.

Tests d'intégration vs tests unitaires



Tests : boîte noire/vs boîte blanche

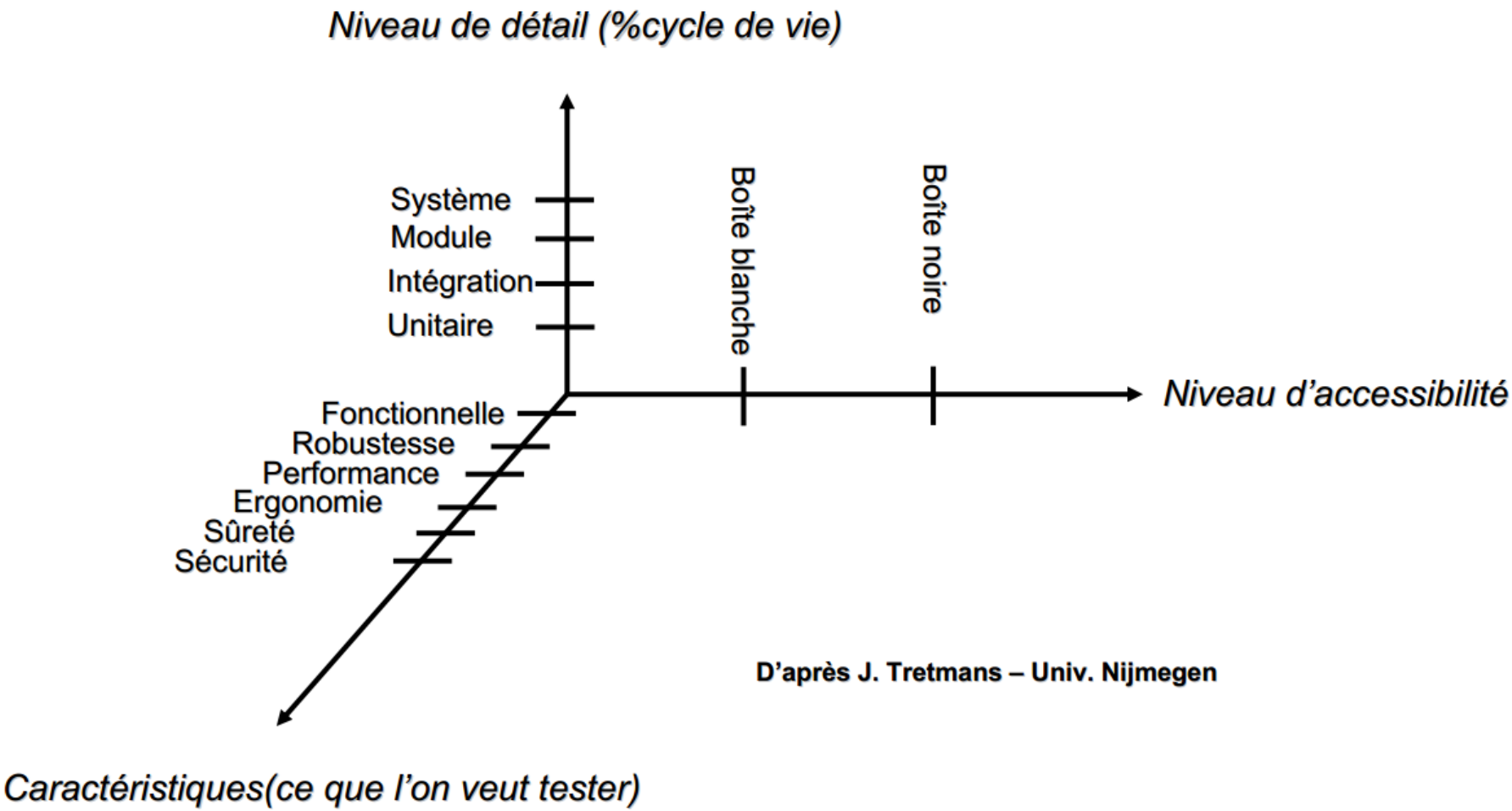
Boîte noire :

- Test de l'extérieur vis-à-vis des spécifications (formelles ou informelles) sans connaissance de l'implantation.
- Technique applicable à tous les niveaux du cycle en V.
- Écriture des tests possible avant le code.

Boîte blanche :

- Tests en ayant connaissance du code.
- Si on change le code il faut mettre à jour les tests.

