

Automatisation du Test Logiciel

Génération de tests par exécution symbolique

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

`sebastien.bardin@cea.fr`

`http://sebastien.bardin.free.fr`

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 $\varphi = \text{DT}$ 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

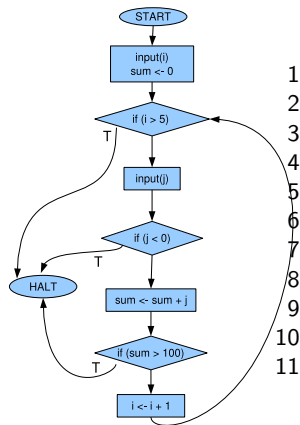
- théories plus simples : sans quantificateur, juste conjonction
- mais énumération de chemins
- nous verrons deux techniques
 - ▶ exécution symbolique
 - ▶ exécution symbolique dynamique (dite aussi exécution concolique)

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

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
```

- 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.)
 - ex : $D = \mathbb{N} \times \mathbb{N}$ pour `void foo(int a, int b)`
- $V \in D$ une entrée du programme
- On note $P(V)$ la trace d'exécution de P lancé sur la donnée d'entrée V

Notion centrale : le prédicat de chemin (2)

- On se donne une théorie logique T
 - théorie : ensemble d'opérateurs / prédicats permis, et leurs axiomes
 - ex : arithmétique linéaire

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 (dans T) 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 $\pi \preceq P(V)$.

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

- exécution concrète : m à j des valeurs des variables
- exécution symbolique : m à j 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 (4)

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

Prédicat de chemin (entrées Y_0 et Z_0)

$\top$

Prédicat de chemin (4)

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 (4)

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 (4)

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 (4)

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 (4)

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 $Y_0 + 4 < 2 \times Z_0 \wedge Y_0 + 4 \geq Z_0$

Compléments sur le prédicat de chemin

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 $x \in D$.

$\llbracket \varphi \rrbracket$ 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 $\llbracket \varphi \rrbracket = wpre(t, \llbracket \varphi' \rrbracket)$

$post(X, t)$: 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 $\llbracket \varphi' \rrbracket = post(\llbracket \varphi \rrbracket, t)$

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

Alors

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

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

prédicat de chemin de π calculé en avant :

$$\bar{\varphi}_{\pi}' = \text{post}(\text{post}(\text{post}(\top, t_1), t_2) \dots, t_n)$$

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

$$\blacksquare \quad \bar{\varphi}_{\pi}' \Leftrightarrow \bar{\varphi}_{\pi}$$

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

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 , $\top$)

/ m à j 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: | halt  $\rightarrow$  /* end node */
3:   try  $S_p := \text{SOLVE}(\Phi)$ ; Tests := Tests +  $\{(S_p, \pi)\}$  /* new TD */
4:   with unsat  $\rightarrow$  ();
5:   end try
6: | block i  $\rightarrow$  SEARCH(node.next,  $\pi \cdot \text{node}$ ,  $\Phi \wedge \text{SYMB}(i)$ )
7: | goto tnode  $\rightarrow$  SEARCH(tnode,  $\pi \cdot \text{node}$ ,  $\Phi$ )
8: | ite(cond, inode, tnode)  $\rightarrow$  /* branching */
9:   SEARCH(inode,  $\pi \cdot \text{node}$ ,  $\Phi \wedge \text{SYMB}(\text{COND})$ );
10:  SEARCH(tnode,  $\pi \cdot \text{node}$ ,  $\Phi \wedge \neg \text{SYMB}(\text{COND})$ )
11: end case
```

Procédure SYMB : $\text{Instr} \mapsto T$

- transforme une instruction de base en formule de la théorie T
- 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 SOLVE : $T \mapsto \{SAT(Sol), UNSAT\}$

- procédure de décision pour la théorie T
- retourne SAT (+ une solution) ou UNSAT

Procédure SYMB, exemple

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

Expressions de la théorie logique T définies par

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

let SYMB $e = \text{match } e \text{ with}$

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

SYMB définit de manière similaire sur les conditions

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 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

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

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

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 principaux de la méthode : théorie logique, énumération de chemins, critère d'arrêt

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

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

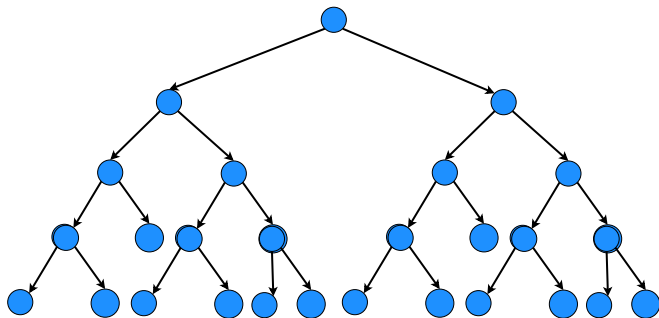
Exploration (inutile) de chemins infaisables (PB1)

- 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)

