Automatisation du Test Logiciel

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr

Plan

- Introduction
- Automatisation de la génération de tests
- Critères de test avancés

Plan

- Contexte
- Définition du test
- Aspects pratiques
- Discussion

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 (2)

Validation et Vérification (V & V)

- Vérification : est-ce que le logiciel fonctionne correctement ?
  - *"are we building the product right ?"*
- Validation : est-ce que le logiciel fait ce que le client veut ?
  - *"are we building the right product ?"*

Quelles méthodes ?

- revues
- simulation/ animation
- tests méthode de loin la plus utilisée
- méthodes formelles encore très confidentielles, même en syst. critiques

Coût de la V & V

- 10 milliards $/an en tests rien qu'aux US
- plus de 50% du développement d'un logiciel critique (parfois $> 90\%$)
- en moyenne 30% du développement d'un logiciel standard

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

Plan

- Contexte
- Définition du test
- Aspects pratiques
- Discussion

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)

Process du test

Process (1 seul test)

1. choisir un cas de tests (CT) = scénario à exécuter
2. estimer le résultat attendu du CT (Oracle)
3. déterminer (1) une donnée de test (DT) suivant le CT, et (2) son oracle concret (concrétisation)
4. exécuter le programme sur la DT (script de test)
5. comparer le résultat obtenu au résultat attendu (verdict : pass/fail)

Script de Test : code / script qui lance le programme à tester sur le DT choisi, observe les résultats, calcule le verdict

Suite / Jeu de tests : ensemble de cas de tests

Exemple

Spécification : tri de tableaux d'entiers + enlèver la redondance

Interface : `int[] my-sort (int[] vec)`

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    int[] oracle1 = [3,5,15] ;   /* prepare oracle */
5    int[] res1 =  my-sort(td1); /* run CT and  */
6                                /* observe result */
7    if (array-compare(res1,oracle1)) /* assess validity */
8    then print(``test1 ok'')
9    else {print(``test1 erreur'')};
10
11
12   int[] td2 = [] ;  /* prepare data  */
13   int[] oracle2 = [] ;   /* prepare oracle */
14   int[] res2 =  my-sort(td2);
15   if (array-compare(res2,oracle2)) /* assess validity */
16   then print(``test2 ok'')
17   else {print(``test2 erreur'')};
18
19
20   ...  /* same for TD3  */
21
22
23 }
```

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 test)
- notamment
  - tests nominaux : cas de fonctionnement les plus fréquents
  - tests de robustesse : cas limites / délicats

Deux aspects différents du test

1- Contribuer à assurer 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- Validation : Démontrer la qualité à un tiers

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

Éléments de classification

Critère de tests

- boite blanche / boite noire / probabiliste

Phase du processus de test

- test unitaire, d'intégration, système, acceptation, regression

Phase du processus de test

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

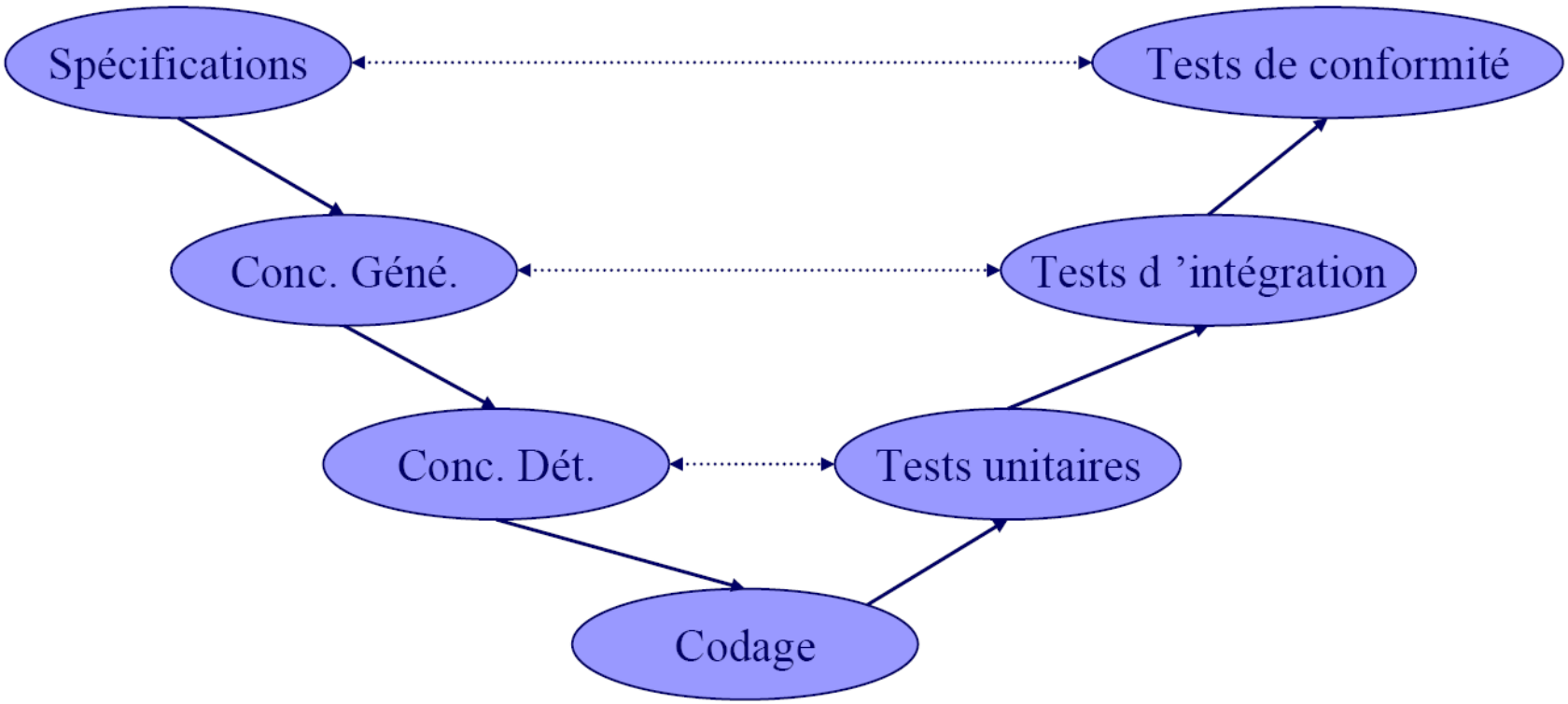
  sécurité, performances, etc.

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

Phase du processus de test (2)



Critères de tests

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

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 : taille raisonnable
- 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

Test en “boîte blanche”

- La structure interne du système doît être accessible
- Se base sur le code : très précis, mais plus “gros” que les spécifications
- 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

Test aléatoire / statistique

Les données sont choisies dans leur domaine selon une loi statistique

- loi uniforme (test aléatoire)
- loi statistique du profil opérationnel (test statistique)

Pros/Cons du test aléatoire

- sélection aisée des DT en général
- test massif si oracle (partiel) automatisé
- "objectivité" des DT (pas de biais)
- PB : peine à produire des comportements très particuliers (ex : x=y sur 32 bits)

Pros/Cons du test statistique

- permet de déduire une garantie statistique sur le programme
- trouve les défauts les plus probables : défauts mineurs ?
- PB : difficile d'avoir la loi statistique

Plan

- Contexte
- Définition du test
- Aspects pratiques
- Discussion

Oracle et verdict

La définition de l'oracle est un problème très difficile

- limite fortement certaines méthodes de test (ex : probabiliste, BN)
- impose un trade-off avec la sélection de tests
- point le plus mal maitrisé pour l'automatisation

Oracle et verdict (2)

Quelques cas pratiques d'oracles parfaits automatisables

- comparer à une référence : logiciel existant, tables de résultats
- résultat simple à vérifier (ex : solution d'une équation)
- disponibilité d'un logiciel similaire : test dos à dos

Des oracles partiels mais automatisés peuvent être utiles

- oracle le plus basique : le programme ne plante pas
- instrumentation du code (`assert`)
- plus évolué : programmation avec contrats (Eiffel, Jml pour Java)

Exécution du test : Script de Test

Composition du script de test

- **préambule** : amène le programme dans la configuration voulue pour le test (ex : initialisation de BD, suite d'émissions / réceptions de messages, etc.)
- **corps** : appel des "stimuli" testés (ex : fonctions et DT)
- **identification** : opérations d'observations pour faciliter / permettre le travail de l'oracle (ex : log des actions, valeurs de variables globales, etc.)
- **postambule** : retour vers un état initial pour enchainer les tests

Le script doit souvent inclure de la glue avec le reste du code

- **bouchon** : simule les fonctions appelées mais pas encore écrites

Exécution du test : pas toujours simple

Quelques exemples de problèmes

Code manquant (test incrémental)

Exécution d'un test très coûteuse en temps

Hardware réel non disponible, ou peu disponible

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

- comment le prendre en compte ? (émulation ?)

Réinitialisation possible du système ?

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

Forme du script et moyens d'action sur le programme ?

- 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 ?)

Exécution : pas toujours simple (2)

Message :

- code “desktop” sans environnement : facile
- code embarqué temps réel peut poser de sérieux problèmes, solutions ad hoc

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
- détecter les tests dont les scripts ne sont plus (syntaxiquement) corrects

Solution “à la JUnit”

Junit pour Java : idée principale = tests écrits en Java

- simplifie l'exécution et le rejeu des tests (juste tout relancer)
- simplifie la détection d'une partie des tests non à jour : tests recompilés en même temps que le programme
- simplifie le stockage et la réutilisation des tests (tests de `MyClass` dans `MyClassTest`)

JUnit offre :

- des primitives pour créer un test (assertions)
- des primitives pour gérer des suites de tests
- des facilités pour l'exécution des tests
- statistiques sur l'exécution des tests
- interface graphique pour la couverture des tests
- points d'extensions pour des situations spécifiques

Solution très simple et extrêmement efficace

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

Sélection des Tests (2)

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

---

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =
  if (x=500 and y=600) then x-y          (bug 1)
  else x+y                               (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Sélection des Tests (2)

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

---

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Sélection des Tests (2)

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

---

```
fun (x:int, y:int) : int =
  if (x=500 and y=600) then x-y          (bug 1)
  else x+y                               (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

Sélection des Tests (2)

Les deux familles de tests BB et BN sont complémentaires

- Les approches structurelles trouvent plus facilement les défauts de programmation
- Les approches fonctionnelles trouvent plus facilement les erreurs d'omission et de spécification

---

Spec : retourne la somme de 2 entiers modulo 20 000

```
fun (x:int, y:int) : int =
  if (x=500 and y=600) then x-y          (bug 1)
  else x+y                               (bug 2)
```

- fonctionnel : bug 1 difficile, bug 2 facile
- structurel : bug 1 facile, bug 2 difficile

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
- concis
- automatisable

Critères boite blanche (1)

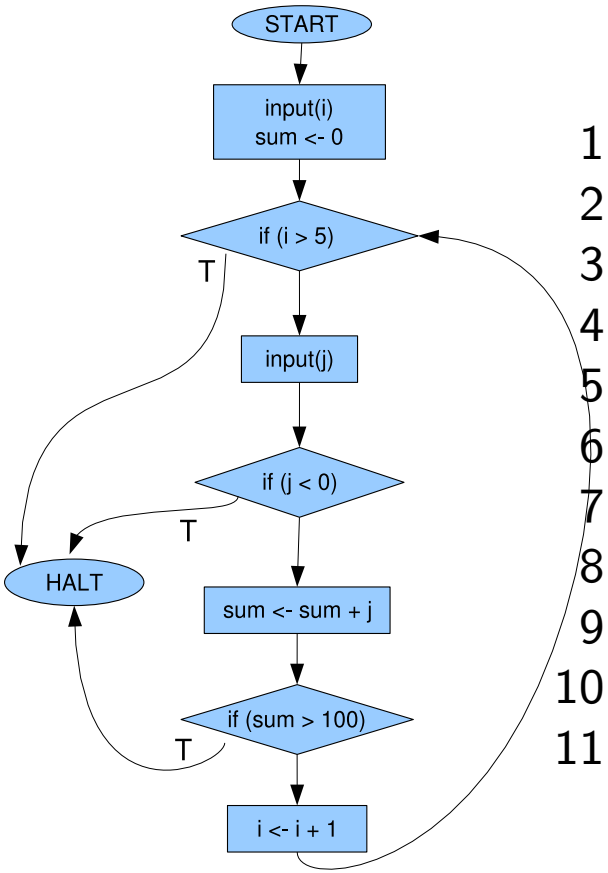
Le graphe de flot de contrôle d'un programme est défini par :

- 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`

Critères boite blanche (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
```

Critères boite blanche (3)

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

Critères boite blanche (4) - MCDC

Utilisé en avionique (DO-178B). But :

- puissance entre DC et MC
- ET garde un nombre raisonnable de tests

Définition

- critère DC
- ET les tests doivent montrer que chaque condition atomique peut influencer la décision :

  par exemple, pour une condition $C = a \wedge b$, les deux DT $a = 1, b = 1$ et $a = 1, b = 0$ prouvent que b seul peut influencer la décision globale C

Critères boite blanche (5) - Hiérarchie des critères

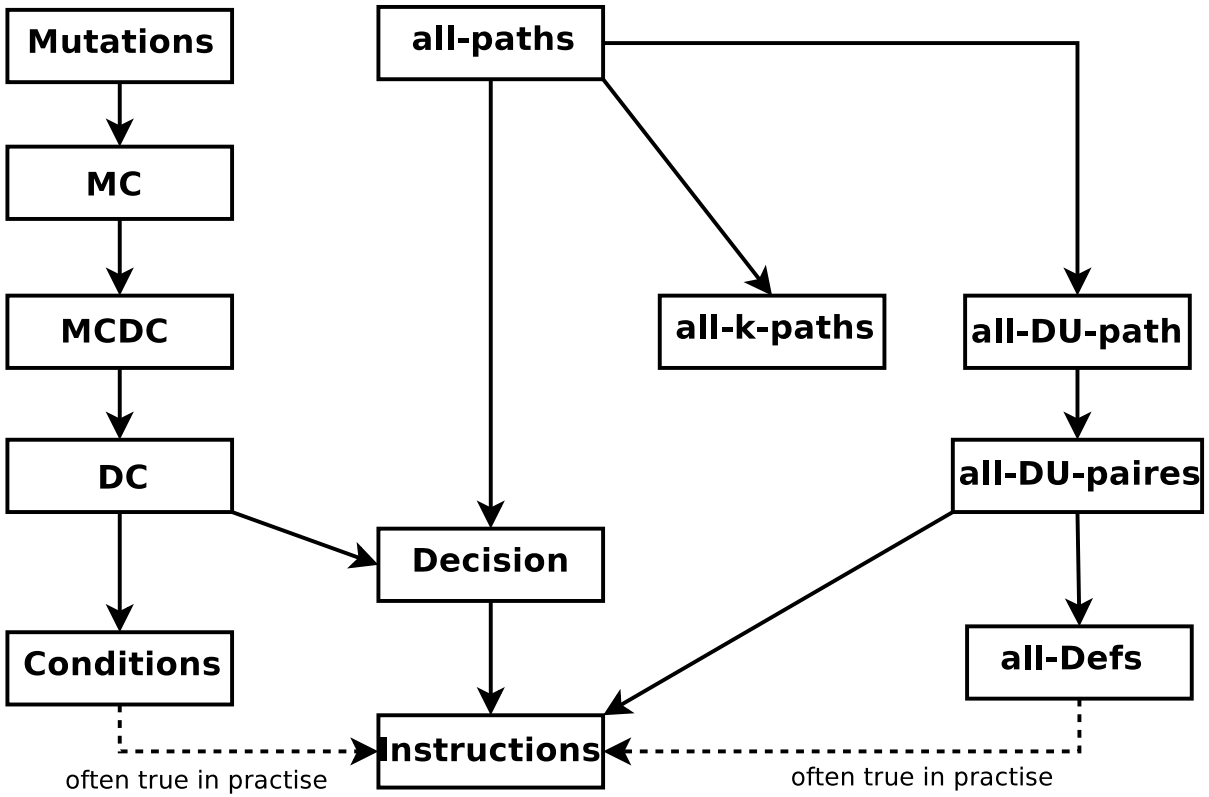
Notion de hiérarchie entre ces différents critères de couverture

Le critère CT1 est plus fort que le critère CT2 (CT1 *subsumes* CT2, noté CT1 $\succeq$ CT2) si pour tout programme P et toute suite de tests TS pour P, si TS couvre CT1 (sur P) alors TS couvre CT2 (sur P).

Exercice : supposons que TS2 couvre CT2 et trouve un bug sur P, et TS1 couvre un critère CT1 tq CT1 $\succeq$ CT2.

Question : TS1 trouve-t-il forcément le même bug que TS2 ?

Critères boite blanche (6) - Hiérarchie des critères



Limite des critères de tests existants

Ne sont pas reliés à la qualité finale du logiciel (MTBF, PDF, ...)

- sauf test statistique

Ne sont pas non plus vraiment reliés au # bugs /kloc

- exception : mcdc et contrôle-commande
- exception : mutations

Mais toujours mieux que rien ...

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)

Bibliographie

Livres

 Introduction to software testing [Ammann-Offutt 08]

 Foundations of Software Testing [Mathur 08]

 Art of Software Testing (2nd édition) [Myers 04]

 Software Engineering [Sommerville 01]

Plan

- Contexte
- Définition du test
- Aspects pratiques
- Discussion

Automatisation du test

Test = activité difficile et coûteuses

Difficile

- trouver les défauts = pas naturel (surtout pour le programmeur)
- qualité du test dépend de la pertinence des cas de tests

Coûteux : entre 30 % et 50 % du développement

Besoin de l'automatiser/assister au maximum

Gains attendus d'une meilleur architecture de tests

- amélioration de la qualité du logiciel
- et/ou réduction des coûts (développement - maintenance) et du time-to-market

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

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

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

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

- déjà utilisé : ne modifie ni les process ni les équipes
- retour sur investissement proportionnel à l'effort
- simple : pas d'annotations complexes, de faux négatifs, ...
- robuste aux bugs de l'analyseur / hypotheses d'utilisation
- trouve des erreurs non spécifiées

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

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

Offre des solutions (partielles) pour

- librairies sous forme binaire (COTS)
- codes mélangeant différents langages (assembleur, SQL, ...)
- code incomplet

Citations

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

- Donald Knuth (1977)

It has been an exciting twenty years, which has seen the research focus evolve [. . .] from a dream of automatic program verification to a reality of computer-aided [design] debugging.

- Thomas A. Henzinger (2001)

Plan

- Introduction
- Automatisation de la génération de tests
- Critères de test avancés

Génération de tests : Approches structurelles

On se concentre dans cette partie sur la génération de données de test à partir du code

L'oracle est vu comme un problème orthogonal

On suppose qu'on dispose d'un oracle automatisé

- oracle exact dans certains cas (test dos à dos)
- oracle partiel sinon : assertions, contrats (JML, Spec#)

Génération de tests structurels par contraintes

Principe : transformer tout ou partie du programme en une formule logique φ telle que solution de φ = DT cherchée

Approche globale : tout le programme est transformé en une formule logique

- théories complexes : quantificateurs ou points fixes pour les boucles
- comment transformer le programme (boucles) ?

Approche locale / orientée chemin : un seul chemin est considéré à la fois

- théories plus simples : sans quantificateur, juste conjonction
- mais énumération de chemins
- nous verrons deux techniques
  - exécution symbolique
  - exécution symbolique dynamique (dite aussi exécution concolique)

Plan

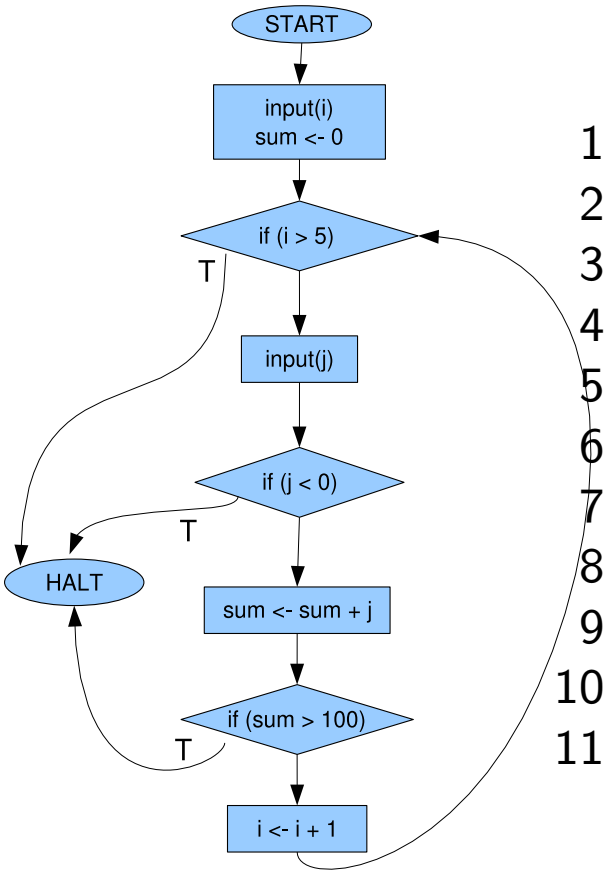
- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Control-Flow Graph (CFG)

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)

Control-Flow Graph (CFG)