Tests


Niveau de détail (%cycle de vie)
Système
Module
Intégration
Unitaire
Boîte blanche
Boîte noire
Niveau d'accessibilité
Fonctionnelle
Robustesse
Performance
Ergonomie
Sûreté
Sécurité
D'après J. Tretmans – Univ. Nijmegen
Caractéristiques(ce que l'on veut tester)

Processus de test

Les tests peuvent être écrits:

- avant, (développement guidé par le test, *test driven development*)
- pendant ou
- après

le code.

Les tests peuvent être réalisés:

- par le développeur (tests unitaires)
- par une équipe dédiée

Oracle

Le test est réalisé vis-à-vis d'un comportement attendu. Celui-ci peut être défini par :

- des spécifications
- une norme
- des contrats (préconditions, postconditions et invariants)
- des versions précédentes

Tests unitaires

Principe:

Test des unités de programme de façon isolée, à l'échelle de la classe ou de la fonction. Pour chaque classe on crée une classe de test avec au moins une méthode de test par méthode.

En pratique:

1. Des *annotations* pour préciser quelles sont les fonctions de test et d'initialisation des tests (quand le language le permet).
2. Des *assertions* pour tester des propriétés : un prédicat + un message d'erreur.
3. Des *lanceurs de tests* : une boucle un peu évoluée.

Exemples

Java	junit
.Net	nunit, xunit
haskell	hunit
ocaml	ounit, `ppx_inline_test`, `ppx_assert`
	...[a]

[a] `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`

Les objets factices

Les *Mocks* sont des objets *factices* qui imitent le comportement d'objets réels. Les *Mocks* ne définissent la sortie que de certaines méthodes et pour une quantité limitée d'entrées.
Ils sont nécessaires quand il faut remplacer :

- un comportement non déterministe (une méthode qui donne l'heure pour tester un réveil par exemple)
- un événement rare (une déconnexion réseau par exemple)
- une classe qui n'est pas encore implantée
- une classe qui produit des temps de calcul trop longs
- une classe qui utilise des ressources externes que l'on ne souhaite pas tester[1] (accès à une BDD, connexion réseau, envoie de mails, . . .)

Les *objets factices* peuvent être écrits :

- à la main ou
- automatiquement en utilisant la réflexivité pour certains langages (EasyMock, Mockito, . . .)

[1] Ça sera l'objet des tests d'intégration

Doublures

On peut distinguer plusieurs types de doublures :

bouchons Une doublure qui sert simplement à remplacer un autre code pour que ça compile.

mock Une doublure qui vérifie l'interaction avec l'objet.

espions Une doublure qui enregistre les traitements qui lui sont fait.

Bouchons, en OCaml ?

En OCaml, on peut utiliser les foncteurs pour écrire des bouchons. On crée une implémentation simplifiée alternative.

Best practice tests unitaires

- Écrire un test par bug trouvé (tests de régression).
- Les tests ne doivent pas faire d'hypothèses sur le reste du monde.
- Les tests ne doivent comporter qu'une seule assertion (sinon découper en plusieurs tests).
- Ne pas mélanger le code de test avec le reste du code.
- Les tests doivent être indépendants les uns des autres :
  - ne pas appeler un test dans un autre
  - les tests doivent pouvoir être appelés dans n'importe quel ordre
- Attention à ne pas mettre trop de code dans les @Before @After pour garder les tests lisibles. Seules les données utilisées dans *tous* les tests de la classe doivent être initialisés dans la méthode @Before.

Analyse partitionnelle

Idée:

Partitionner les domaines d'entrée en un nombre fini de classes d'équivalences correspondant à des comportements "**identiques**" du programme.
Réaliser un test par classe d'équivalence.

Exemple: valeur absolue

- abs : int $\rightarrow$ int
- 2^{32} cas
- trois classes "naturelles": $x < 0$, $x = 0$, $x > 0$

Test aux limites

Après une **analyse partitionnelle**, introduire des tests se trouvant aux **frontières** des classes d'équivalence.

- fichier vide, fichier de plus de 4Go, fichier sans droit d'écriture
- ensemble vide, ensemble à un élément, ensemble très gros
- chaîne vide, chaîne comportant des caractères spéciaux
- entier maximum/minimum, zéro
- float proche de zéro, très grand
- premier élément, dernier élément d'un tableau
- ...

Tests combinatoires

Sélection d'un petit nombre de configurations de test parmi un ensemble de configurations dont la combinatoire explose.

- Pour les variables ayant peu de valeurs possibles.
- Utile par exemple pour les tests d'intégration ou de configurations.

