Automatisation du Test Logiciel

Objectifs du cours

Compétences à aquérir

- connaître les principes généraux du test logiciel
- comprendre l'état de la technique en automatisation du test logiciel

Pour quelles industries ?

- bagage général de l'honnête informaticien [Agile Programming, etc.]
- systèmes critiques
- systèmes "de qualité" (sécurité, etc.)

Plan Général

- Introduction au test logiciel
- Exécution symbolique pour l'automatisation du test
- Exécution symbolique et critères de test avancés

Motivations

Coût des bugs

- Coûts économique : 64 milliards $/an rien qu'aux US (2002)
- Coûts humains, environnementaux, etc.

Nécessité d'assurer la qualité des logiciels

Domains critiques

- atteindre le (très haut) niveau de qualité imposée par les lois/normes/assurances/... (ex : DO-178B pour aviation)

Autres domaines

- atteindre le rapport qualité/prix jugé optimal (c.f. attentes du client)

Motivations (3)

- La vérification est une part cruciale du développement
- Le test est de loin la méthode la plus utilisée
- Les méthodes manuelles de test passent très mal à l'échelle en terme de taille de code / niveau d'exigence
- fort besoin d'automatisation

Définition du test

Le test est une méthode dynamique visant à trouver des bugs

Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts

- G. J. Myers (The Art of Software Testing, 1979)

Deux aspects du test

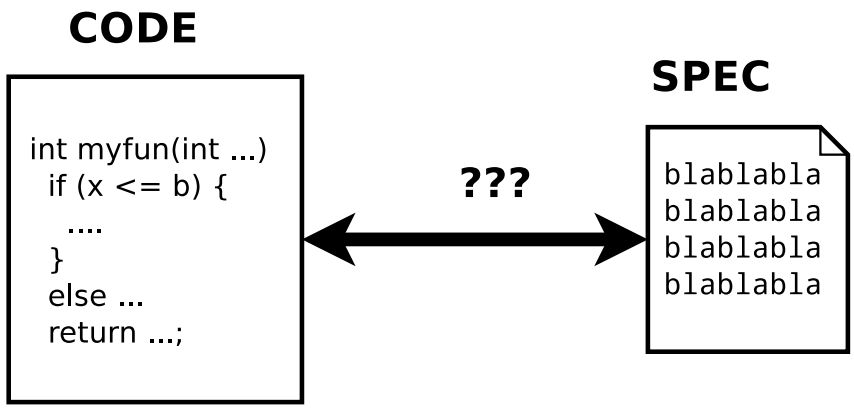
1- Construire la qualité du produit

- lors de la phase de conception / codage
- en partie par les développeurs (tests unitaires)
- but = trouver rapidement le plus de bugs possibles (avant la commercialisation)
  - test réussi = un test qui trouve un bug