```
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
```

Notion centrale : le prédicat de chemin

π un chemin (fini) du programme P (c-à-d $\pi \in L(P)$, si P vu comme automate)

D l'espace des entrées du programme (arguments, variables volatiles, etc.)

$V \in D$ une entrée du programme

T une théorie logique

On note $P(V)$ la trace d'exécution de P lancé sur la donnée d'entrée V

On note par $\preceq$ la relation de préfixe entre les chemins
($\approx$ préfixes de mots, $ab \preceq abc$)

Prédicat de chemin

Un prédicat de chemin pour π est une formule logique $\varphi_\pi \in T$ interprétée sur D telle que si $V \models \varphi_\pi$ alors l'exécution du programme sur V suit le chemin π, c'est à dire que $\pi \preceq P(V)$.

Prédicat de chemin (2)

Soit des variables sur un domaine D quelconque, φ une formule dans une logique interprétée sur D, et t une transition d'un programme.

On note $x \xrightarrow{t} y$ pour indiquer que la valuation $y \in D$ est obtenue en appliquant la transition t à la valuation $x \in D$.

$[\![\varphi]\!]$ est l'ensemble des $d \in D$ tq $d \models \varphi$.

$wpre(t, X')$: ensemble $X \subseteq D$ tq $\forall x \in X, \forall y$ *tq* $x \xrightarrow{t} y$ *alors* $y \in X'$

ensemble des éléments dont tous les successeurs par t sont dans X'

$wpre(t, \varphi')$: formule φ tq $[\![\varphi]\!] = wpre(t, [\![\varphi']\!])$

$post(X, t)$: ensemble $X' \subseteq D$ tq $\forall y \in X', \exists x \in X$ *telque* $x \xrightarrow{t} y$

ensemble des éléments ayant au moins un prédécesseur par t dans X

$post(\varphi, t)$: formule φ' tq $[\![\varphi']\!] = post([\![\varphi]\!], t)$

Prédicat de chemin (3)

Soit un un chemin du programme P : $\pi = \xrightarrow{t_1} \xrightarrow{t_2} \ldots \xrightarrow{t_n}$

Alors

- le prédicat de chemin le plus faible de π est défini par : $\bar{\varphi_\pi} = wpre(t_1, wpre(t_2, \ldots wpre(t_n, \top)))$
- conséquence : un prédicat de chemin quelconque φ_π pour π vérifie : $\varphi_\pi \Rightarrow \bar{\varphi_\pi}$

Prédicat de chemin (4)

Un prédicat de chemin pour π peut se calculer en exécutant symboliquement le chemin

- exécution concrète : màj des valeurs des variables
- exécution symbolique : màj des relations logiques entre variables (calcul avec *post*)

Mémoire concrète / symbolique

- concret : (variable, point de contrôle) $\rightarrow$ valeur concrète
- symbolique : (variable, point de contrôle) $\rightarrow$ formule logique

Prédicat de chemin (5)

Formellement, on utiliser le calcul en avant (*post*) pour calculer un prédicat de chemin

prédicat de chemin de π calculé en avant :
$\bar{\varphi_\pi}' = post(post(post(\top, t_1), t_2) \ldots, t_n)$

relation entre les deux approches (valable pour un chemin du programme) :

- $\bar{\varphi_\pi}' \Leftrightarrow \bar{\varphi_\pi}$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Prédicat de chemin (6)

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Prédicat de chemin (entrées Y_0 et Z_0)

$$\top \wedge W_1 = Y_0 + 1 \wedge X_2 = W_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$$

Projection sur les entrées $Y_0 + 4 < 2 \times Z_0 \wedge Y_0 + 4 \geq Z_0$

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Exécution symbolique

Génération de tests basée sur les chemins

1. choisir un chemin π du CFG
2. calculer un de ses prédicats de chemin φ_π
3. résoudre φ_π : une solution = une DT exerçant le chemin π
4. si couverture incomplète, goto 1

Idée ancienne, mais automatisation complète récente

PathCrawler, Dart, Cute, Exe

- concept introduit par King dans les années 1970
- au début : tout à la main, utilisateur se débrouille
- ensuite : on crée φ_π, puis utilisateur se débrouille
- automatisation complète sur des programmes : 1995-2005 (CEA)
- regain d'intérêt récent (Berkeley, CMU, Microsoft, Stanford)

Procédure symbolique de base (parcours DFS)

Variable globale **Tests** initialisée à $\emptyset$

Procédure principale : SEARCH(node_init, ε, $\top$)
/ màj Tests, ensemble de paires (TD,π) */*

procedure SEARCH(node, π, Φ)
input : CFG node, path prefix π, path predicate Φ for π
output : no result, update Tests

```
 1: Case node of
 2: | ε →        /* end node */
 3:       try S_p := SOLVE(Φ) ; Tests := Tests + {(S_p, π)} /* new TD */
 4:       with unsat → () ;
 5:       end try
 6: | block i → SEARCH(node.next, π · node, Φ ∧ SYMB(i))
 7: | goto tnode → SEARCH(tnode, π · node, Φ)
 8: | ite(cond,inode,tnode) →     /*branching*/
 9:       SEARCH(inode, π · node,Φ ∧ symb(cond)) ;
10:       SEARCH(tnode, π · node,Φ ∧ ¬symb(cond))
11: end case
```

Procédure symbolique de base (2)

Procédure SOLVE : $T \mapsto \{OK(TD), KO\}$

Procédure SYMB : Instr $\mapsto T$

- transforme une instruction de base en formule
- exemple : `x:=x+1` $\rightarrow X_1 = X_0 + 1$
- attention : introduire une nouvelle variable logique à chaque nouvelle utilisation d'une variable du programme

Procédure symbolique de base (3)

$expr \ ::= \ | \ V_C \ | \ k \in \mathbb{N} \ | \ expr \ (+,\text{-},*) \ expr$

Expressions de la théorie logique T définies par

$$\text{termF} : := k \in \mathbb{N} \mid V_F \\ \mid \text{termF} +_F \text{termF} \mid \text{termF} -_F \text{termF} \mid \text{termF} \times_F \text{termF}$$

let SYMB e = match e with

- $V_C \rightarrow \alpha(V_c)$ // *fonction de renommage*
- $k \rightarrow k$
- $e_1 \ (+,\text{-},*) \ e_2 \rightarrow \text{SYMB}(e_1) \ (+_F, -_F, \times_F) \ \text{SYMB}(e_2)$

SYMB définit de manière similaire sur les conditions

Procédure symbolique de base (4)

Pourquoi $\alpha(V_c)$:

- les “variables” du programme C peuvent être modifiées à chaque étape de l'exécution
- les “variables” de la théorie T sont des inconnues, de valeur constante
- le renommage est nécessaire pour prendre en compte la dynamique de l'exécution
  - prédicat de chemin pour `x := x+1` ?
  - $X_{n+1} = X_n + 1$, plutôt que $X = X + 1$

Propriétés : correction et complétude

Le calcul de prédicat de chemin est :

- correct s'il produit un prédicat de chemin plus fort que le prédicat de chemin le plus lâche
- complet s'il produit un prédicat de chemin *équisatisfiable* au prédicat de chemin le plus lâche

Le calcul symbolique symbolique est correct (resp. complet) si :

- le calcul de prédicat de chemin est correct (resp. complet)
- le solveur est correct et complet pour la théorie considérée

Propriétés

Correction si le calcul symbolique est correct, alors la procédure est correcte : chaque DT généré suit le chemin prévu

Complétude si le calcul symbolique est complet, alors la procédure est complète : quand la procédure termine, chaque chemin faisable est couvert

Terminaison la procédure termine ssi le nombre de chemins est fini

Intérêt pour la vérification

La procédure produit des témoins d'accessibilité :

- on peut vérifier le résultat fournit par des outils externes simples (calcul de couverture)
- un couple (DT, π) est plus facile à comprendre humainement que des invariants
- les DT peuvent être exportées vers des outils classiques de gestion de tests (couverture, tests de régression)

Correction : chaque DT généré suit le chemin prévu

- pas de faux positifs ! !
- un bug reporté est un bug trouvé
- la couverture du jeu de tests fourni est effectivement atteinte
- les instructions couvertes lors de l'exécution symboliques sont vraiment atteignables

MAIS : La complétude n'est que rarement obtenue, car le nombre de chemins doit être limité a priori

Ajouts classiques à la procédure de base (1)

Méthode de base : couverture de chemins, mais ...

Base pour d'autres critères (instructions ou branches)

- arrêt lorsque le taux de couverture est suffisant
- guide le choix des chemins

Paramètres : théorie logique, énumération de chemins

Ajouts classiques à la procédure de base (2)

Ajouts classiques à la procédure

1. borne sur la longueur des chemins
2. time out sur le solveur
3. gestion de couverture (instructions, branches)

Les points 1. et 2. cassent la propriétés de complétude pour assurer terminaison et temps de calcul raisonnable

En pratique, l'hypothèse de calcul symbolique parfait (correct + complet) est difficile à obtenir.

- pour du test, il vaut mieux garder la correction et sacrifier la complétude (cohérent avec la restriction arbitraire du nombre de chemins)
- remarque : dans le cas concolique (cf + tard), on peut imaginer se passer dans une certaine mesure de la correction du solveur

Problèmes de l'exécution symbolique

Passage à l'échelle / Performances (cf plus tard dans le cours)

- coût d'un appel au solveur
- nombre de chemins

Exploration (inutile) de chemins infaisables (PB1)

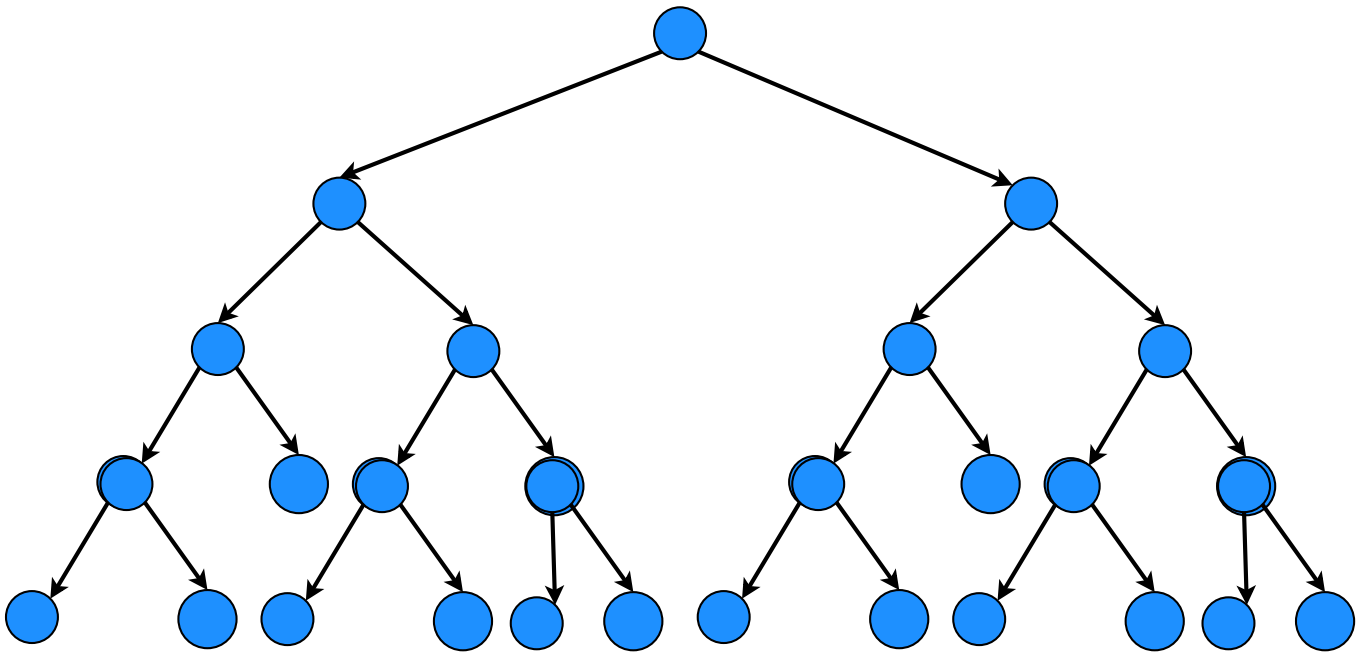
- pas de détection : coûteux en # chemins inutiles explorés
- détection au plus tôt : coûteux en # appels solveurs

Constructions du langage hors de portée de la théorie choisie (PB2)

- opérations non linéaire
- assembleur incorporé, bibliothèques en code natif

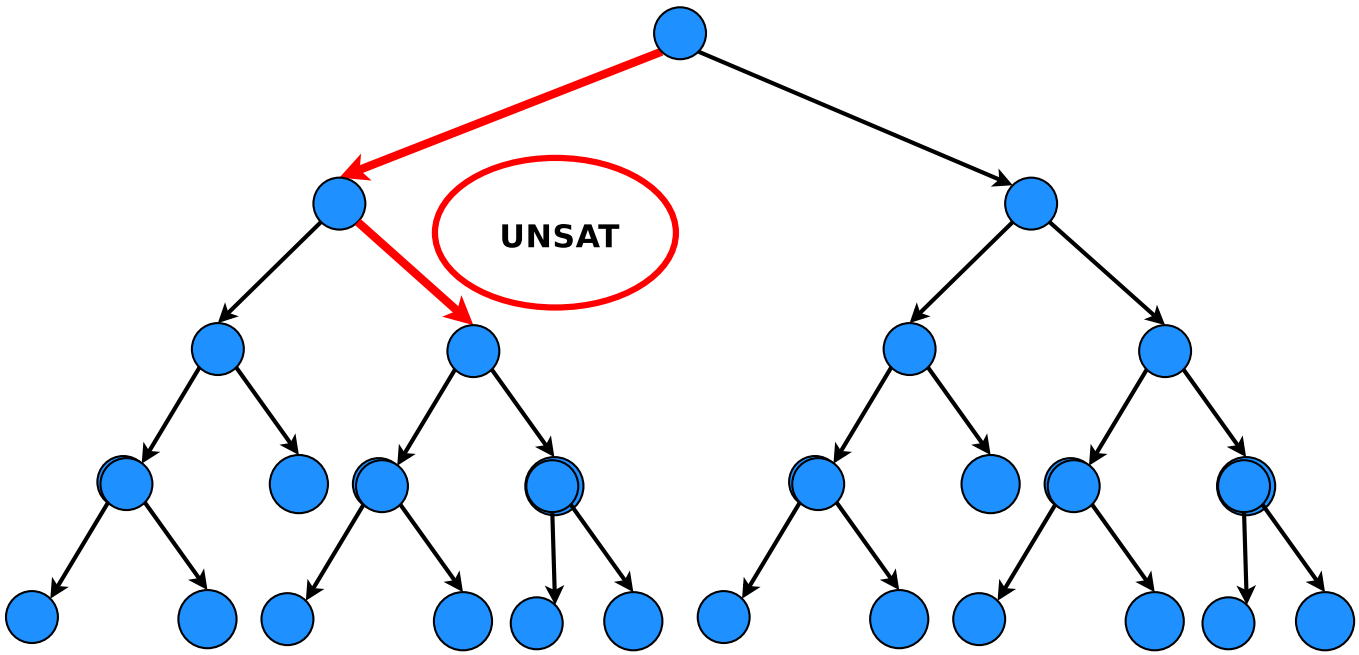
L'exécution concolique apporte des solutions aux 2 derniers problèmes (cf après)

Problème PB1 : chemins infaisables



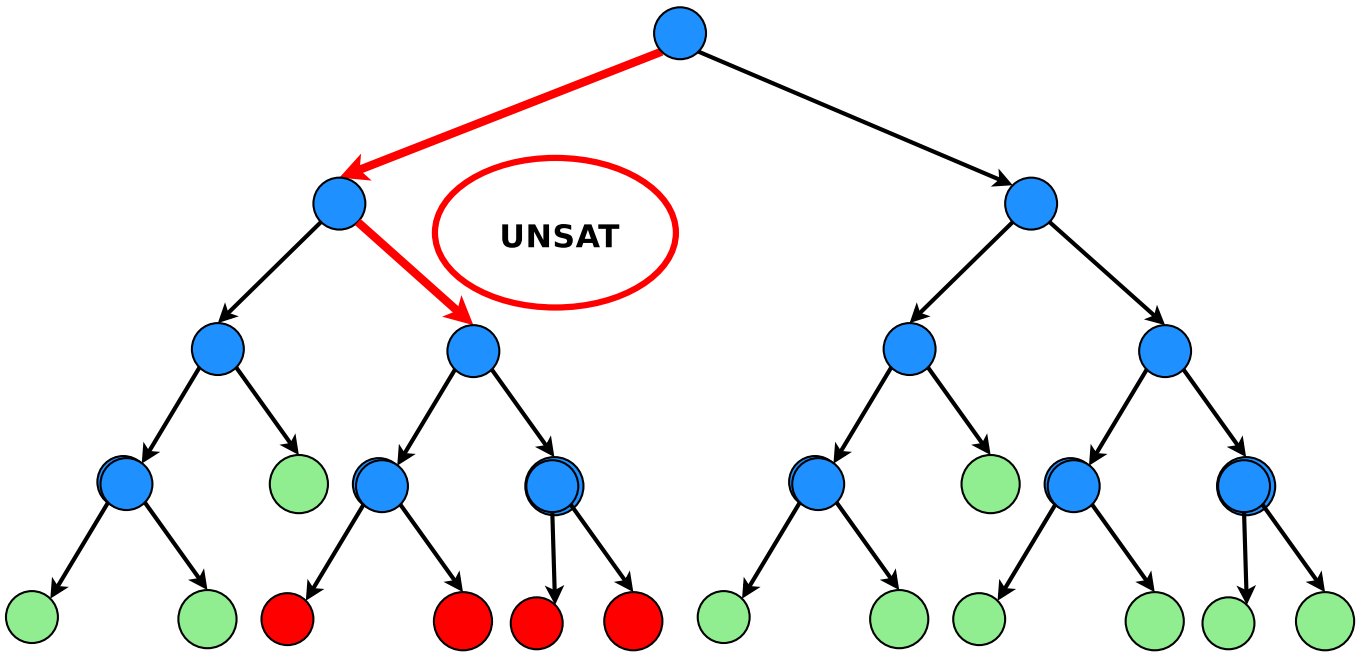
Supposons un chemin infaisable dans l'arbre des exécutions possibles

Problème PB1 : chemins infaisables



Supposons un chemin infaisable dans l'arbre des exécutions possibles

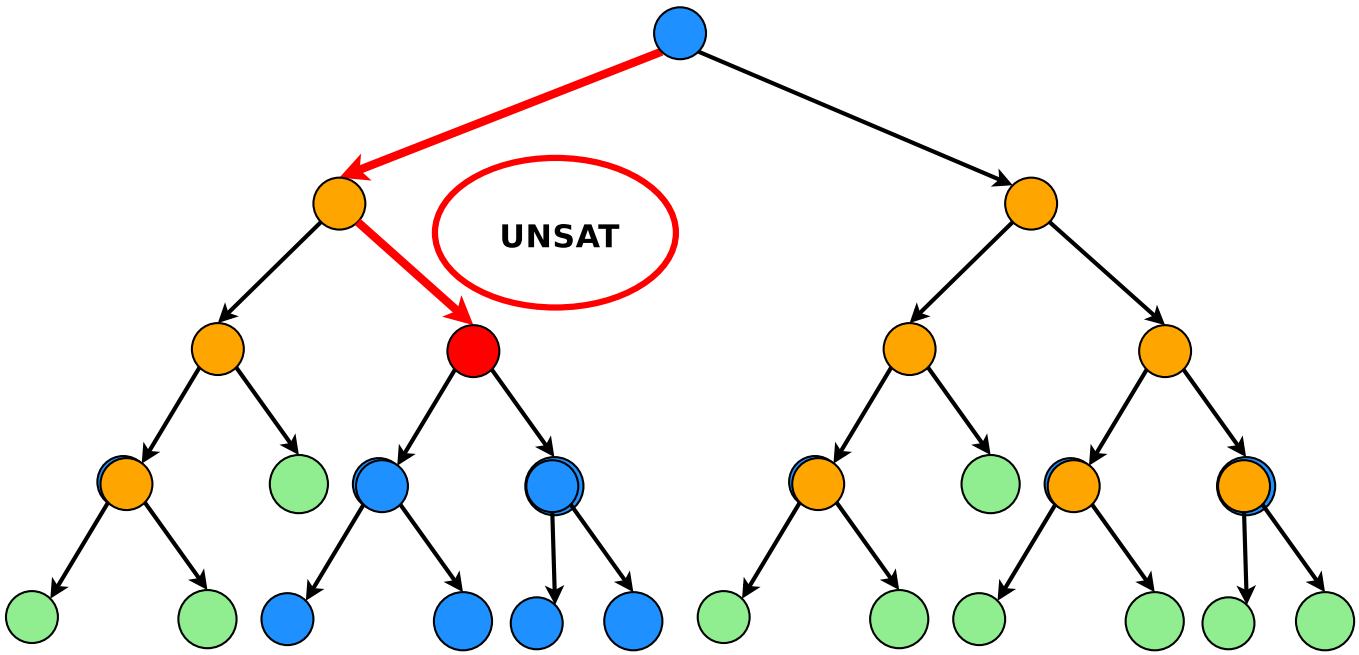
Problème PB1 : chemins infaisables



Méthode usuelle : résoudre le prédicat à la fin du chemin

- + : un appel au solveur par chemin (sur un arbre : 2^N)
- - : on peut continuer la recherche à partir de préfixes UNSAT
- KO sur programmes avec beaucoup de chemins infaisables

Problème PB1 : chemins infaisables



Alternative : résoudre le prédicat à chaque branche

- $+$: détecte UNSAT au plus tôt
- - : un appel au solveur par préfixe de chemin faisable, et un appel au solveur par préfixe minimal infaisable (sur un arbre : $2 * 2^N - 1$)
- KO sur programmes avec peu de chemins infaisables

Problème PB2

Un problème classique : constructions du langage hors de portée de la théorie choisie

Générer un test pour `f` atteignant `ERROR` ci-dessous
(théorie = arithmétique linéaire)