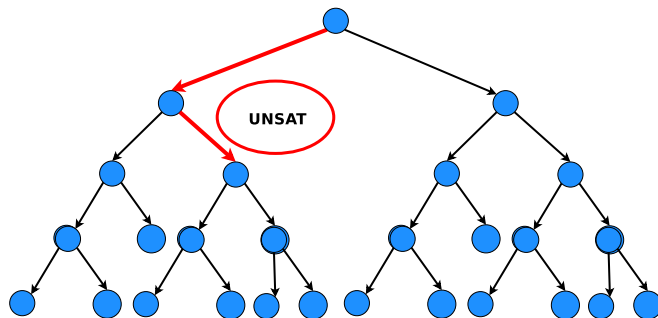
- opérations non linéaire
- assembleur incorporé, bibliothèques en code natif

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

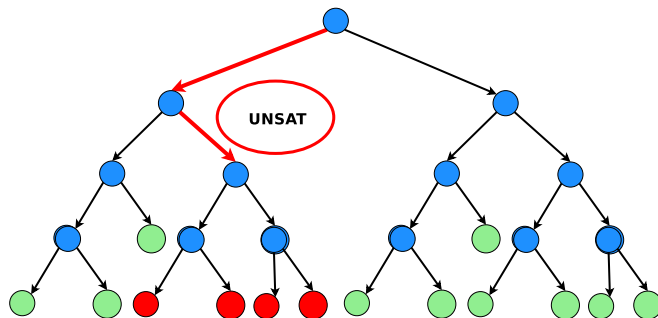


Supposons un chemin infaisable dans l'arbre des exécutions possibles

Problème PB1 : chemins infaisables



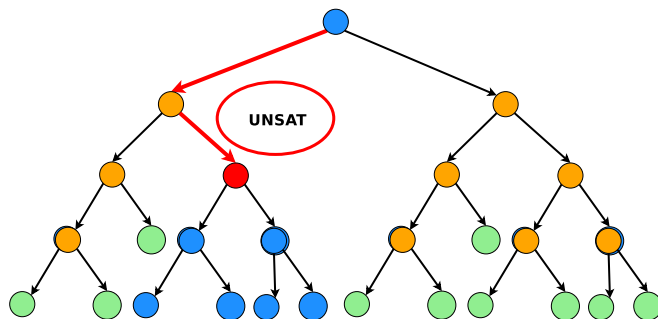
Supposons un chemin infaisable dans l'arbre des exécutions possibles



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

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 $Z = X * X + (X \text{ modulo } 2)$
- 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

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

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

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

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 $Z = X * X + (X \text{ modulo } 2)$
- 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

Hyp 1 : code source de f non disponible

Hyp 2 : $\times$ pas dans la théorie logique utilisée

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

Résultat idéal : $C_1 = f(Z_0) \wedge X_2 = Y_0 \times C_1 \wedge \text{AXIOMS_FUNCTION}(f)$