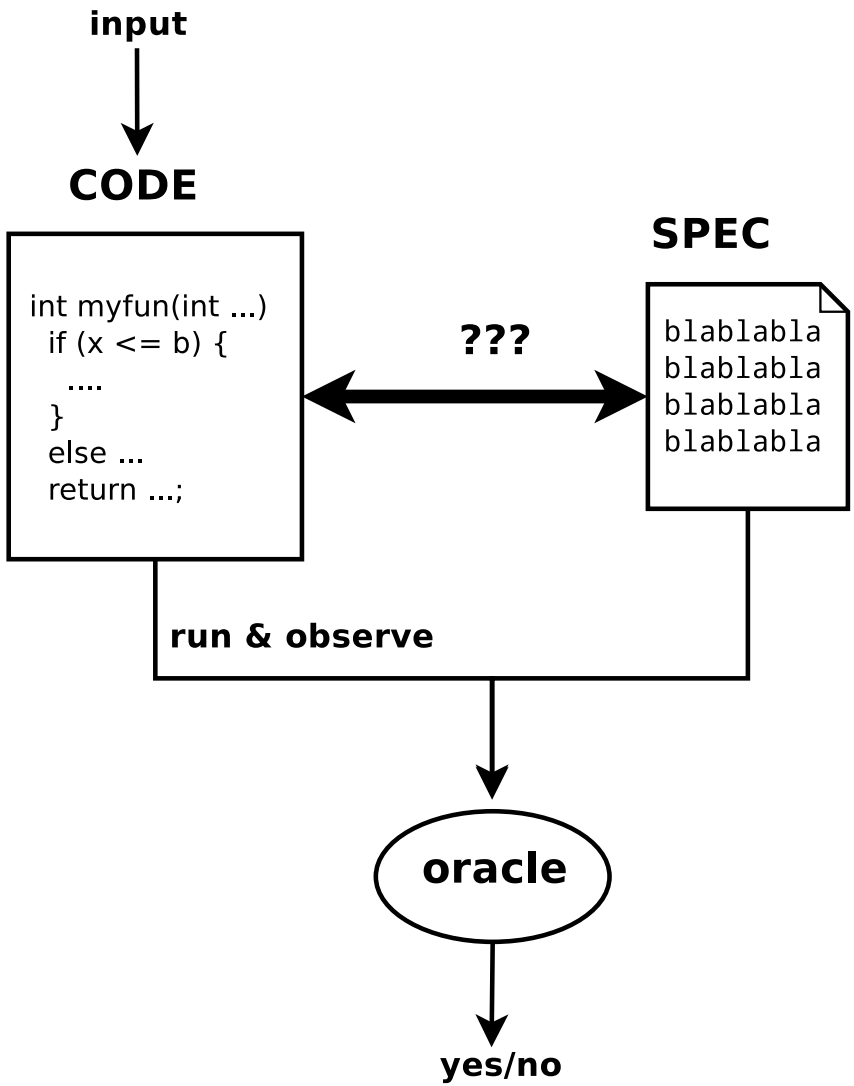
2- Démontrer la qualité du produit à un tiers

- une fois le produit terminé
- idéalement : par une équipe dédiée
- but = convaincre (organismes de certification, hiérarchie, client)
  - test réussi = un test qui passe sans problème
  - + tests jugés représentatifs
    (systèmes critiques : audit du jeu de tests)

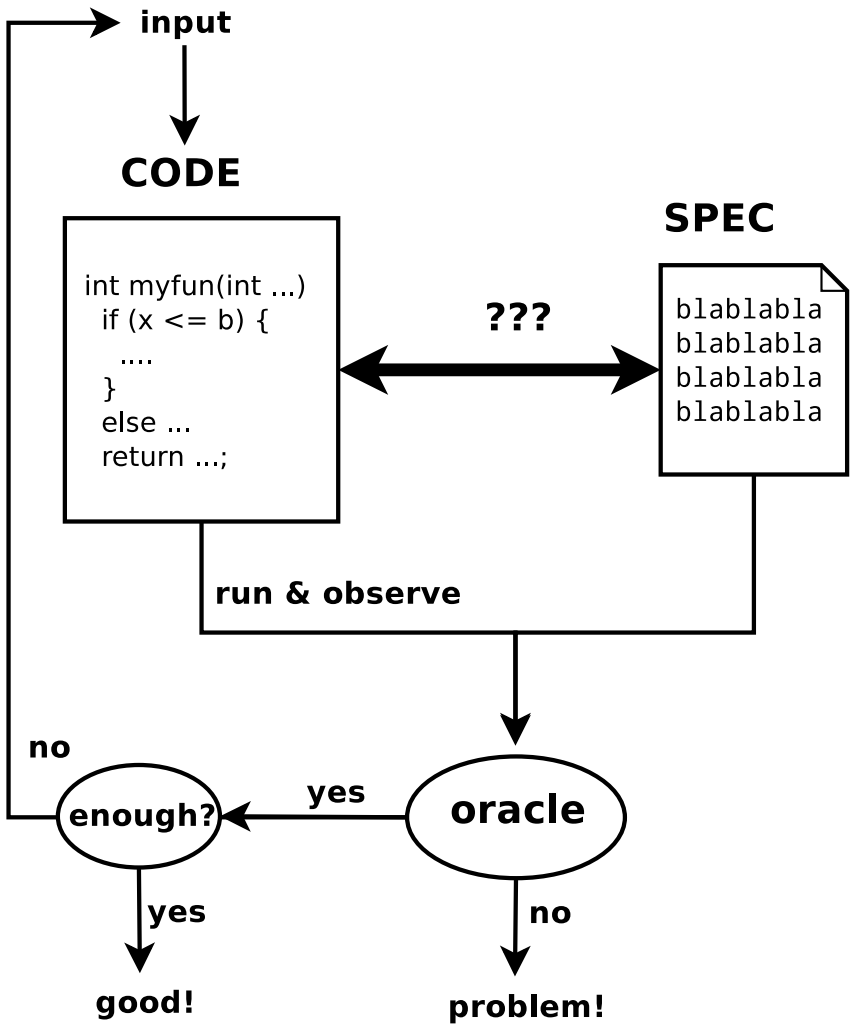
Process du test (1)