```
g(int x) {return x*x+(x modulo 2); }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }else OK }
```

Problème

- Une exécution symbolique génère une expression symbolique de type $Z = X * X + (X \ \mathit{modulo}\ 2)$
- Cette expression n'est pas solvable en arithmétique linéaire

Solutions classiques tirées de l'analyse statique / preuve de programme

- surapproximation
- ici par exemple $Z = \top$.
- PROBLEME : on perd la correction
- le TD généré peut ne pas suivre le chemin prévu

Teaser

L'exécution *concolique* offre :

- une solution très élégante à PB1
  - détecte UNSAT au plus tôt
  - un appel au solveur par chemin (maximal) faisable + un appel par prefixe minimal infaisable
- une solution pragmatique à PB2

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Exécution concolique

Combinaison d'exécutions symboliques et concrètes

[GKS-05] [SMA-05] [WMM-04]

Exécution concrète : collecte des infos pour aider le raisonnement symbolique

- concrétisation : force une variable symbolique à prendre sa valeur concrète courante

Deux utilisations typiques

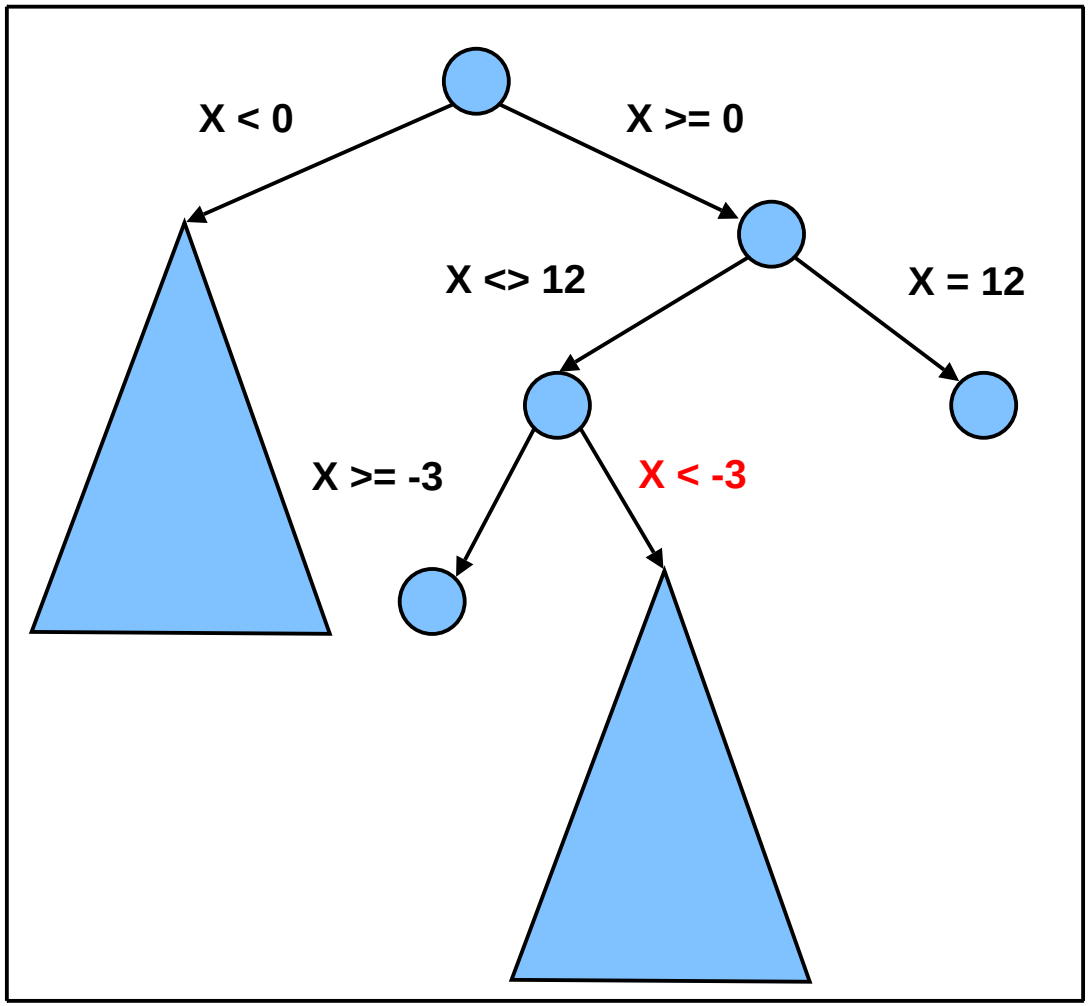
- suivre uniquement des chemins faisables à moindre coût

  toujours suivre une exécution concrète + résoudre au plus tôt
- approximation de constructions du langage "difficiles"

  concrétisation d'une partie des entrées/sorties

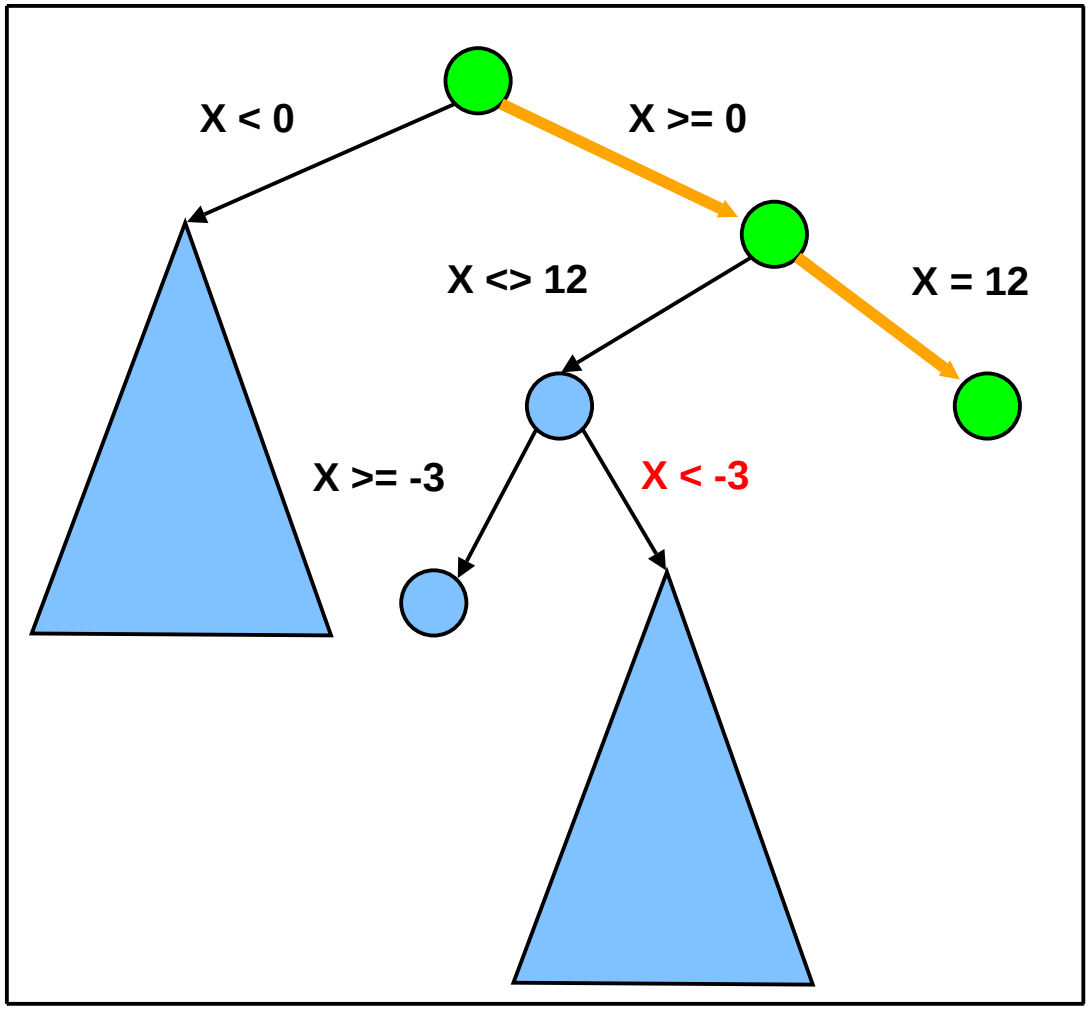
  approximations correctes

Suivre seulement des chemins faisables



concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

Suivre seulement des chemins faisables



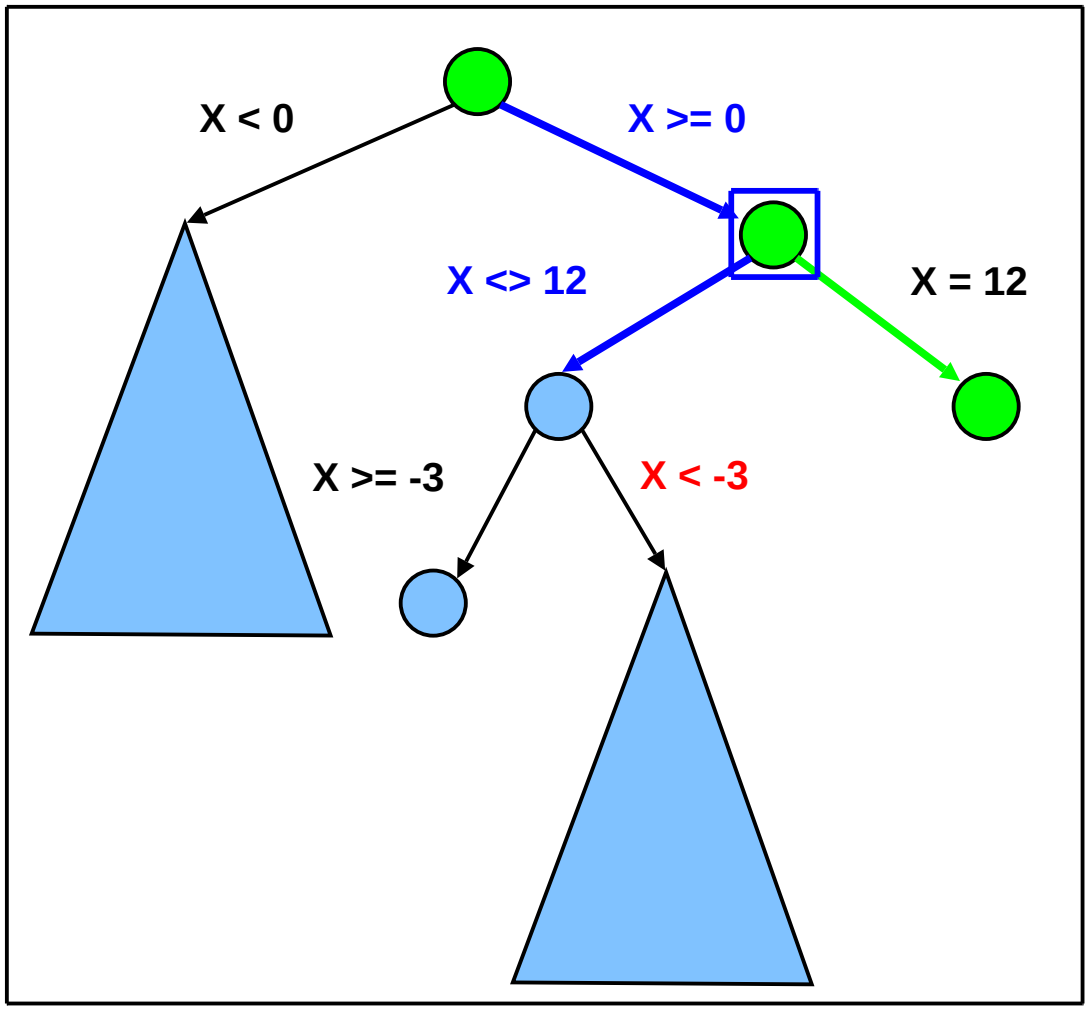
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

backtrack + résolution, unsat

Suivre seulement des chemins faisables



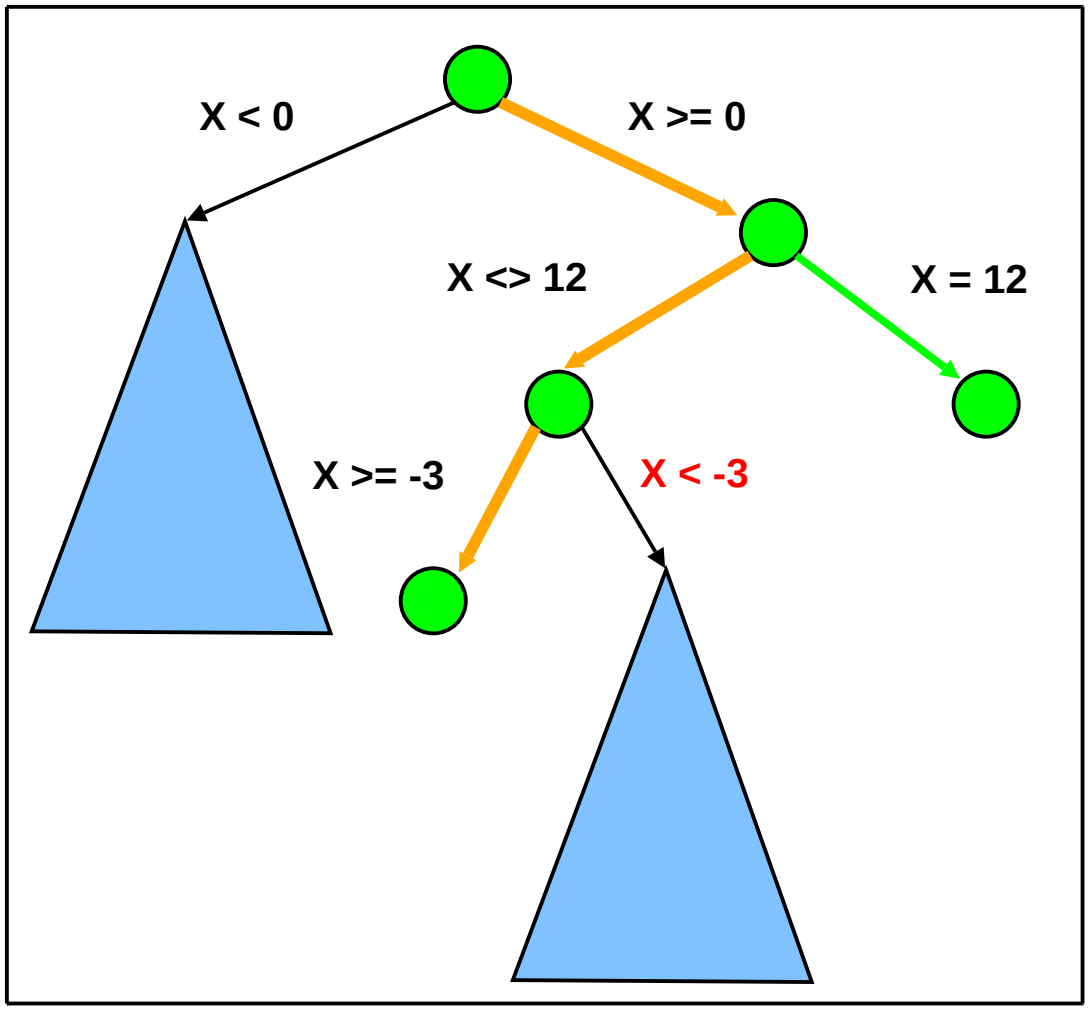
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

backtrack + résolution, unsat

Suivre seulement des chemins faisables