Contre-exemple

Addition de deux `int`:

$$2^{32} * 2^{32} = 2^{64} \text{ cas possibles.}$$

Exemple

OS	Connexion	CPU	Java
Linux	Ethernet	PowerPC	Java 1.4
Windows	Wifi	ARM	Java 1.5
MacOS	Bluetooth	Intel	Java 1.6

$$3^4 = 81 \text{ cas possibles.}$$

Approche pair-wise

Idée sous-jacente

- La majorité des erreurs sont détectées par des combinaisons de 2 valeurs de variable.
- Un seul test couvre plusieurs paires
- Le nombres de paires est bien plus petit que le nombre de combinaisons.

Exemple

OS	Connexion	CPU	Java
Linux	Ethernet	PowerPC	1.4
Linux	Wifi	ARM	1.5
Linux	Bluetooth	Intel	1.6
Windows	Ethernet	Intel	1.5
Windows	Wifi	PowerPC	1.6
Windows	Bluetooth	ARM	1.4
MacOS	Ethernet	ARM	1.6
MacOS	Wifi	Intel	1.4
MacOS	Bluetooth	PowerPC	1.5

9 cas de test.

Génération

Le problème de la génération d'un ensemble de données de tests de cardinal minimal est NP complet. Mais des outils existent : `http://www.pairwise.org/tools.asp`

N-wise / partionnelle

Lorsque l'on a plusieurs critères l'approche *pairwise* peut être combinée à l'approche partionnelle.

Exemple

Une fonction qui prend en entrée un `int` représentant un numéro de mois et un `int` représentant le numéro de jour et renvoie le jour suivant.

- partition des entrées valides et invalides :
  $m \leq 0$, $1 \leq m \leq 12$ $m > 12$.
  $j \leq 0$, $1 \leq j \leq 28$, $j = 29$, $j = 30$, $j = 31$, $j > 31$.
- mois de 28, 30 ou 31 jours.

Génération aléatoire

- nécessite un oracle qui fonctionne dans tous les cas (spécification)
- niveau de couverture souvent faible
- difficulté de tester des cas particuliers (ex: trois points alignés)
- mais c'est facile à mettre en œuvre

QuickCheck (Haskell puis porté à d'autres langages)

- La bibliothèque fournit des générateurs aléatoires pour les types de base : int, string, float, list, array, ...
- On définit des générateurs.
- On teste des prédicats capturant des propriétés de la fonction. Par exemple:
  - encrypt(decrypt(x))= x
  - reverse(reverse(l))=l
  - ...

QuickCheck en OCaml

https://github.com/c-cube/qcheck

```
Test.make float (fun f -> floor f <= f)

Test.make ~name:"rev twice"
(list int)
(fun xs -> List.rev (List.rev xs) = xs)
```

Gestion des pré-conditions

```
Test.make (list int) (fun l ->
  assume (l <> []);
  List.hd l :: List.tl l = l)
```

Composition de générateurs

- `pair g1 g2` pour construire un générateur de paires grâce à deux générateurs
- `oneof [g1; ... ; gn]` construit un générateur qui choisit parmi des générateurs
- `map f g` transforme un générateur `g` de type `'a` en un générateur de type `'b` en utilisant une fonction de type `f: 'a -> 'b`

Exemple :

```
let even_gen : int Gen.t = Gen.map (fun n -> n * 2) Gen.int
```

Construire ses propres générateurs

- `'a arbitrary` couvre la génération et l'affichage
-
```
type 'a arbitrary = {
gen : 'a Gen.t; (* "pure" generator
*)
print : ('a -> string) option;
```
- On peut utiliser `make ~print:p : 'a Gen.t -> 'a arbitrary` pour construire un générateur (arbitrary) à partir d'un pur générateur et d'une fonction d'affichage.

Exemple générateur d'arbres

Considérons les expressions arithmétiques :

```
type aexp =
| Const of int
| Plus of aexp * aexp
| Times of aexp * aexp
```

Un générateur de feuilles:

```
let leafgen =  Gen.map (fun i -> Const i) Gen.int
```

Exemple générateur d'arbres

```
let rec mygen n = match n with
  | 0 -> leafgen
  | n ->
      Gen.oneof
        [leafgen;
         Gen.map2 (fun l r -> Plus(l,r))
                  (mygen (n/2)) (mygen (n/2));
         Gen.map2 (fun l r -> Times(l,r))
                  (mygen (n/2)) (mygen (n/2)) ]

let aexp_arb = make ~print:exp_to_string (mygen 8)
```

Extensions

- On peut ajouter des fonctions pour minimiser les contre-exemples.