Process du test (1)



Process du test (1)



Process du test (2)

1. choisir un cas de test (CT) [≈ scénario] à exécuter
2. estimer le résultat attendu [oracle]
3. déterminer une donnée de test (DT) exerçant le CT, et son oracle concret [concrétisation]
4. exécuter le programme sur la DT [script de test]
5. comparer le résultat obtenu à l'oracle [verdict : pass/fail]
6. a-t-on assez de tests ? si oui stop, sinon goto 1
   [notion de critère de couverture]

Suite / Jeu de tests : ensemble de cas de tests

Exemple

```
int[] my-sort (int[] vec)
```
//@ Ensures : tri du tableau d'entrée + élimination de la redondance

Quelques cas de tests (CT) et leurs oracles :

CT1	tableau d'entiers non redondants	le tableau trié
CT2	tableau vide	le tableau vide
CT3	tableau avec 2 entiers redondants	trié sans redondance

Concrétisation : DT et résultat attendu

DT1	vec = [5,3,15]	res = [3,5,15]
DT2	vec = []	res = []
DT3	vec = [10,20,30,5,30,0]	res = [0,5,10,20,30]

Exemple (2)

Script de test

```
1  void testSuite() {
2
3    int[] td1 = [5,3,15] ;  /* prepare data  */
4
5    int[] oracle1 = [3,5,15] ;   /* prepare oracle */
6
7    int[] res1 =  my-sort(td1); /* run CT and  */
8                                /* observe result */
9
10   if (array-compare(res1,oracle1)) /* assess validity */
11   then print(``test1 ok'')
12   else {print(``test1 erreur'')};
13
14
15
16   ...  /* same for other test data  */
17
18
19 }
```

Qu'apporte le test ?

Le test ne peut pas prouver au sens formel la validité d'un programme

Testing can only reveal the presence of errors but never their absence.
- E. W. Dijkstra (Notes on Structured Programming, 1972)

Par contre, le test peut "augmenter notre confiance" dans le bon fonctionnement d'un programme

- correspond au niveau de validation des systèmes non informatiques

---

Un bon jeu de tests doit donc :

- exercer un maximum de "comportements différents" du programme [notion de critères de couverture]
- notamment
  - tests nominaux : cas de fonctionnement les plus fréquents
  - tests de robustesse : cas limites / délicats