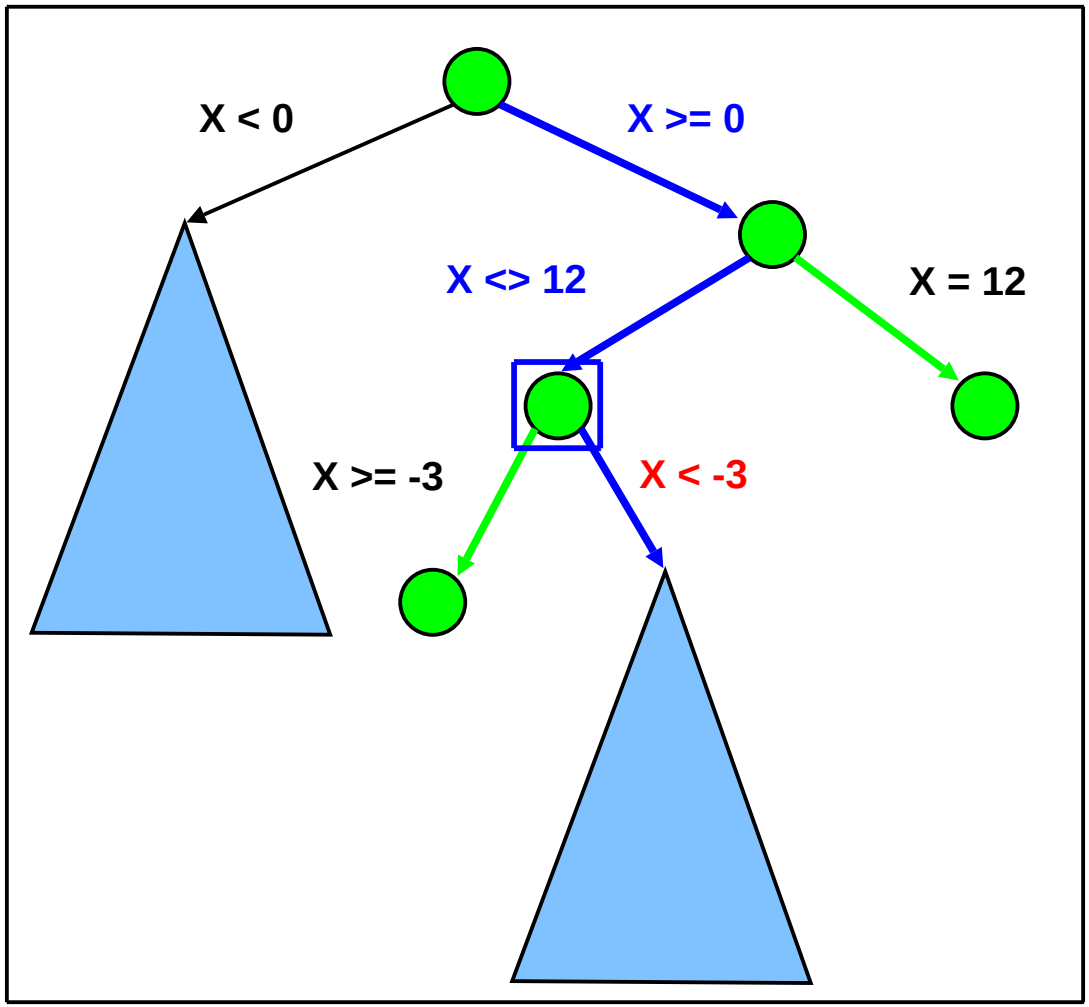
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

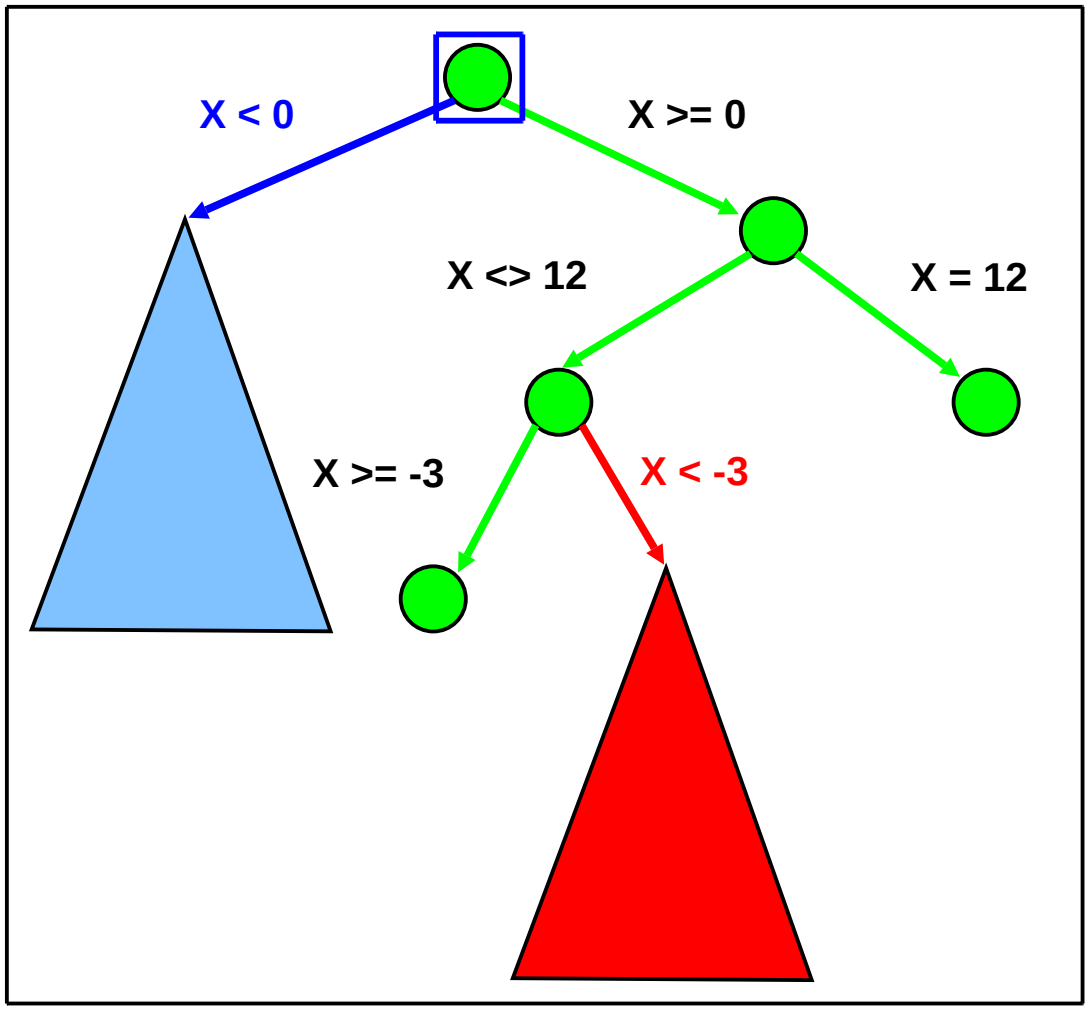
backtrack + résolution, unsat

Suivre seulement des chemins faisables



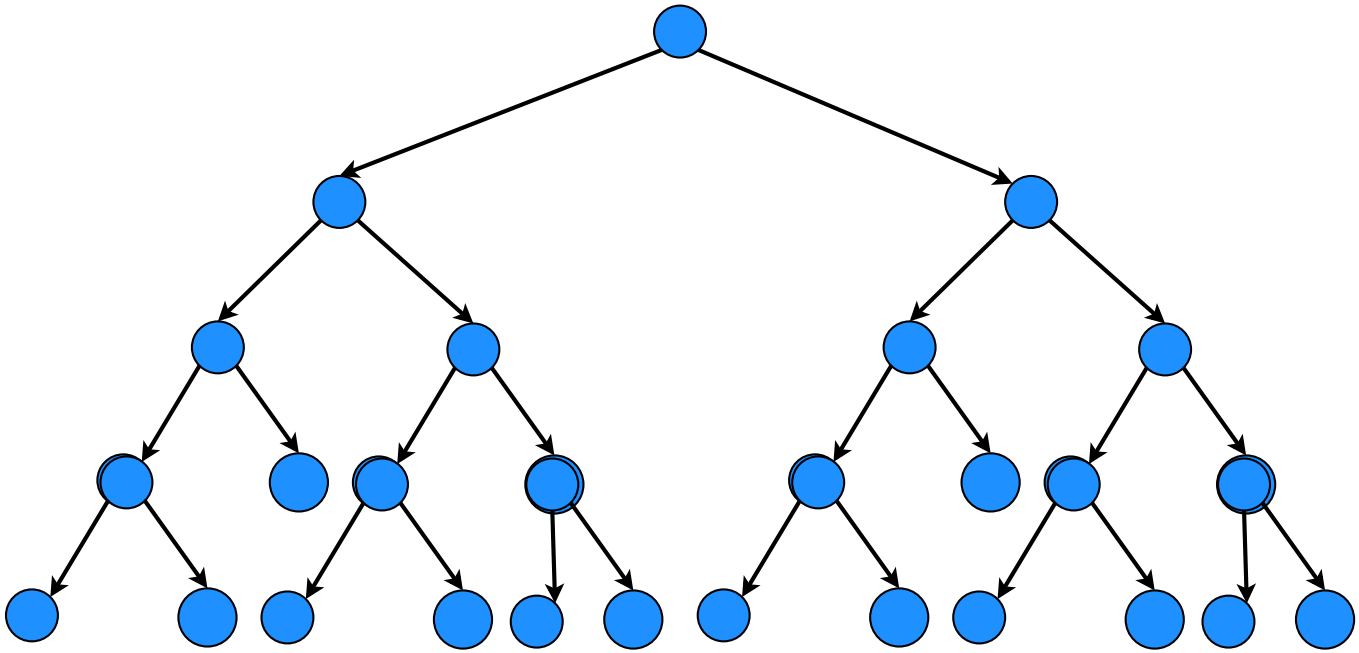
concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

Suivre seulement des chemins faisables

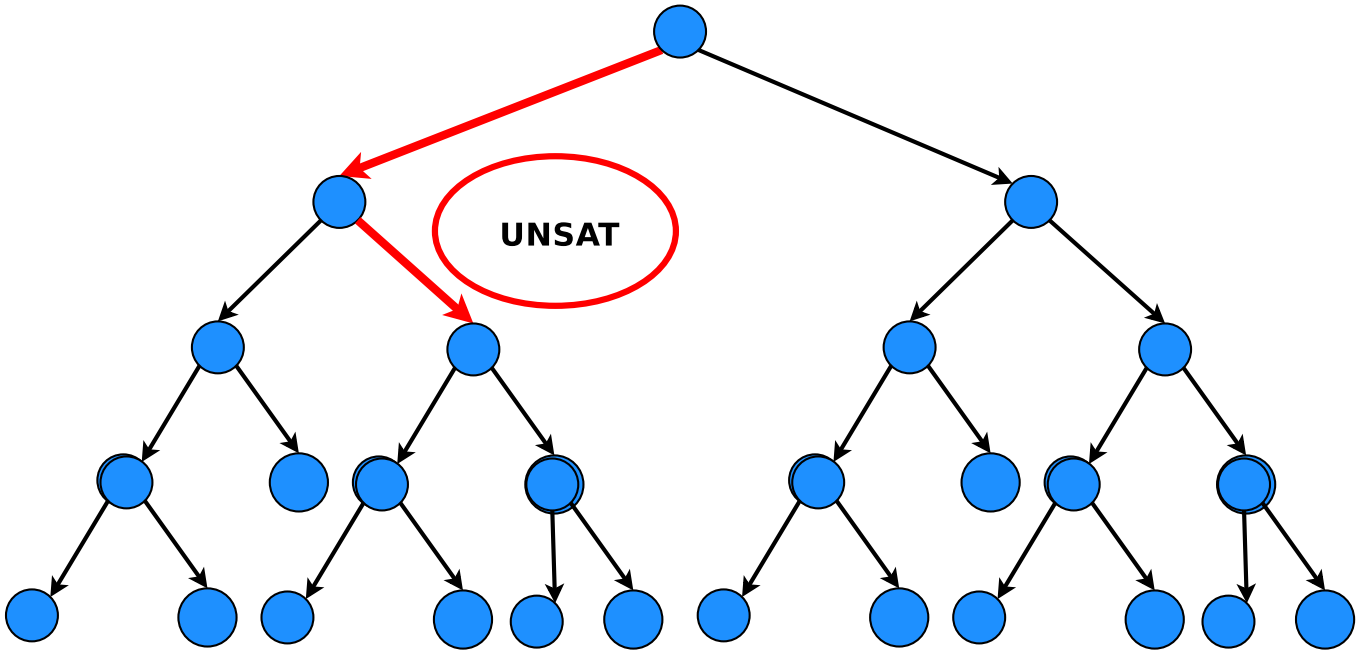


concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

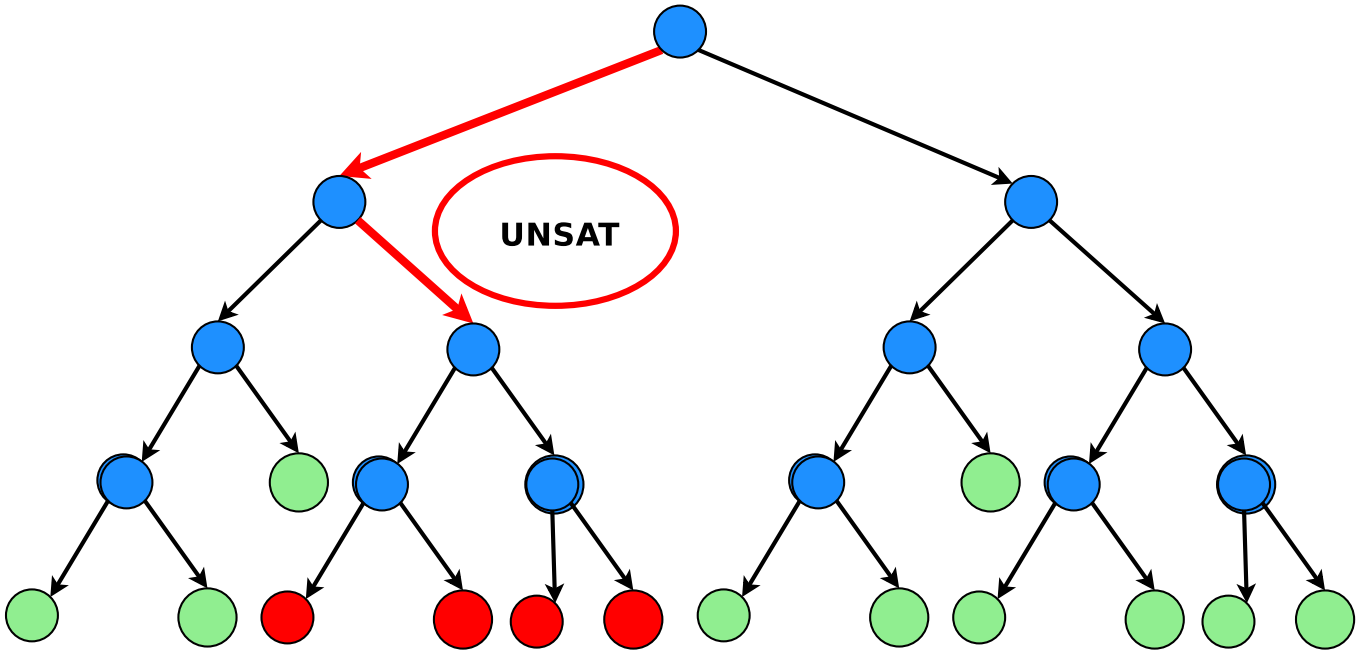
Suivre seulement des chemins faisables (2)



Suivre seulement des chemins faisables (2)



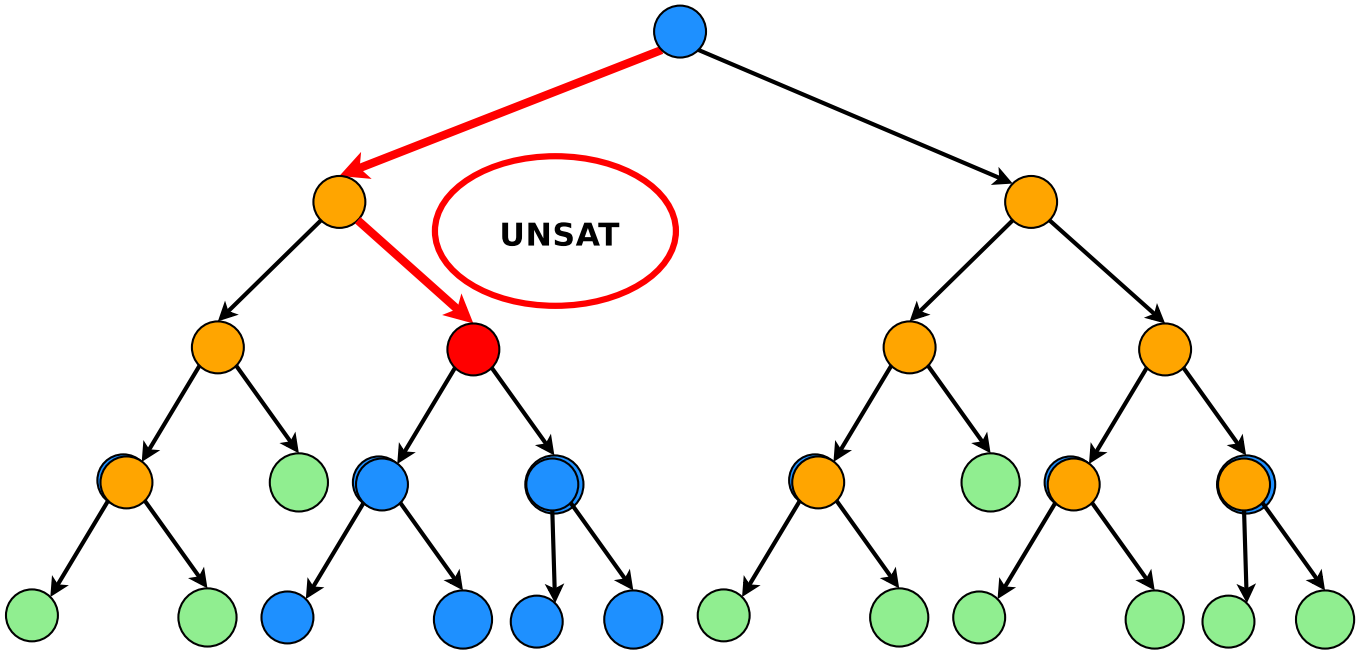
Suivre seulement des chemins faisables (2)



Méthode usuelle : résoudre le prédicat à la fin du chemin

- $+$: un appel au solveur par chemin (sur un arbre : 2^N)
- - : on peut continuer la recherche à partir de préfixes UNSAT
- KO sur programmes avec beaucoup de chemins infaisables

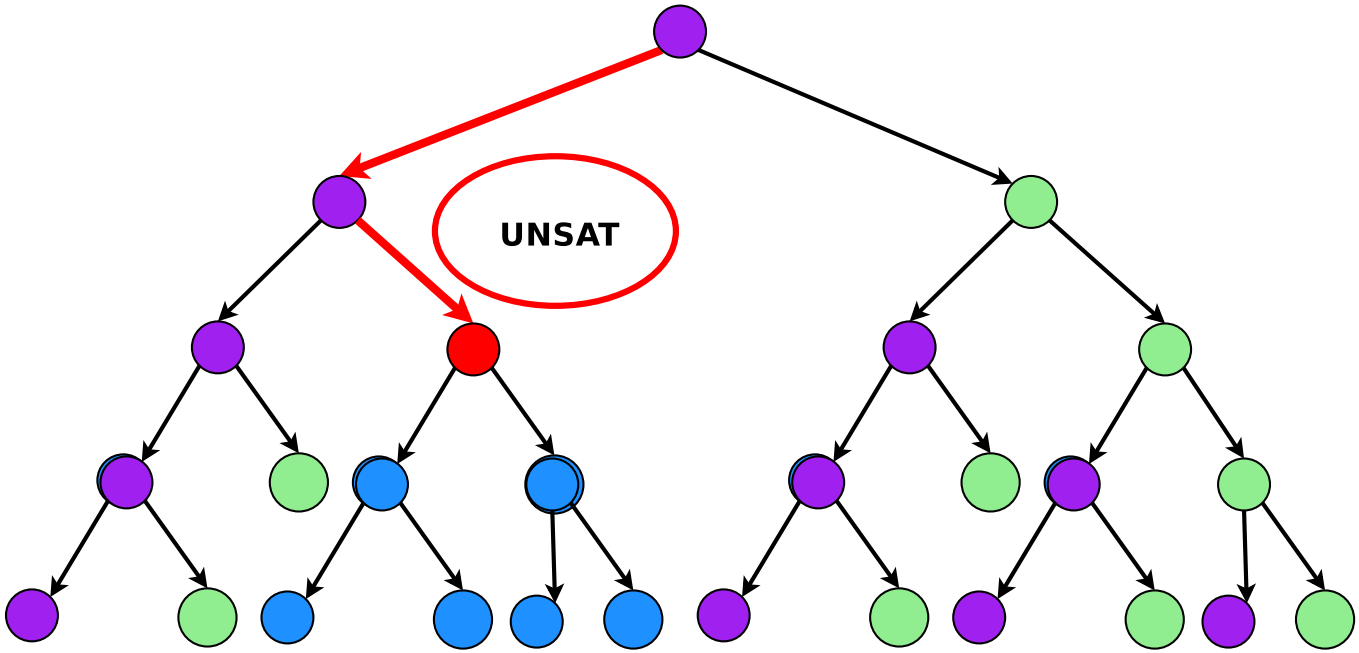
Suivre seulement des chemins faisables (2)



Alternative : résoudre le prédicat à chaque branche

- $+$: détecte UNSAT au plus tôt
- - : un appel au solveur par préfixe de chemin faisable
  et un appel au solveur par préfixe minimal infaisable
  (sur un arbre : $2 * 2^N - 1$)
- KO sur programmes avec peu de chemins infaisables

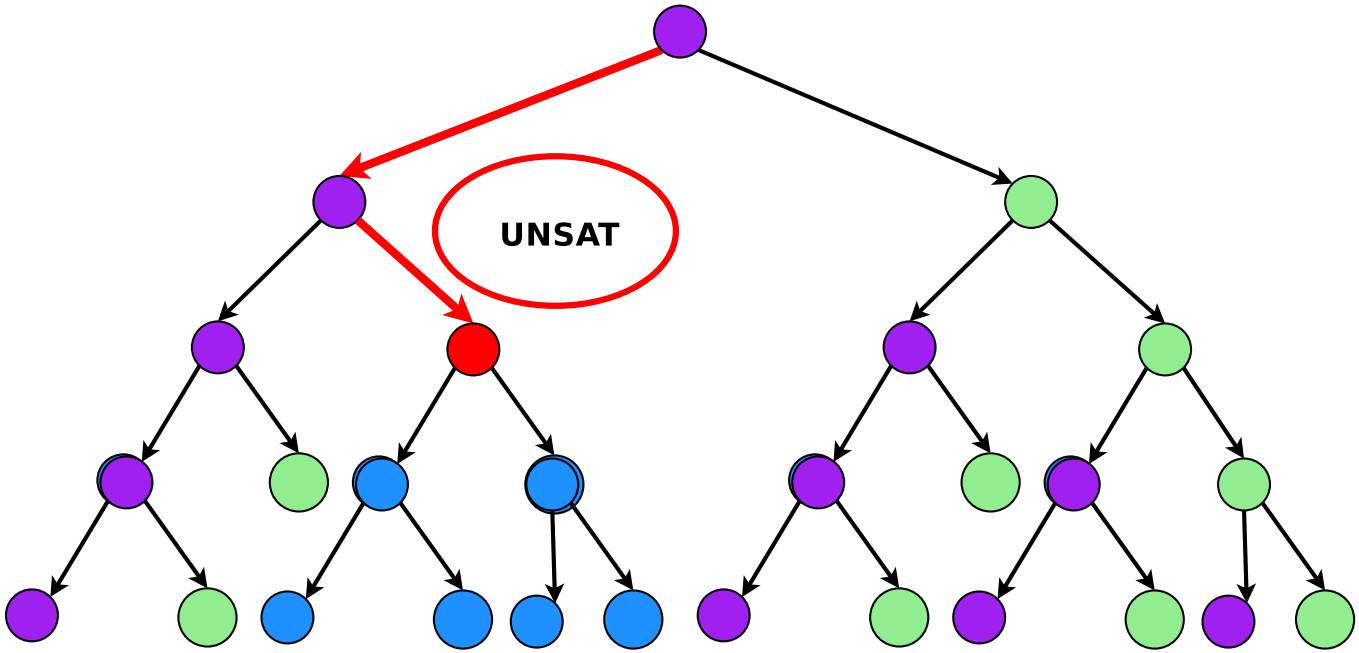
Suivre seulement des chemins faisables (2)



Un exemple possible d'exécution concolique

- (hyp : exéc. concrète suit le fils gauche)
- (noeud violet : couvert par exec concrète)

Suivre seulement des chemins faisables (2)



Un exemple possible d'exécution concolique

- + : détecte UNSAT au plus tôt
- + : un appel au solveur par chemin (maximal) faisable + un appel par prefixe minimal infaisable
- + : toujours moins d'appels que les deux autres méthodes

(Rappel) Problème 2

Générer un test pour `f` atteignant `ERROR` ci-dessous
(théorie = arithmétique linéaire)