Fuzzing

- Génération d'entrées aléatoires pour essayer de faire planter le programme.
- Génération à partir de rien ou bien par génération par mutation d'entrées valides
- Smart Fuzzing: en connaissance de la structure de l'entrée.
- Dumb Fuzzing: sans connaitre la structure de l'entrée.
- En boite grise: plus efficace si on instrumente le code binaire.
- Nombreux succès en pratique, par exemple afl: Dumb Fuzzer en boite grise à base de mutations

Fuzzing en OCaml avec afl

```
let _ =
  let s = read_line () in
  match Array.to_list
     (Array.init (String.length s) (String.get s)) with
    ['s'; 'e'; 'c'; 'r'; 'e'; 't'; ' '; 'c'; 'o'; 'd'; 'e']
        failwith "uh oh"
  | _ -> ()
```

```
ocamlopt -afl-instrument readline.ml -o readline
```

Source : https://ocaml.org/manual/5.2/afl-fuzz.html

Génération à partir d'un modèle

- Les exigences sont formalisées sous forme d'un modèle.
- Le modèle est une représentation abstraite des comportements d'un système.
- Le modèle est plus ou moins formel : cas d'utilisation, diagrammes d'activités, automates, ...

Méthodes de test de couverture en boîte blanche

Couverture structurelle

- par couverture du graphe de flot de contrôle (CFG)
- ou par couverture du graphe de flot de données (DFG)

Test de couverture par mutation

Sélection des cas de tests en changeant le programme.

Quelques critères de couverture:

Tous les noeuds le plus faible

Tous les arcs / décisions (D)

Toutes les conditions simples (C) peut ne pas couvrir toutes les décisions

Toutes les conditions et toutes les décisions (DC)

Toutes les combinaisons de conditions (MC) explosion du nombre de cas, revient à tester exhaustivement dans le domaine des booléens.

Tous les chemins de longeur k

Tous les chemins le plus fort, impossible à réaliser

Notion de hiérarchie de critères

On dit qu'un critère C1 est plus fort qu'un critère C2 si:
pour tout programme P et pour toute suite de tests TS, si TS couvre C1 pour P alors TS couvre C2 pour P.

Problème de l'accessibilité

- Attention tous les chemins ne sont pas atteignables.
- Toutes les instructions ne sont pas forcement atteignables (code mort, code de débuggage).
- Savoir si un chemin est atteignable est indécidable.
- Le taux de couverture est donc réduit.

Exercice

Limite du critère instruction

Trouver un programme pour lequel le critère instruction n'est pas suffisant.

Exercice

Limite du critère instruction

Trouver un programme pour lequel le critère instruction n'est pas suffisant.

Solution

```
if x != 0 then x := 1;
y := a/x
```

Exemple

```
if ((A && B) || C) then 1
  else if D then 3 else 5
```

Pour couvrir:

Exemple

```
if ((A && B) || C) then 1
  else if D then 3 else 5
```

Pour couvrir:

D `(A && B) || C=true`, `(A && B) || C=false`
`D=true`, `D=false`

Exemple

```
if ((A && B) || C) then 1
  else if D then 3 else 5
```

Pour couvrir:

D `(A && B) || C=true, (A && B) || C=false`
`D=true, D=false`

C `A=true, A=false, B=true, B=false, C=true, C=false`
`D=true, D=false`

Exemple

```
if ((A && B) || C) then 1
  else if D then 3 else 5
```

Pour couvrir:

D `(A && B) || C=true, (A && B) || C=false`
`D=true, D=false`

C `A=true, A=false, B=true, B=false, C=true, C=false`
`D=true, D=false`

MC
```
A=true,  B=true,  C=true
A=true,  B=true,  C=false
A=true,  B=false, C=true
A=true,  B=false, C=false
A=false, B=true,  C=true
A=false, B=true,  C=false
A=false, B=false, C=true
A=false, B=false, C=false
D=true,  D=false
```