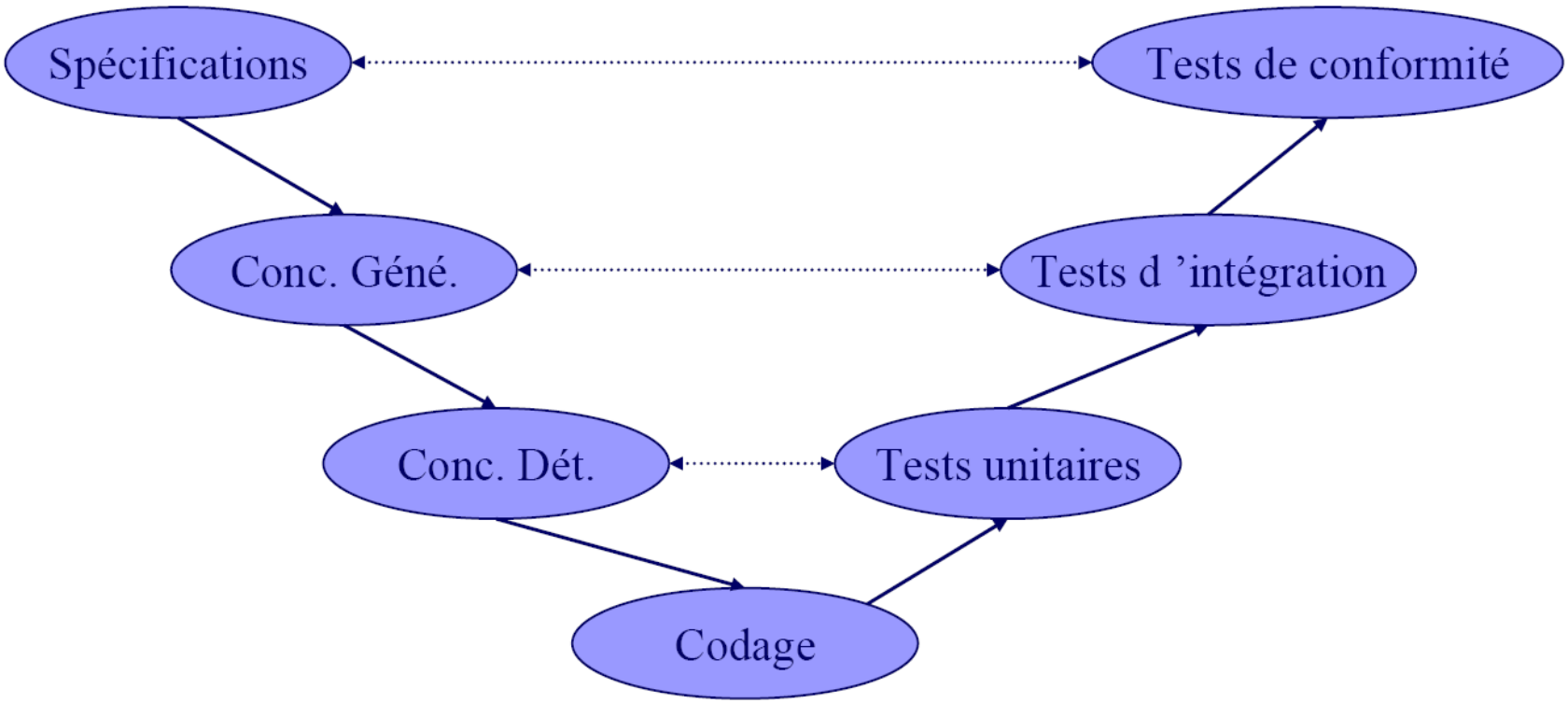
Plan

- Introduction au test logiciel
  - Éléments de classification

Selon la propriété testée

- correction fonctionnelle
- robustesse
- performances
- sécurité
- ergonomie
- ...

Selon le cycle de développement (1)



aussi : phase d'évolution / maintenance : tests de (non) régression

Selon le cycle de développement (2)

Tests unitaire : tester les différents modules en isolation

- définition non stricte de “module unitaire” (procédures, classes, packages, composants, etc.)
- uniquement test de correction fonctionnelle

Tests d'intégration : tester le bon comportement lors de la composition des modules

- uniquement test de correction fonctionnelle

Tests système / de conformité : valider l'adéquation du code aux spécifications

- on teste aussi toutes les caractéristiques émergentes

  sécurité, performances, etc.

Tests de validation / acceptance : valider l'adéquation aux besoins du client

- souvent similaire au test système, mais réaliser / vérifier par le client

Tests de régression : vérifier que les corrections / évolutions du code n'ont pas introduits de bugs

Selon la source de cas de tests (1)

Boîte Noire : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèles formels ou semi-formels

Boîte Blanche : à partir du code

- critères basés sur le graphe de flôt de contrôle
- critères basés sur le graphe de flôt de données
- mutations syntaxiques

Selon la source de cas de tests (1)

Boîte Noire : à partir de spécifications

- dossier de conception
- interfaces des fonctions / modules
- modèles formels ou semi-formels