```
g(int x) {return x*x+(x modulo 2); }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }else OK }
```

- Une exécution symbolique génère une expression symbolique de type $Z = X * X + (X \ \mathit{modulo}\ 2)$
- Cette expression n'est pas solvable en arithmétique linéaire
- On ne peut donc rien faire
- une solution : sur-approximation : $Z = \top$. Pas correcte

Exploiter l'information dynamique

```
g(int x) {return x*x+(x%2); }
f(int x, int y) {z=g(x); if (y == z) {ERROR; }}
```

Exploitation d'une exécution concrète

- première exécution avec comme entrées de `f` : $x = 3, y = 4$
- lors du calcul de prédicat, x*x+(x%2) est reconnu non traitable
- l'expression est "concrétisée" à 10, ET ses opérandes (ici x) sont aussi concrétisés.
- l'expression symbolique de z est approximée par sa valeur concrète $3 * 3 + (3\%2) = 10$
- l'exécution aboutit au prédicat de chemin $P = (Y! = 10)$ (branche else du test)
- un nouveau chemin est obtenu par négation du prédicat $not(P) = (Y == 10)$ (branche then du test)
- on résoud ce prédicat, on a toujours $x = 3$ et on fait varier y
- on trouve $x = 3, y = 10$, ce TD atteint bien `ERROR`

Approximations

Hyp 1 : code source de `f` non disponible
Hyp 2 : $\times$ pas dans la théorie logique utilisée

Instruction	Concret	Symbolique	Concolique
input(y,z)	y = 5, z = 10	Y_0, Z_0	Y_0, Z_0
c := f(z)	c = 4	$C_1 = \top$	$Z_0 = 10 \wedge C_1 = 4$
x := y * c	x = 20	$X_2 = \top$	$X_2 = Y_1 \times 4$

La procédure symbolique calcule une sur-approximation (grossière ?)

- attention à la correction

La procédure concolique calcule une sous-approximation (intelligente ?)

- attention à la complétude
- en test : correction $>$ complétude

Intérêts de la concrétisation

Le mécanisme général de concrétisation peut être utiliser de multiples manières, pour donner des sous-approximations pertinentes

- instructions du programme avec une sémantique hors de T
- instructions du programme hors scope de l'analyseur (ex : asm, sql, librairies en binaire, etc.)
- programmes avec alias et structures complexes : imposer un ensemble fini mais réaliste de relations d'alias entre variables, ou de "formes mémoires"
- multi-thread : imposer un (des) entrelacement(s) réaliste(s) des processus
- contraintes générées trop complexes dû au nombre de variables trop élevé : réduction a priori du nombre de variables via concrétisation

Intérêts de la concrétisation (2)

Le mécanisme de concrétisation est un levier très utile pour adapter la méthode sur des cas difficiles

- compromis d'utilisation
- robustesse à la classe de programmes supportés
- conserve la correction
- ex : pas besoin de gérer parfaitement toutes les constructions d'un langage pour développer une analyse concolique *correcte* pour ce langage

Procédure concolique basique

Nouvel argument : état mémoire concret C
lancement : SEARCH(node.init, ε , $\top$, 0)

procedure SEARCH(n, π, Φ, C)