Problème

Mais il existe dans le code d'un avion une décision avec 76 conditions (un if avec un expression booléennes comportant 76 atomes)!

```
  Bv or (Ev /= El) or Bv2 or Bv3 or Bv4 or Bv5 or Bv6 or Bv7 or Bv8 or
Bv9 or Bv10 or Bv11 or Bv12 or Bv13 or Bv14 or Bv15 or Bv16 or
Bv17 or Bv18 or Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or
Bv25 or Bv26 or Bv27 or Bv28 or Bv29 or Bv30 or Bv31 or Bv32 or
Bv33 or Bv34 or Bv35 or Bv36 or Bv37 or Bv38 or Bv39 or Bv40 or
Bv41 or Bv42 or Bv43 or Bv44 or Bv45 or Bv46 or Bv47 or Bv48 or
Bv49 or Bv50 or Bv51 or (Ev2 = El2) or
((Ev3 = El2) and (Sav /= Sac)) or Bv52 or Bv53 or Bv54 or Bv55 or
Bv56 or Bv57 or Bv58 or Bv59 or Bv60 or Bv61 or Bv62 or Bv63 or
Bv64 or Bv65 or Ev4 /= El3 or Ev5 = El4 or Ev6 = El4 or Ev7 = El4 or
Ev8 = El4 or Ev9 = El4 or Ev10 = El4
```

Source:`http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf`

DO-178B

La norme DO-178B est utilisée en aéronautique.

- **Level A** catastrophic failure condition “prevent continued safe flight and landing“
- **Level B** hazardous / severe-major failure condition “serious or potentially fatal injuries to a small number of . . . occupants”
- **Level C** major failure condition “discomfort to occupants, possibly including injuries”
- **Level D** minor failure condition “some inconvenience to occupants”
- **Level E** no effect on aircraft operational capability or pilot workload

DO-178B : Couverture de code requise

	Level A	Level B	Level C	Level D
Statement Coverage	✓	✓	✓	
Decision Coverage	✓	✓		
Modified Condition / Decision Coverage	✓			

MCDC

DO-178B/ED-12B:

Condition A Boolean expression containing no Boolean operators.

Decision A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.

Decision Coverage Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once.

Modified Condition/Decision Coverage Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition

Problème d'interprétation

There seems to be a variety of opinions about the definition of decision and coverage requirements.

```
if ((A or B) and (C or D))
  then toto();
```

vs

```
E:= (A or B) and (C or D);
if (E) then toto();
```

The certification authorities recommend the "literal" definition of decision for DC and MC/DC. The logic and control structure in Levels A and B software must be thoroughly exercised, whether it occurs at a branch point or not.

Voir: `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf`

Problème: cette définition n'est pas invariante vis à vis d'une transformation du programme:

```
A:= B or C;
E := A and D;
```

est équivalent logiquement à

```
E:= (B or C) and D;
```

mais les tests à faire pour une couverture MC/DC ne sont pas les mêmes !

http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf

Couverture structurelle vs tests en boîte blanche

Attention!

- Il ne faut pas confondre la **vérification d'une couverture** des tests avec la **génération** des tests.
- Les tests construits à partir du code ne peuvent pas déceler que des spécifications n'ont pas été implantées.
- La norme DO-178B exigent que les tests sont construits à partir des **exigences**.
- La méthode de couverture est utilisée uniquement pour **vérifier** que l'on a réalisé suffisamment de tests.

Outils

Outils pour évaluer les couvertures

Python Gcovr `https://gcovr.com/en/stable/`

Java Cobertura: `http://cobertura.github.io/cobertura/`

OCaml Bisect: `https://github.com/aantron/bisect_ppx`

...

Outils de génération de tests

- Microsoft PEX
  `http://research.microsoft.com/en-us/projects/pex/`
- Microsoft SAGE

Génération de tests en deux mots

1. On choisit un chemin du CFG.
2. On génère un prédicat de chemin P par execution symbolique.
3. On trouve une solution de P.

Remarque

Le test à besoin de la preuve !

Test des tests par mutation

Idée

On crée des erreurs plausibles dans le programme afin de vérifier qu'il y a assez de tests pour capturer ces erreurs.

Remarque

Les mutations doivent préserver la syntaxe et le typage.

Exemples de mutations

- Effacer une instruction.
- Remplacer une expression booléenne par `true` ou `false`.
- Remplacer une opération arithmétique par une autre, par exemple '+' par '*'.
- Remplacer 'n+1' par 'n' et vice-versa.
- Remplacer une variable par une autre.
- Remplacer une variable par une constante.
- Remplacer une constante par une autre.
- Remplacer un opérateur booléen par un autre : par exemple 'et' par 'ou', $>$ par $<$.

Les mutants

- En pratique assez peu utilisé.
- Implique facilement d'autres critères de couverture.

En OCaml

https://github.com/jmid/mutaml