Boîte Blanche : à partir du code

- critères basés sur le graphe de flôt de contrôle
- critères basés sur le graphe de flôt de données
- mutations syntaxiques

Probabiliste : *domaines des entrées + arguments statistiques*

- *random : distribution uniforme [ou pas ...]*
- *statistique : distribution mimant le profil opérationnel attendu*

Selon la source de cas de tests (2)

	boîte noire	boîte blanche
nécessite le code ?	non	oui
fautes de conformance	oui	erreur fonctionnelle : oui omission de fonctionalité : non
fautes de codage	non	oui
critères classiques	partition des entrées limites	instructions, branches
# tests	ok	potentiellement énorme
pb spécifique	concrétisation des CT en DT	oracle
quel niveau ?	tous	plutôt unitaire (ou critères light : couv. fonctions)

Boite Noire vs Boite Blanche (2)

Exemples de complémentarités des critères

Bug du Pentium : cause = erreurs sur quelques valeurs (parmis des millions) dans une table de calculs

- impossible à trouver en test boite noire
- aurait pu être trouvé facilement en test unitaire du module concerné

Bug du "Mars Climate Orbiter" : cause = problème de métriques (mètres vs pouces)

- chaque module fonctionne correctement, test unitaire inutile
- aurait pu être détecté en phase de tests d'intégration, en se basant sur l'interface de communication (boite noire)

Exécution du test : pas toujours simple

. Code manquant (test incrémental)

. Présence d'un environnement (réseau, Base de Données, etc.)

. Exécution d'un test très coûteuse en temps

. Hardware réel non disponible, ou peu disponible

. Possibilité de réinitialiser le système ?

- si non, l'ordre des tests devient très important

. Moyens d'observation et d'action sur le système

- sources dispo, compilables et instrumentables : cas facile, script = code
- si non : difficile, "script de test" = succession d'opérations (manuelles ?) sur l'interface disponible (informatique ? électronique ? mécanique ?)

Tests de (non) régression

Tests de régression : à chaque fois que le logiciel est modifié, s'assurer que "ce qui fonctionnait avant fonctionne toujours"

Pourquoi modifier le code déjà testé ?

- correction de défaut
- ajout de fonctionnalités

Quand ?

- en phase de maintenance / évolution
- ou durant le développement

Quels types de tests ?

- tous : unitaires, intégration, système, etc.

Objectif : avoir une méthode automatique pour

- rejouer automatiquement les tests *nécessaires* [perfs !]
- détecter les tests dont les scripts ne sont plus corrects

Dilemmes du test

Si la campagne de tests trouve peu d'erreurs

- choix 1 (optimiste) : le programme est très bon
- choix 2 (pessimiste) : les tests sont mauvais

Si la campagne de tests trouve beaucoup d'erreurs

- choix 1 (optimiste) : la majeure partie des erreurs est découverte
- choix 2 (pessimiste) : le programme est de très mauvaise qualité, encore plus de bugs sont à trouver

solution : qualité du jeu de tests [critères de couverture]

Sélection des tests

Problèmes de la sélection de tests :

- efficacité du test dépend crucialement de la qualité des CT/DT
- ne pas “râter” un comportement fautif
- MAIS les CT/DT sont coûteux (design, exécution, stockage, etc.)

Deux enjeux :

- DT suffisamment variées pour espérer trouver des erreurs
- maîtriser la taille : éviter les DT redondantes ou non pertinentes

Critères de couverture

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
- concis
- automatisable

Récapitulatif des problèmes

Sélection des CT/DT pertinents : très difficile

- expériences industrielles de synthèse automatique

Script de test : de facile à difficile, mais toujours très ad hoc

Verdict et oracle : très difficile

- certains cas particuliers s'y prêtent bien
- des oracles partiels automatisés peuvent être utiles

Régression : bien automatisé (ex : JUnit pour Java)

Rappel : deux aspects du test

1- Construire la qualité du produit

- lors de la phase de conception / codage
- en partie par les développeurs (tests unitaires)
- but = trouver rapidement le plus de bugs possibles (avant la commercialisation)
  - test réussi = un test qui trouve un bug