```
 1: Case n of
 2: | ε → ()        /* end node */
 3: | block i → SEARCH(n.next, π · n, Φ ∧ SYMB(i), update(C,i))
 4: | goto n' → SEARCH(n', π · n,Φ, C)
 5: | ite(cond,in,tn) →
 6:       Case eval(cond,C) of     /* follow concrete branch */
 7:          | true →
 8:             SEARCH(in, π · n, Φ ∧ cond, C) ;
 9:             try /* solve new branch directly */
10:                S_p := SOLVE(Φ ∧ ¬cond) ; Tests := Tests + {(S_p, π.tn)}
11:                C' := UPDATE_C_FOR_BRANCHING(S_p)
12:                SEARCH(tn, π · n,Φ ∧ ¬cond,C')  /* branching */
13:             with unsat → ()
14:             end try
15:          | false → .....        /* symmetric case */
16:       end case
17: end case
```

Procédure concolique basique (2)

update(C : mem-conc,i : instr) $\rightarrow$ mem-conc : màj de l'état mémoire actuel

eval(cond : predicat,C : mem-conc) $\rightarrow$ bool : évalue la condition cond vis à vis de l'état mémoire actuel

update_C_for_branching(Sp : DT)$\rightarrow$ mem-conc : créer un nouvel état mémoire cohérent avec le préfixe de chemin suivi par DT

(Explication du dernier point : à chaque moment, invariant : Φ et C cohérents avec chemin suivi jusque là. Cet invariant est cassé quand on impose un branchement car l'exécution concrète ne "partait pas dans ce sens là").

Réflexions sur concolique

Récent

- déjà dans PathCrawler (CEA, 2004) pour chemins infaisables
- popularisé par DART et CUTE (2005)

Symbolic execution : $\approx$ analyses statiques sur un chemin

- plus facile car juste un chemin, mais incomplet
- garde les défauts des analyses statique pure (enfermé dans une théorie)

Concolic execution : statique + dynamique

- paradigme vraiment différent de statique pure
- grande robustesse
- compromis de mise en œuvre

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Théories utilisées (1)

Chaque instruction doit être traduite vers une formule

- le langage des formules peut être très riche

Les contraintes doivent être résolues automatiquement

- beaucoup moins de liberté ! !

Remarques

- trade-off expressivité VS décidabilité / complexité
- si théorie pas assez expressive : approximations

Théories utilisées (2)

Avantage d'être sur un chemin : QF, seulement des conjonctions

- beaucoup de classes décidables, voir solubles efficacement

Théories pour les types de base

- $\mathbb{N}$, $x - y \# k$ (logique de différence, P)
- $(\mathbb{R}, +, \times k)$ (arithmétique linéaire, P)
- $(\mathbb{N}, +, \times k)$ (arithmétique linéaire, NP-complet)
- $\mathcal{B}$ (booléens, NP-complet)
- $(\mathbb{N}_{\leq}, +, \times)$ (arithmétique bornée non linéaire, NP-complet)
- *BV* (bitvecteurs, NP-complet)
- *FLOAT* (arithmétique flottante, NP-complet)

Théories utilisées (3)

Théorie utile : fonctions non interprétées (EUF)

signature : $\langle =, \neq, x, f(x) \rangle$

axiomatique : $(FC)\ x = y \Rightarrow f(x) = f(y)$

Utilité :

- pratique pour relier des éléments entre eux de manière implicite

  `&x` en C devient $addr(X)$

  `x` une structure avec deux champs `num` et `flag` : $num(X)$ et $flag(X)$
- et aussi :
  - preprocess léger (polynomial)
  - surapproximation ($X = f(A, B)$ plutôt que $X = \top$)

Théories utilisées (4)

Théorie utile : les tableaux

signature : $\langle ARRAY, I, E, =_I, \neq_I, =_E, \neq_E, load, store \rangle$

sémantique :

- $load : ARRAY \times I \mapsto E$
- $load : ARRAY \times I \times E \mapsto ARRAY$
- axiomes : FC pour $load/store$, plus
  $(RoW1)\ i = j \Rightarrow load(store(A, i, v), j) = v$
  $(RoW2)\ i \neq j \Rightarrow load(store(A, i, v), j) = load(A, j)$

Utilité :

- tableaux bien sûrs
- mais aussi map, vectors, etc.

RMQ : dans la théorie de base, taille non contrainte

Théories utilisées (5)

expressivité $\nearrow$: moins d'échecs mais résolution sur un chemin + chère

expressivité $\searrow$: risque plus d'échecs (concrétisation), mais résolution sur un chemin - chère

---

Compromis idéal ? ?

Observation 2004-2011 : théories de + en + puissantes

Deux technologies de solveurs

- SMT : schéma très intéressant de combinaison de solveurs (Nelson-Oppen) intégré à une gestion efficace des booléens
- (plus confidentiel) Constraint Programming :
  - pour les variables à domaines finis
  - des approches intéressantes pour FLOAT, BV, $(\mathbb{N}_{\leq}, +, \times)$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 1 :

$X_{n+1} = A_n + B_n$

Bien mais ne pourra prendre en compte les pointeurs ...

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 2 : ajout d'un état mémoire M

$store(M, addr(X), load(M, addr(A)) + load(M, addr(B)))$

ok pour les pointeurs, mais on ne peut écrire au milieu de `x` (respect du typage)

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 3 : ajout d'un état mémoire M et encodage niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = A+B
in

$$store(store(store(M, addr(X), nX[0]), addr(X)+1, nX[1]), addr(X)+2, nX[2])$$

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

Traduction 3 : ajout d'un état mémoire M et encodage niveau octet (ici : 3 octets)

let tmpA = load(M,addr(A)) @ load(M,addr(A)+1) @ load(M,addr(A)+2)
and tmpB = load(M,addr(B)) @ load(M,addr(B)+1) @ load(M,addr(B)+2)
in
let nX = A+B
in

$$store(store(store(M, addr(X), nX[0]), addr(X)+1, nX[1]), addr(X)+2, nX[2])$$

ok ... mais la formule est monstrueuse

Modélisation sous-jacente des programmes C

Considérons l'instruction : `x := a + b`

PB ouvert : affiner automatiquement le niveau d'abstraction de la modélisation

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Optimisations avant l'appel au solveur (1)

Des simplifications classiques, toujours bonnes à faire

- système de cache (*)
- propagation des constantes et des égalités de variables
- unification des sous-expressions communes
- séparation de la formule en sous-formules indépendantes (**)
- solveur "léger" (preprocess) pour détecter facilement les unsat

(*) fonctionne bien si on simplifie (cf après) : beaucoup de chemins se ressemblent
(**) fonctionne particulièrement bien dans certains contextes de test (ex : parseurs)

Optimisations avant l'appel au solveur (2)

Slicing dédié : enlever toutes les contraintes qui n'affectent pas le contrôle du chemin courant

- ex : expressions de calcul du résultat final
- à faire sur la formule, ou sur l'expression de chemin (plus simple)

Réutilisation des solutions précédentes :

- on résoud les préfixes de chemin de manière incrémentale
- donc on résoud une formule du type $\phi(...) \wedge pred(A, B)$, en connaissant déjà une solution de $\phi(...)$
- on peut réutiliser l'ancienne solution comme suit : toute la sous-formule de ϕ n'affectant pas A, B est enlevée, les variables concernées prennent leurs valeurs anciennement trouvées, et on résoud ce qui reste

Optimisations avant l'appel au solveur (3)

Réutilisation des solutions précédentes : exemple

* Supposons que l' on a déjà résolu $X = Y + 3 \wedge X \leq 5 \wedge B \geq 0$

* pour résoudre $X = Y + 3 \wedge X \leq 5 \wedge B \geq 0 \wedge B + 12 \leq Z$

  on réutilise les anciennes valeurs de X, Y, et on résoud $B \geq 0 \wedge B + 12 \leq Z$

Énumération des chemins (1)

Technique usuelle : parcours en profondeur (DFS)

Avantage classique de DFS

- un seul contexte ouvert à la fois (mémoire)
- simple à implanter en récursif

Problème

- si #DT limité, se concentre sur portion restreinte du code

Énumération de chemins (2)

Si on veut couvrir les chemins, on ne peut pas optimiser cet aspect. Par contre, quand on veut couvrir les instructions ou les branches, des problèmes intéressants se posent

PB (test à budget limité) : couvrir vite un maximum d'instructions / branches

DFS pas très adaptée

Quelques solutions

- *fitness guided* (EXE, SAGE, PEX) : les branches actives sont évaluées, et celle de plus haut score est étendue
- hybride (CUTE) : DFS + aléatoire

Énumération de chemins (3)

Ingrédients de l'exécution symbolique "Fitness-guided"

- chemin actif : chemin non couvert, dont le plus long (strict) préfix est couvert
- fonction de score d'un chemin actif : $\textsc{score} : \pi \mapsto \mathit{Score}$
- un ordre total sur *Score*
- à chaque étape :
  - choisir le chemin actif ayant le meilleur score
  - "étendre" ce chemin (résolution, exécution)
  - si réussit, ajouter les nouveaux chemins actifs créés cette exécution

Énumération de chemins (4)

Par exemple, on peut baser le score sur :

- longueur du chemin, profondeur d'appel de la dernière instruction
- nb de fois où la dernière instruction a été couverte
- ...

Intérêts : permet d'intégrer facilement de nombreuses heuristiques de parcours de chemin

- chemin choisi aléatoirement
- dfs, bfs, dfs avec seuil
- dfs modulée par la profondeur d'appel et priorité aux branches non couvertes
- ...

Optimiser l'énumération de chemins (1)

PB : certains chemins sont redondants pour le critère de couverture choisi

Couper les chemins qui ne peuvent atteindre de nouvelles instructions (Osmose, Exe)

- à chaque étape, calcul des instructions accessibles (analyse statique globale)
- peut être fait très efficacement
- technique complète vis à vis de la couverture d'instructions

Couper les chemins amenant à un état symbolique déjà couvert, c'est à dire que l'on a deux préfixes π et σ tq $\phi_\pi \Rightarrow \phi_\sigma$: on garde seulement (σ, ϕ_σ)

- technique complète vis à vis de la couverture d'instructions
- potentiellement très coûteuse, demande de vérifer $\Rightarrow$

Optimiser l'énumération de chemins (2)

PB : explosion du #chemins due aux appels de fonction

Quelques solutions (reste un problème ouvert)

- couper l'exploration à une certaine profondeur (OSMOSE, JAVA PATHFINDER)
- gestion paresseuse des fonctions (SAGE)
- construction itérative de résumés de fonctions (DART)
- spécifications de fonctions (PATHCRAWLER)

Plan

- Contexte
- Exécution symbolique : bases
- Exécution concolique : bases
- Niveau avancé
- Discussion

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Questions pratiques (1)

Prise en compte des préconditions de la fonction à tester

Pourquoi : éviter des tests non pertinents

- ex : algorithme de recherche dichotomique : tableau trié

Quel format pour la précondition ?

- formule logique : les préconditions élaborées (ex : tableau trié) demandent des quantificateurs
- code : risque d'augmenter le # chemins

Questions pratiques (2)

Prise en compte de l'oracle de la fonction à tester

Pourquoi : tests plus informatifs, diriger les tests vers les bugs

A quel niveau intervient l'oracle ?

- a posteriori, à titre indicatif
- dans le process de génération

Quel format pour l'oracle ? même pb que pour la précondition

- ici des oracles partiels simples (runtime errors) peuvent être légers et intéressants du point de vue guidage de la génération

Questions pratiques (3)

Sortir les tests dans un format réutilisable

- exporter vers autres outils (JUnit, sélection / minimisation, etc.)
- utilisation conjointe de DT issues d'autres méthodes / outils

Génération en complément de tests existants

- éviter redondance avec tests existants
- génération incrémentale

Quelques prototypes existants

PathCrawler (CEA)	2004
Dart (Bell Labs), Cute (Berkeley)	2005
Exe (Stanford)	2006
Java PathFinder (NASA)	2007
Osmose (CEA), Sage (Microsoft), Pex (Microsoft)	2008

Quelques résultats pratiques

PEX livré dans Visual C#

- cible = aide au programmeur

SAGE en production interne chez Microsoft (sécurité)

- service interne de "smart fuzzing"
- le logiciel tourne en boucle sur de gros serveurs
- nombreux bugs trouvés

Études de cas académiques sur des codes type drivers / kernel (Linux, BSD, Microsoft .NET)

- codes souvent déjà bien testés, nombreux bugs trouvés
- ex : Klee : $>$ 95 % de couverture obtenue automatiquement sur les Unix coreutils (make, grep, wc, etc.)

Démo

PathCrawler online

http ://pathcrawler-online.com/

Plan

- Prédicat de chemins
- Exécution symbolique
- Exécution concolique
- Aspects logiques
- Optimisations
- En pratique
- Discussion

Bilan (subjectif) sur l'approche concolique

Points forts pour une utilisation industrielle