2- Démontrer la qualité du produit à un tiers

- une fois le produit terminé
- idéalement : par une équipe dédiée
- but = convaincre (organismes de certification, hiérarchie, client)
  - test réussi = un test qui passe sans problème
  - + tests jugés représentatifs
    (systèmes critiques : audit du jeu de tests)

Critères boite noire

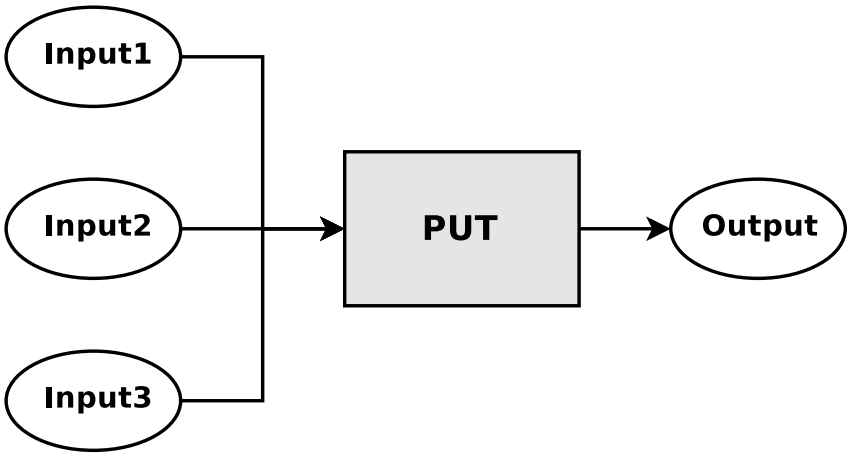
. Couverture des partitions des domaines d'entrée

. Test aux limites

. Approche pair-wise pour limiter la combinatoire

Remarque : si on dispose d'un modèle formel, on peut appliquer les critères de couverture boite blanche au niveau du modèle

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

Partition des entrées : principe

Principe :

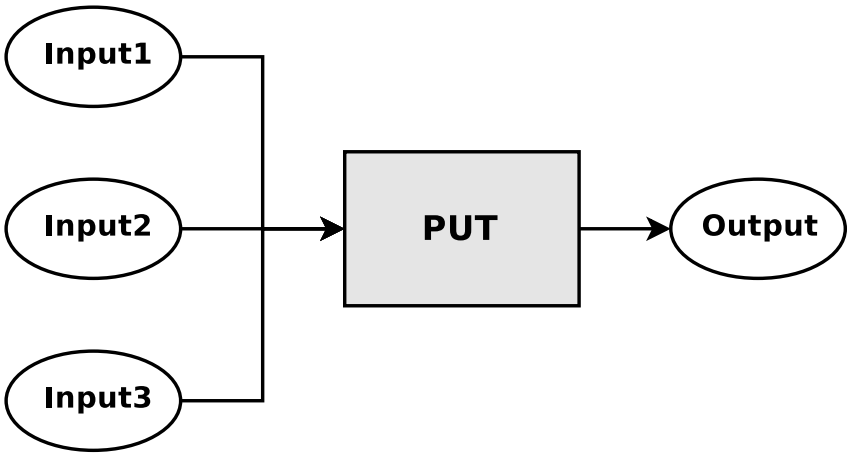
- diviser le domaine des entrées en un nombre fini de classes tel que le programme réagisse pareil (en principe) pour toutes valeurs d'une classe
- conséquence : il ne faut tester qu'une valeur par classe !
- $\Rightarrow$ permet de se ramener à un petit nomlbre de CTs