- totalement automatisée si oracle automatique
- robuste aux "vrais" programmes
- correcte, résultats facilement vérifiables
- s'insère dans pratiques de test existantes (process, outils, etc.)
- utilisation (naturellement) incrémentale
- gain incrémental
- surpasse assez facilement les pratiques usuelles (ex : fuzzbox testing)

Puissance maximale si couplée avec un langage de contrat

Quels domaines d'utilisation ?

- pb pour la certification : traçabilite DT - exigences
- pb si l'on veut absolument 100% de couverture, même sur des codes de taille petite / moyenne
- mais ok pour le débogage intensif

Quelques challenges

Passage à l'échelle (# chemins)

- quelle notion de résumé de fonction / boucle ?
- comment "bouchonner" facilement un morceau de code ?

Prise en compte de préconditions complexes en cas de structures dynamiques

- quantificateurs, axiomes

Au niveau solveurs :

- chaines de caractères, flottants

Opinion : méthodes formelles

Critères de classification usuels des méthodes formelles

- méthodes dynamiques / statiques
- sous approximation / sur approximation
- méthodes complètement automatiques ou avec annotations
- modèle / code

Ces critères sont utiles des points de vue historique et pédagogique, mais de plus en plus d'approches récentes combinent des aspects jusque là considérés antagonistes.

- démarche pragmatique vis à vis de problèmes très difficiles
- l'exécution concolique est une technique pionnière de ce point de vue

Plan

- Introduction
- Automatisation de la génération de tests
- Critères de test avancés

Problème général

Critère mutationnel :

- critère de test puissant en terme de détection de fautes
- difficile à automatiser

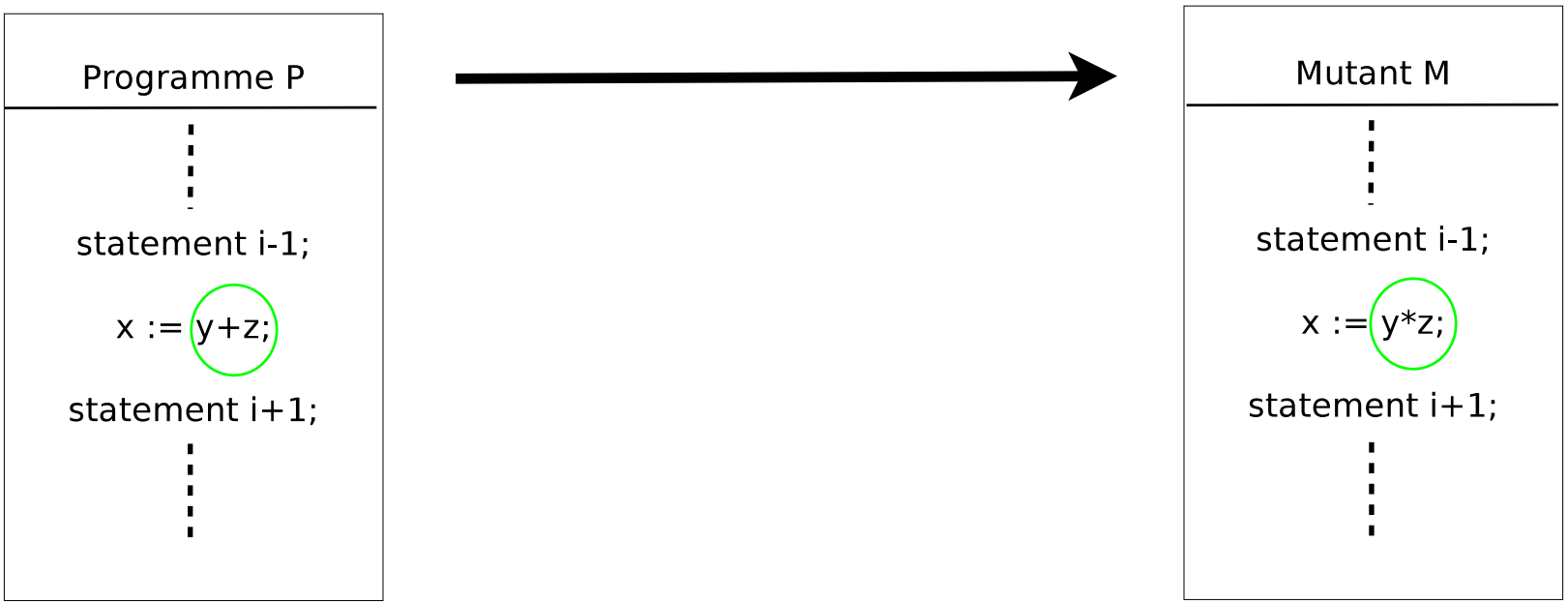
Critères de couverture usuels (instructions, branches) :

- critères moins puissants
- permettent de manière efficace :
  - le calcul de couverture
  - la génération de tests (exécution concolique)

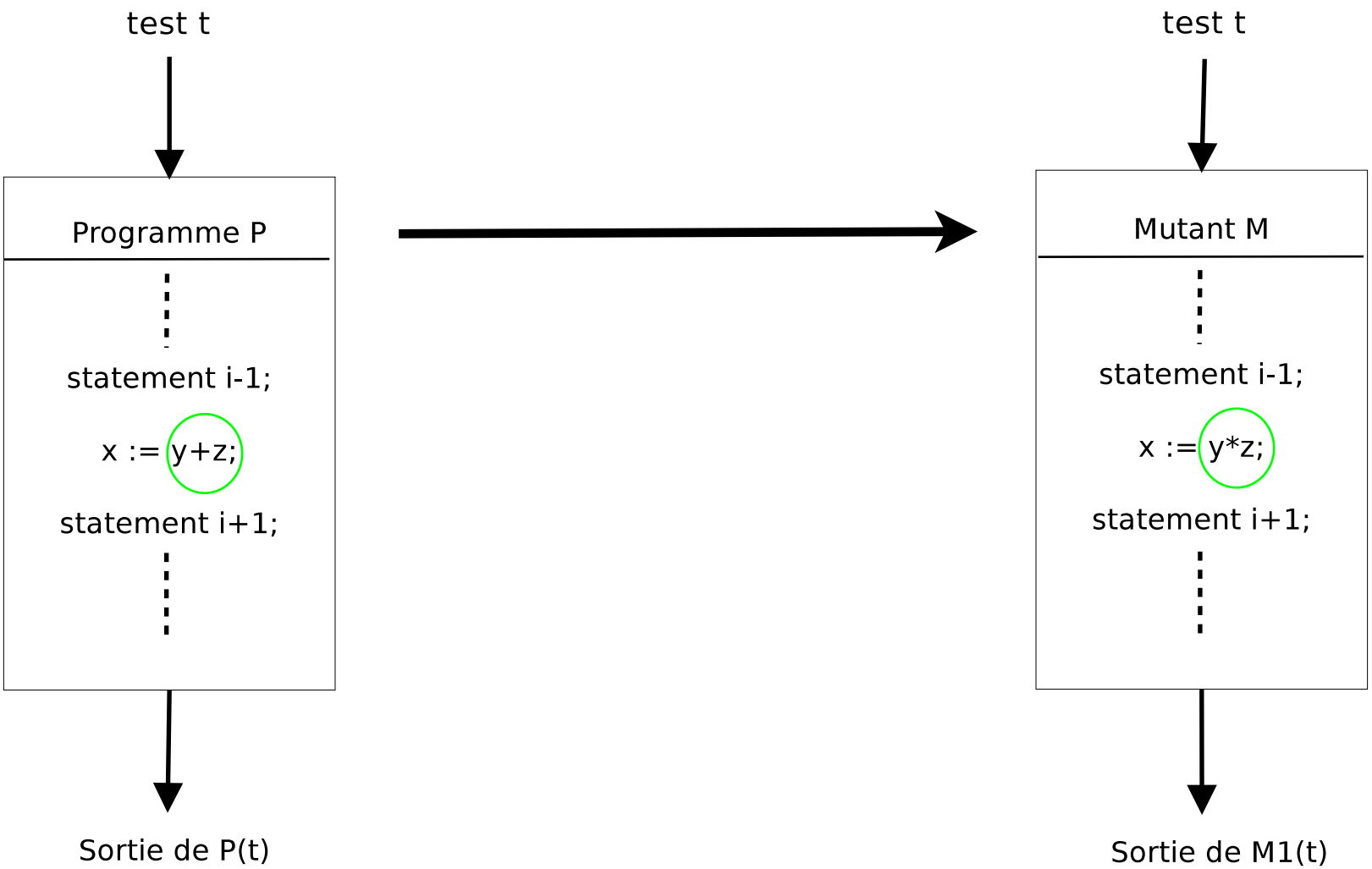
Mutants et labels

- Idée :
  - Transformer les mutants (faibles) en prédicats/labels dans le programme
- Pourquoi :
  - Bénéficier des avantages des deux familles de critères
  - Automatiser efficacement la couverture de mutants
  - Étendre l'exécution concolique à des critères de test avancés

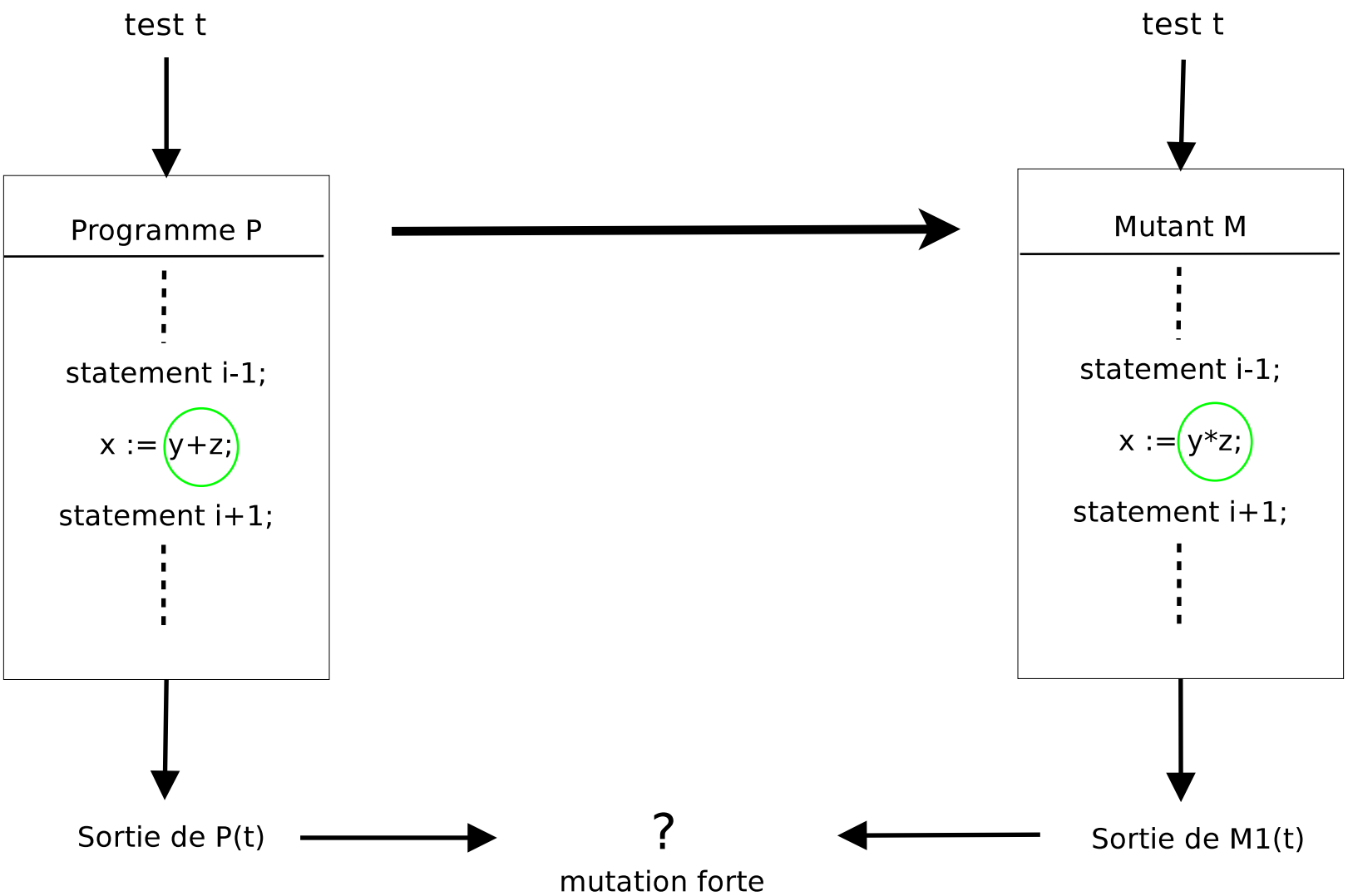
Mutations fortes



Mutations fortes



Mutations fortes



Rappel sur les mutants

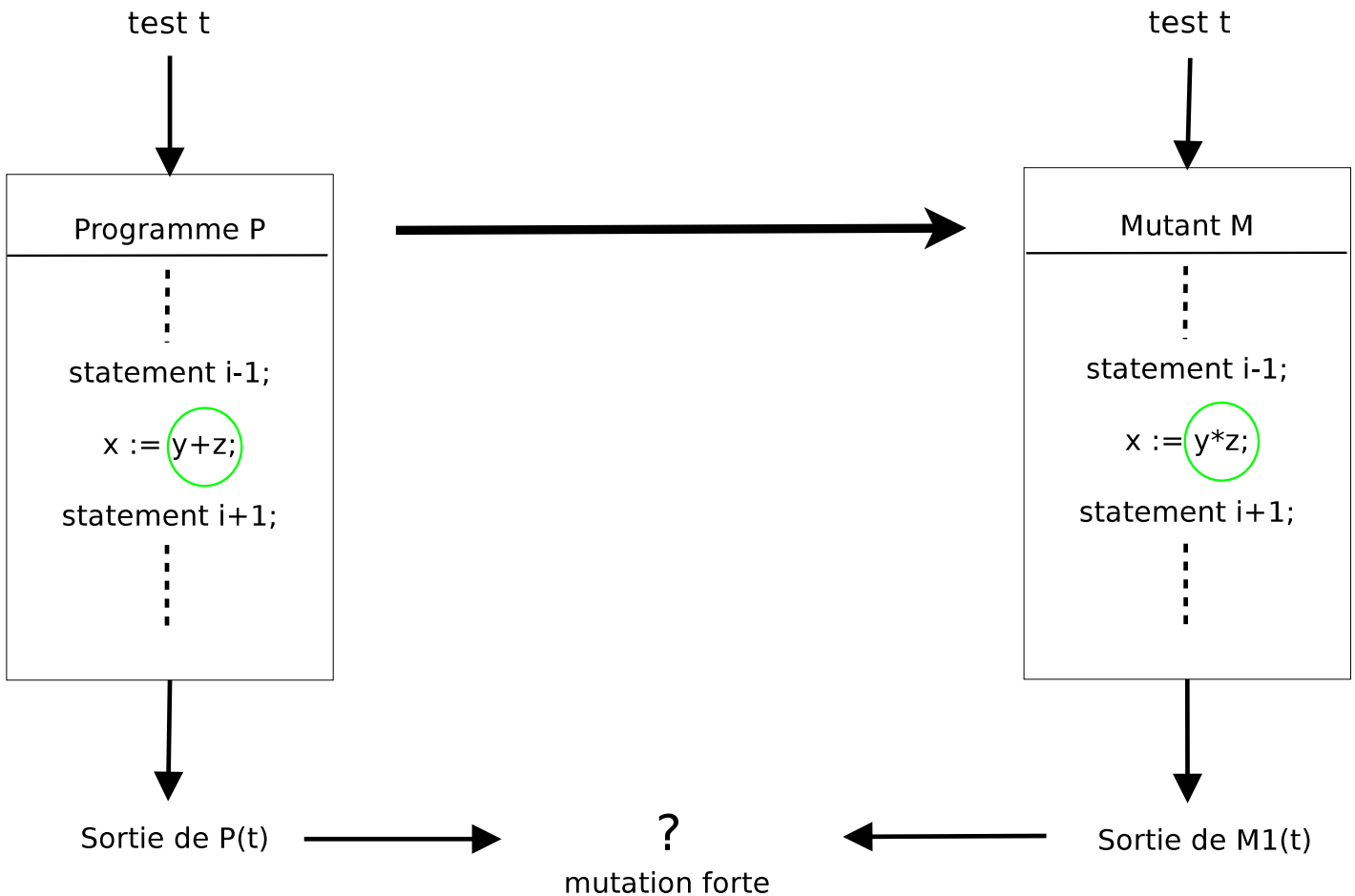
Intérêt

- critère de couverture le plus puissant du point de vue théorique (peut émuler la plupart des autres)
- bien corrélé en pratique à la découverte de bugs

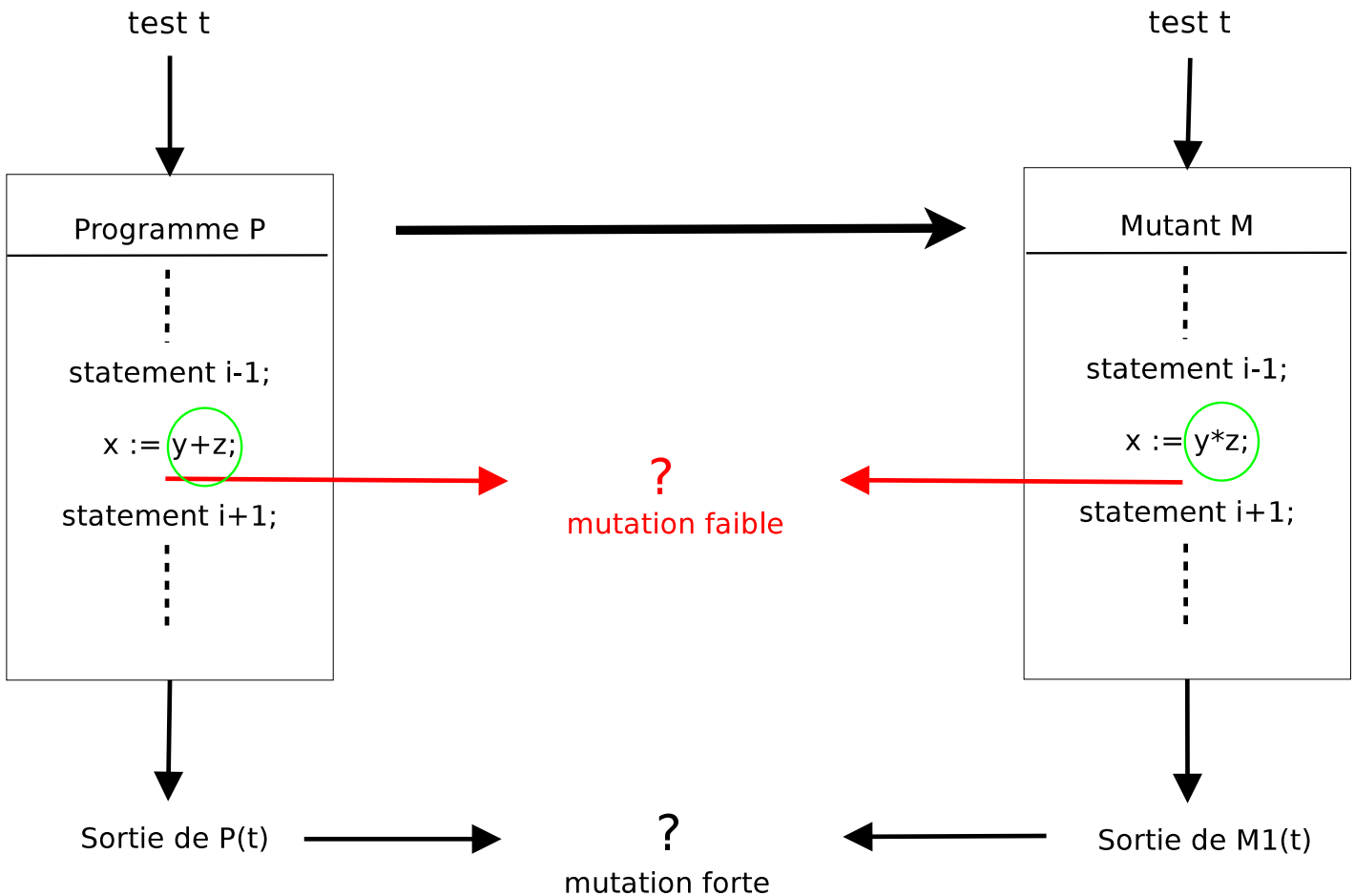
Difficile à automatiser

- calcul de couverture : M compilations, T $\times$ M exécutions (rmq : M souvent très grand)
- génération de TD : inexistant

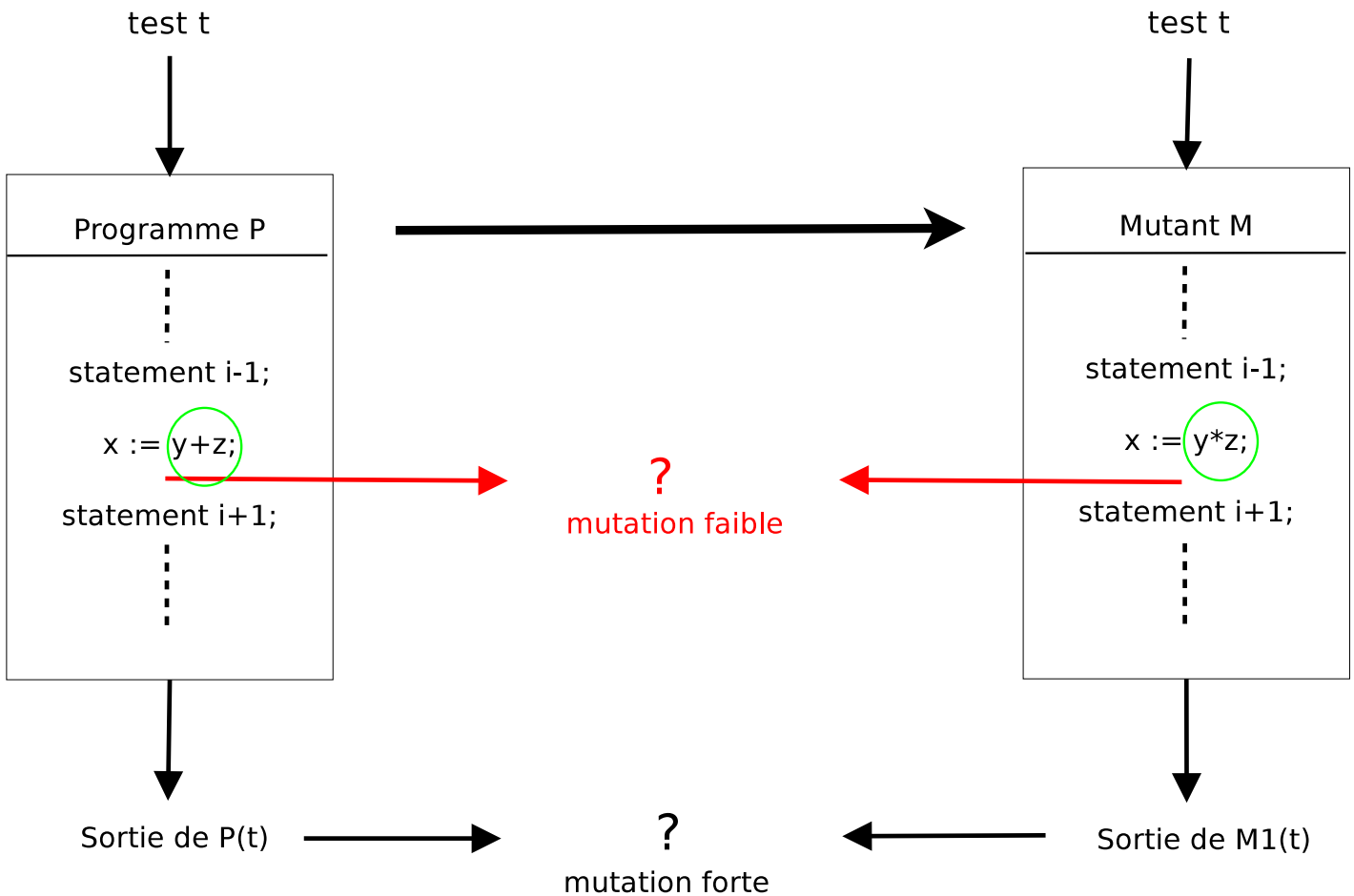
Mutations faibles



Mutations faibles

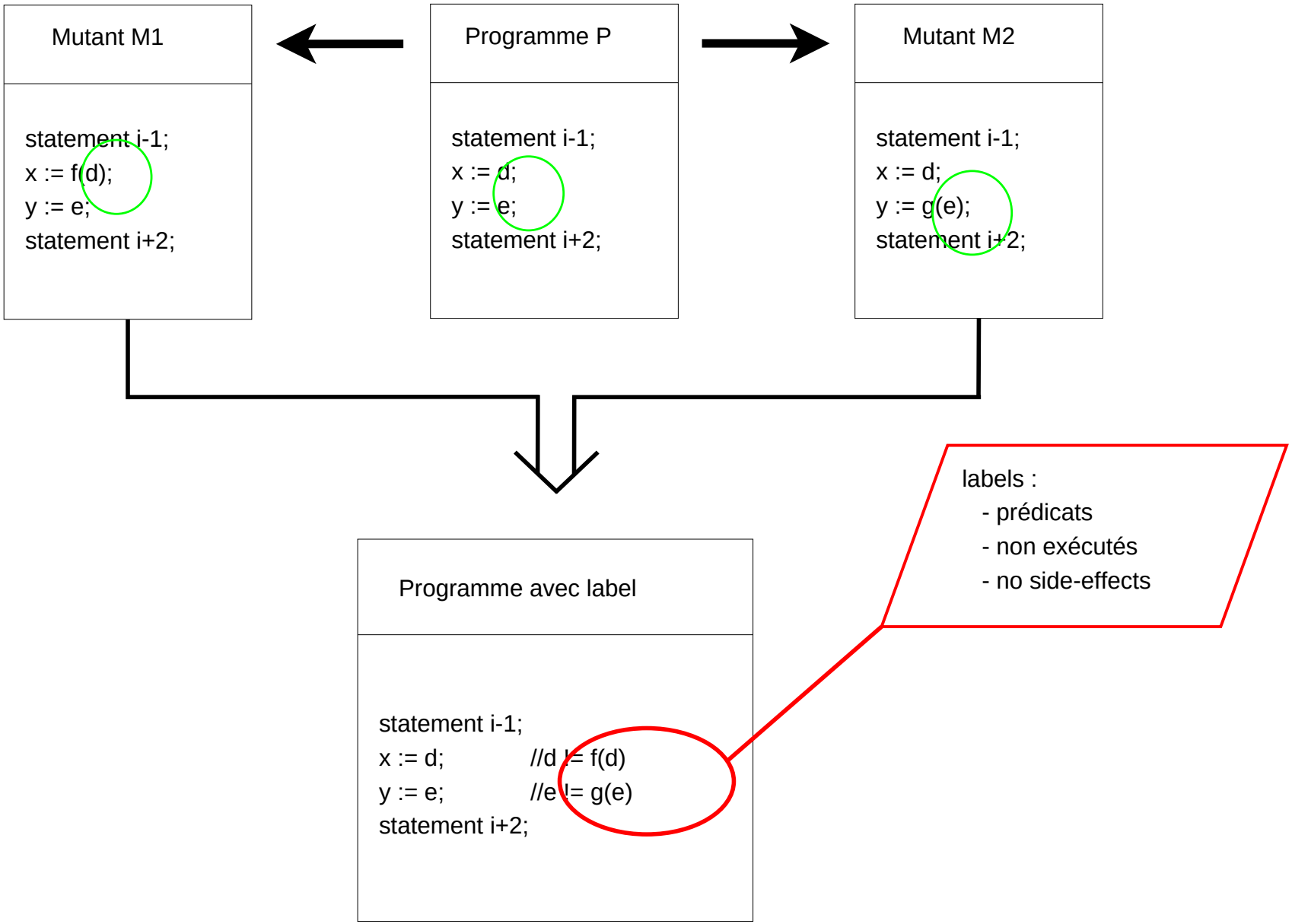


Mutations faibles



Mutation faible : presqu'aussi puissant (en pratique) que mutation forte

Des mutants faibles aux labels



Analogie entre mutants faibles et labels

Correspondance forte des critères :

Mutation faible	$\longleftrightarrow$	Label
Mutant tué de manière faible	$\longleftrightarrow$	Label couvert
Score de mutation faible	$\longleftrightarrow$	Taux de couverture des labels
Mutants équivalents	$\longleftrightarrow$	Labels non couvrables

MAIS : labels plus facilement automatisables :

- réutilisation d'outils de vérification
- automatisation efficace

Mise en œuvre

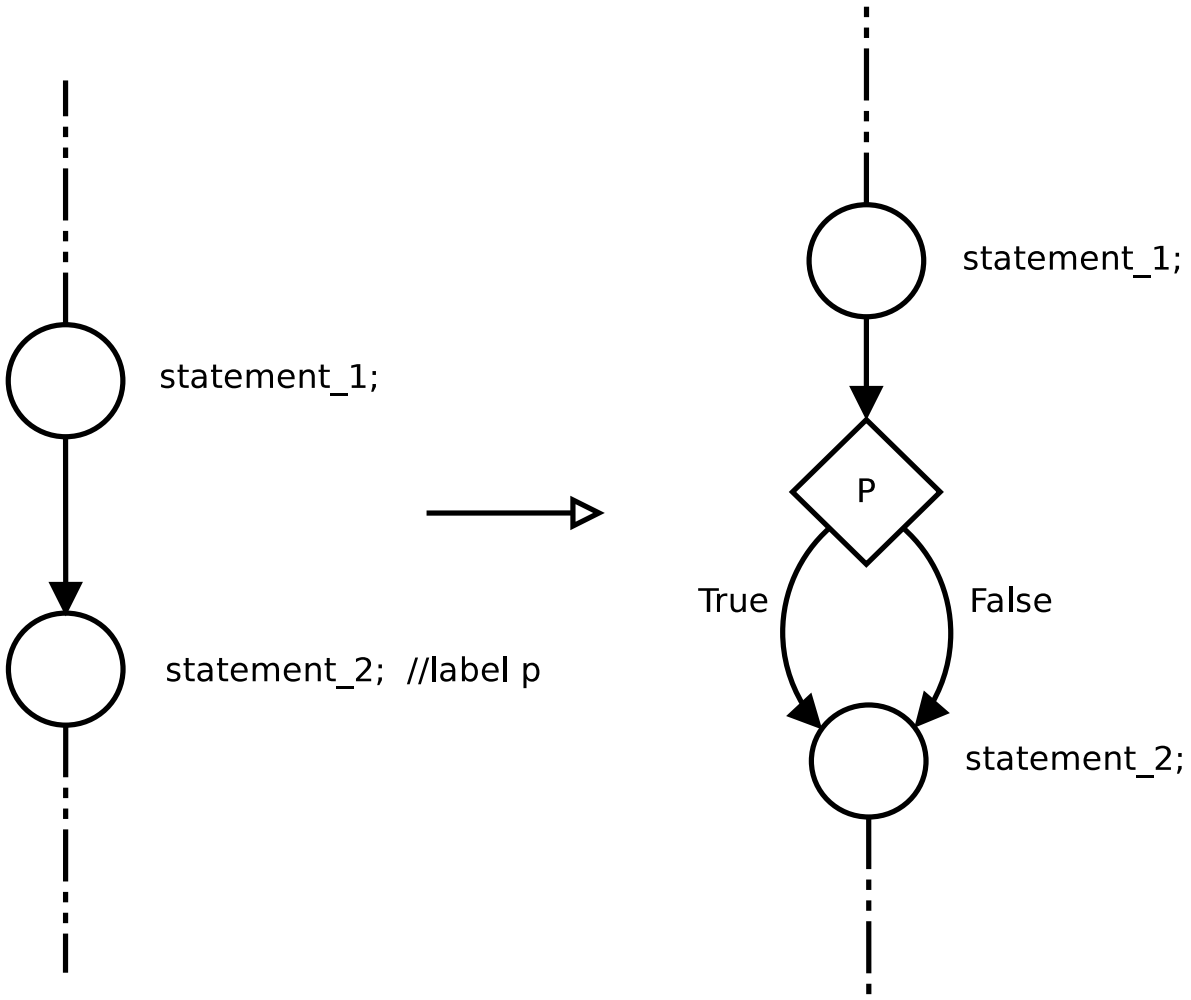
Applications visées :

- couverture
- génération de tests

Méthode

- utilisation d'outils en boîte noire :
  - Emma : couverture
  - PathCrawler : génération de tests
- instrumentation du code pour "simuler" les labels

Instrumentation naïve



couverture du label p $\equiv$ couverture de la branche `True`

Calcul de couverture

temps de calcul
de couverture des labels
(avec Emma)

VS

temps de calcul
de couverture des mutants
(avec MuJava)

↓

M tests / 1 programme

↓

M tests / N programmes

Résultats

Pour un jeu de 100 tests

		Emma		MuJava
		programme initial	programme avec labels	mutants
TCas	124 LOC 111 labels	0,03 s	0,03 s	5 s
Replace	436 LOC 607 labels	0,05 s	0,10 s	40 s
Jtopas	5400 LOC 610 labels	3,22 s	11,72 s	1 400 s

Gain important (facteur 100) et surcoût raisonnable (facteur 4)

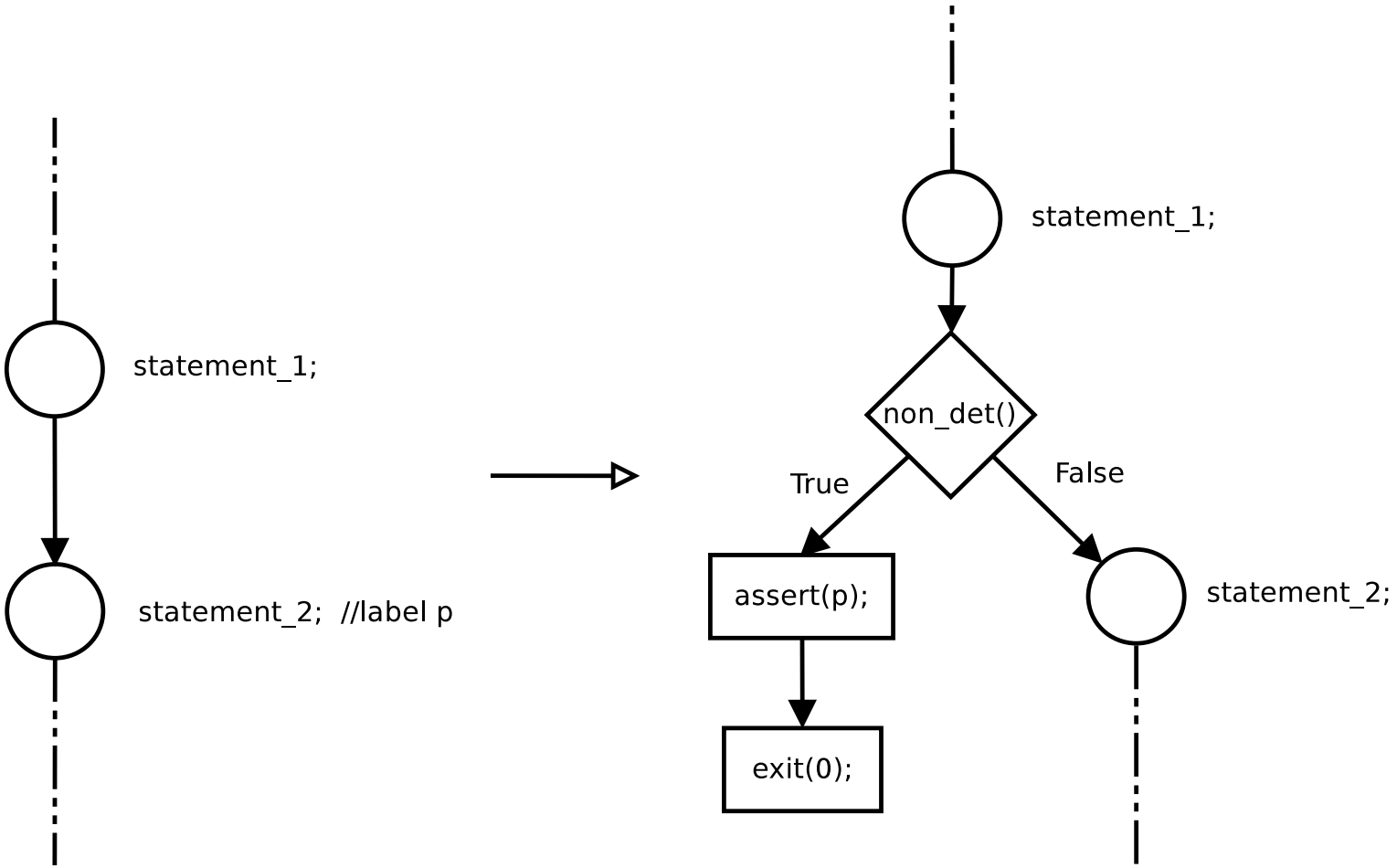
Génération de tests

L'instrumentation naïve ne fonctionne pas pour la génération de tests :

- nombre exponentiel de chemins (cf ci après)
- les chemins ajoutés sont complexes

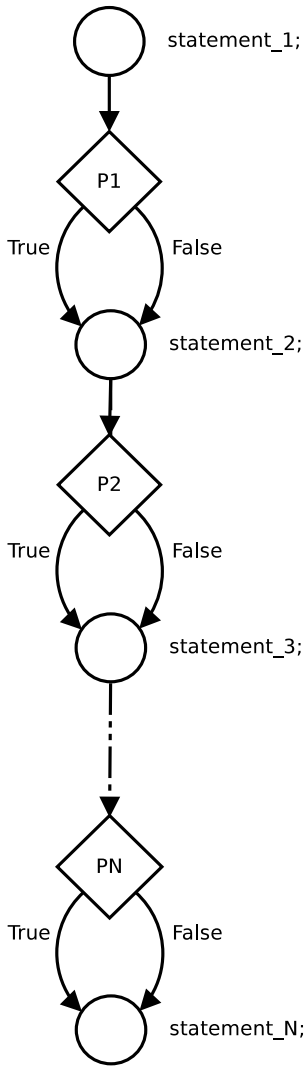
Idée I : instrumentation plus fine

Chaque chemin d'exécution contient au plus un label



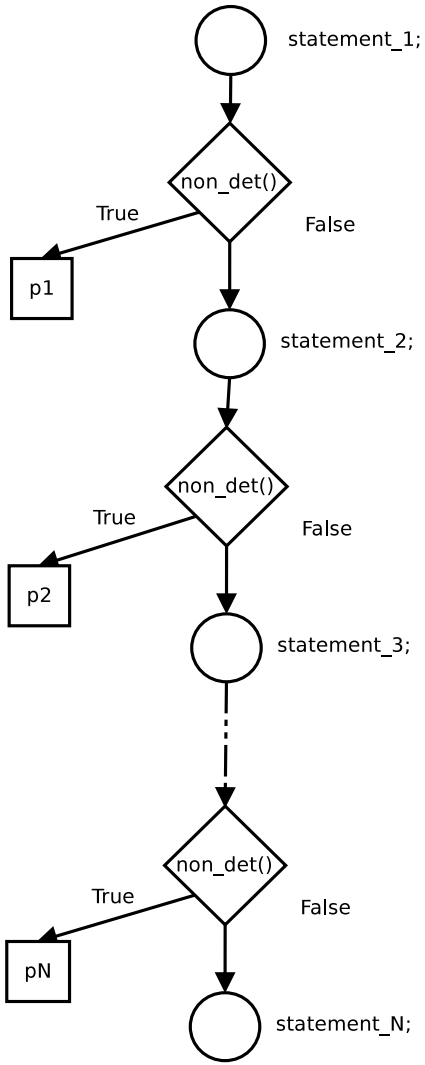
Comparaison des 2 instrumentations

Instrumentation naïve



2^N chemins

Instrumentation fine



N+1 chemins

Idée II : Utilisation incrémentale de PathCrawler

- Partitionner l'ensemble des labels du programme P
- Lancer PathCrawler (PC) successivement sur P muni d'une des partitions
- Entre chaque exécution de PC, élaguer les labels couverts lors des générations lancées sur les partitions précédentes

Résultats

		PC	PC_i	PC_i^{++}
Trityp	50 LOC 141 labels	91% 0 s 14 TC	100% 466 s 63 TC	100% 1 s 84 TC
Replace	100 LOC 79 labels	98% 2 s 121 TC	98% 1 745 s 275 TC	100% 50 s 393 TC
TCas	124 LOC 111 labels	94% 4 s 164 TC	96% 228 767 s 249 TC	100% 72 s 1 049 TC

PC_i (naif) : temps d'exécution trop long $\Rightarrow$ couverture non-maximale

PC_i^{++} : couverture maximale + temps raisonnable

PC (no instrumentation) : très rapide + bonne couverture : méthode hybride ?

Conclusion

L'utilisation des labels permet une automatisation efficace du test de mutations via des techniques concoliques :

- critère de couverture fort (mutants faibles)
- temps raisonnable
- réutilisation de techniques classiques

Une gestion native des labels dans l'algorithme d'exécution concolique devrait permettre d'améliorer encore les performances

- objectif : surcoût de 3x-4x par rapport à la couverture d'instructions