---

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

Rappel : 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

Graphe de flot de contrôle

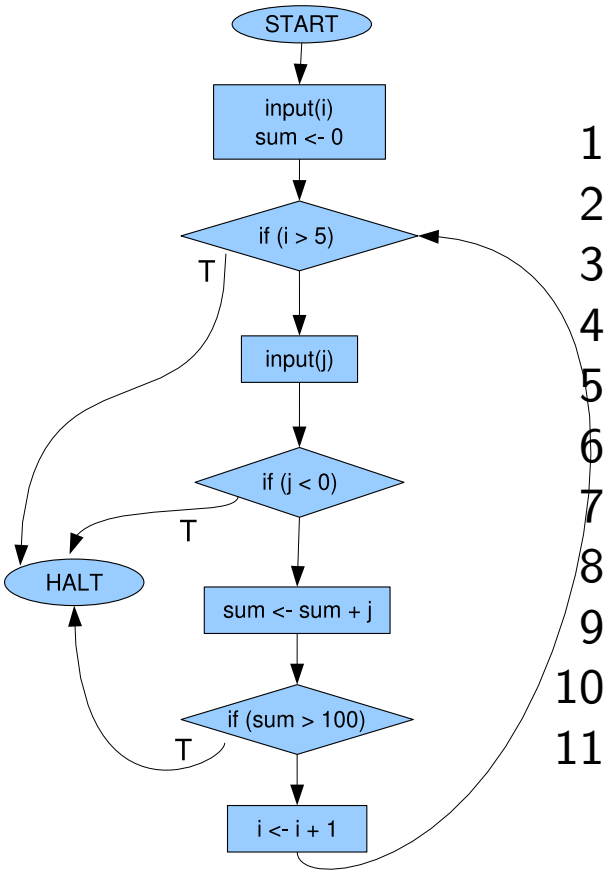
Le graphe de flot de contrôle d'un programme est défini par :

- un noeud pour chaque instruction, plus un noeud final de sortie
- pour chaque instruction du programme, le CFG comporte un arc reliant le noeud de l'instruction au noeud de l'instruction suivante (ou au noeud final si pas de suivant), l'arc pouvant être étiquetté par l'instruction en question

Quelques définitions sur les instructions conditionnelles :
`if (a<3 && b<4) then ... else ...`

- un `if` est une instruction conditionnelle / branchante
- `(a<3 && b<4)` est la condition
- les deux décisions possibles sont (*condition*, *true*) et (*condition*, *false*) (chaque transition)
- les conditions simples sont `a<3` et `b<4`

Graphe de flot de contrôle (2)




```
1     START
2     input(i)
3     sum := 0
4     loop : if (i > 5) goto end
5     input(j)
6     if (j < 0) goto end
7     sum := sum + j
8     if (sum > 100) goto end
9     i := i +1
10    goto loop
11    end : HALT
```

Quelques critères de couverture

Quelques critères de couverture sur flot de contrôle

- Tous les noeuds (I) : le plus faible.
- Tous les arcs / décisions (D) : test de chaque décision
- Toutes les conditions (C) : peut ne pas couvrir toutes les décisions
- Toutes les conditions/décisions (DC)
- Toutes les combinaisons de conditions (MC) : explosion combinatoire !
- Tous les chemins : le plus fort, impossible à réaliser s'il y a des boucles

Remarque : il existe d'autres critères boite blanche

- basés sur la couverture du flot de données
- basés sur les mutations syntaxiques du code

Tests de régression : problème SMP

Compromis entre tout rejouer (sûr mais trop cher) et ne pas rejouer assez.

- certains tests ne passent pas par les modifications : les ignorer

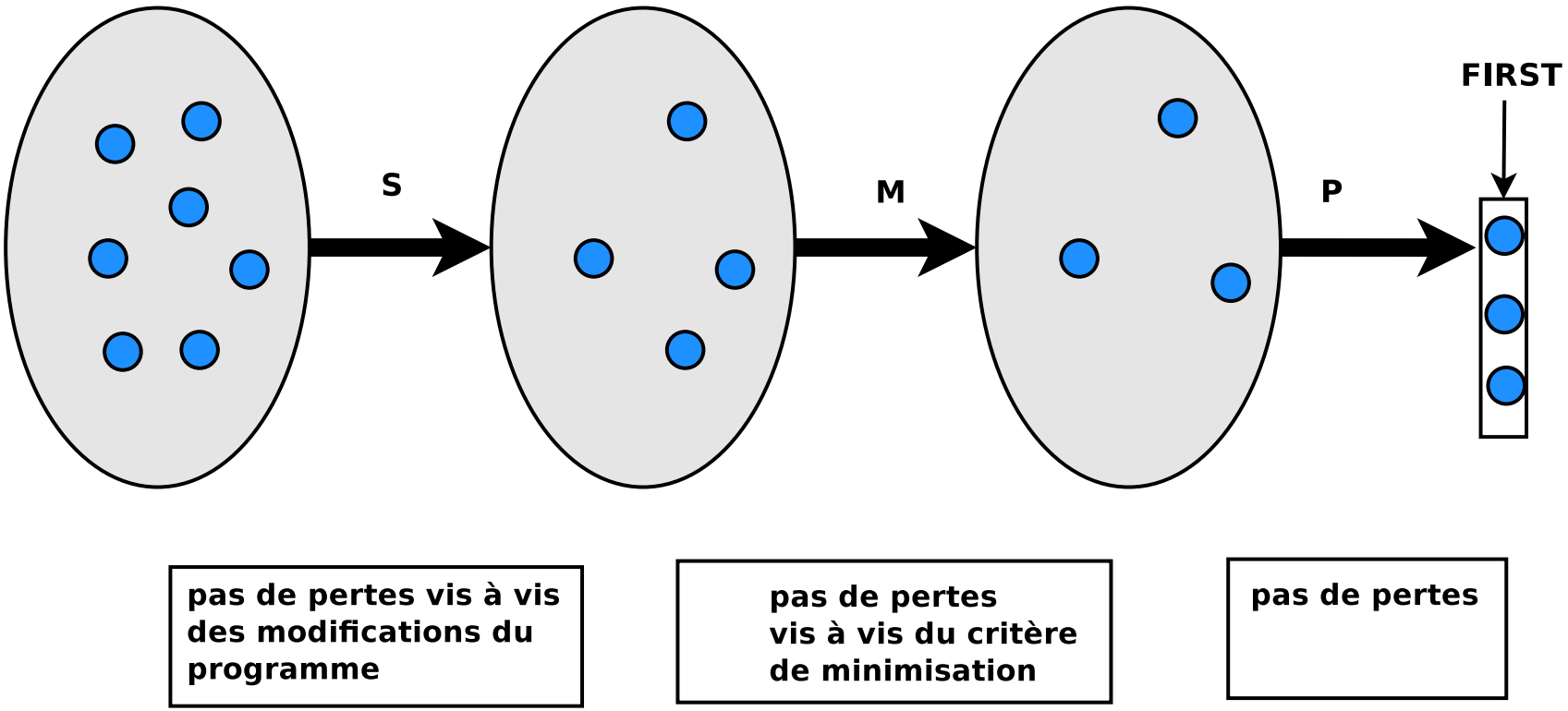
Problème additionnel : temps total pour le rejeu limité

- on arrête après N tests
- avec cette limite, le rejeu total est risqué
- faire tests pertinents d'abord

Trois phases distinctes dans la sélection :

- **S**électionner les tests pertinents (aucune perte)
- **M**inimiser les tests pertinents (perte possible)
- **P**rioritiser les tests restants (aucune perte)

Tests de régression : problème SMP (2)



Philosophie du test

> Testing can only reveal the presence of errors but never their absence
> - E. W. Dijkstra (Notes on Structured Programming, 1972)

Oui, mais ...

- Correspond au niveau de fiabilité exigé du reste du système
- Correspond aux besoins réels de beaucoup d'industriels
- Peut attaquer des programmes + complexes

Test et vérification (avis personnel)

Opposition historique forte

Complémentaire au niveau fonctionnel

- propriété prouvée = pas besoin de tests
- propriété non prouvée = peut être fausse ? (test !)
- certaines classes de propriétés sont pour le moment non modélisables [perfs, ergonomie, etc.]
- facilité de mise en oeuvre : AS unsound $\leq$ test $\leq$ AS sound

De plus en plus similaire en terme de technologie

- preuve d'invariance vs preuve d'accessibilité
- mêmes outils : logique, sémantique, analyse de programme, etc.
- mais approximations différentes (sur- vs sous-), importance de la synthèse
- de plus en plus de techniques "hybrides" (bounded model checking, context-bounded analysis, etc.)
- remarque : langage de spécification utile pour vérification et test [next big step in industry ?]