Exec symbolique : $\top$

Exec concolique : $C_1 = f(Z_0) = 4 \wedge Z_0 = 10 \wedge X_2 = Y_0 \times C_1$

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

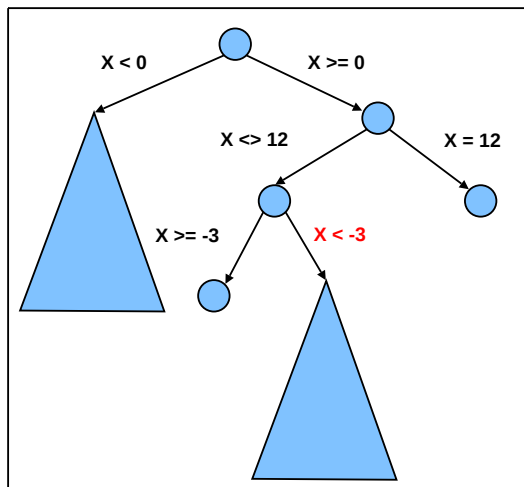
Exploiter l'information dynamique (2)

```
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 $P = (Y \neq 10)$ (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

Suivre seulement des chemins faisables



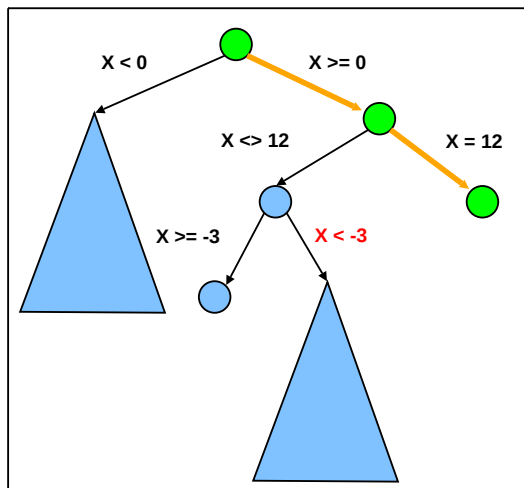
concret : $X=12$

backtrack + résolution, solution $X = 5$

concret : $X=5$

backtrack + résolution, unsat

Suivre seulement des chemins faisables



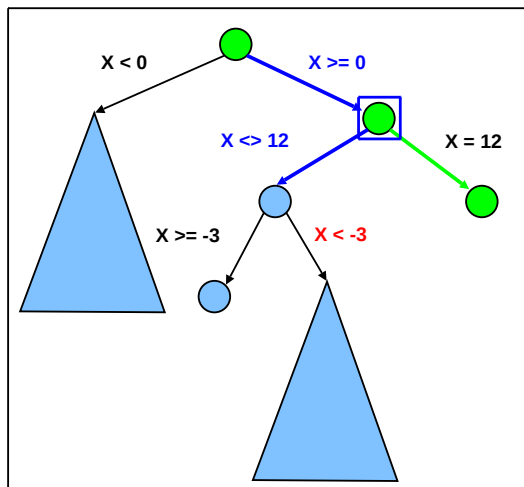
concret : $X=12$

backtrack + résolution, solution $X = 5$

concret : $X=5$

backtrack + résolution, unsat

Suivre seulement des chemins faisables



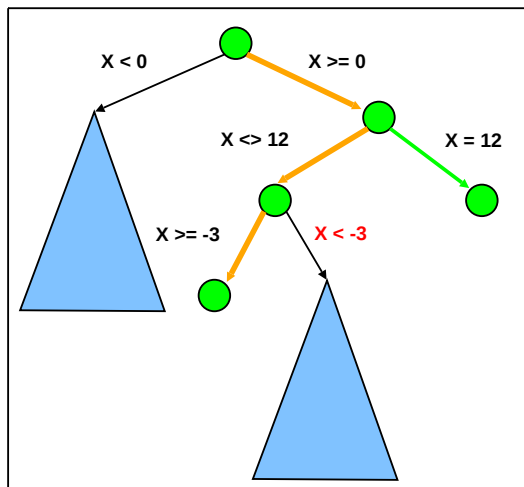
concret : $X=12$

backtrack + résolution, solution $X = 5$

concret : $X=5$

backtrack + résolution, unsat

Suivre seulement des chemins faisables



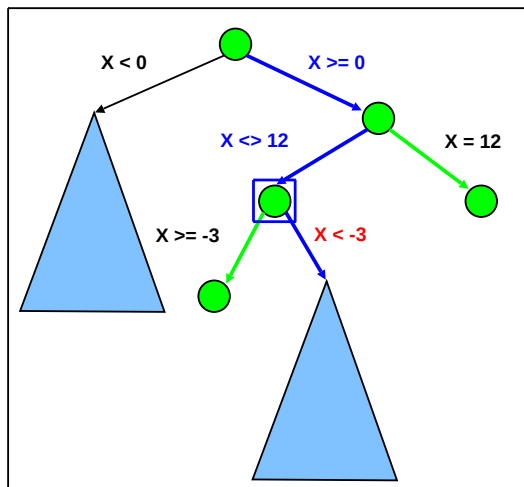
concret : $X=12$

backtrack + résolution, solution $X = 5$

concret : $X=5$

backtrack + résolution, unsat

Suivre seulement des chemins faisables



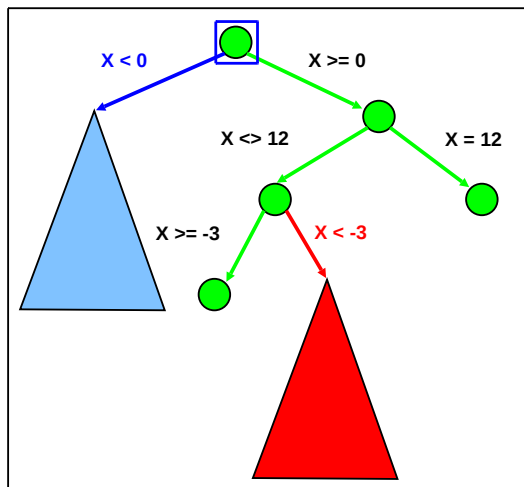
concret : $X=12$

backtrack + résolution, solution $X = 5$

concret : $X=5$

backtrack + résolution, unsat

Suivre seulement des chemins faisables



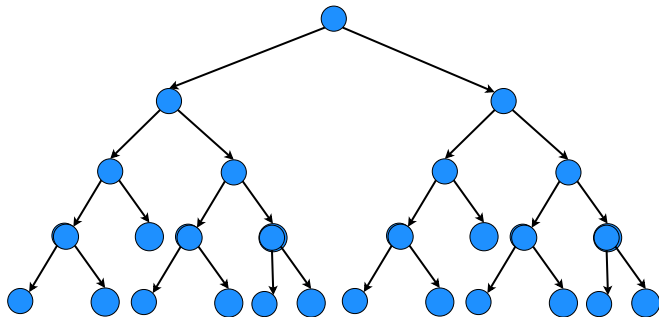
concret : $X=12$

backtrack + résolution, solution $X = 5$

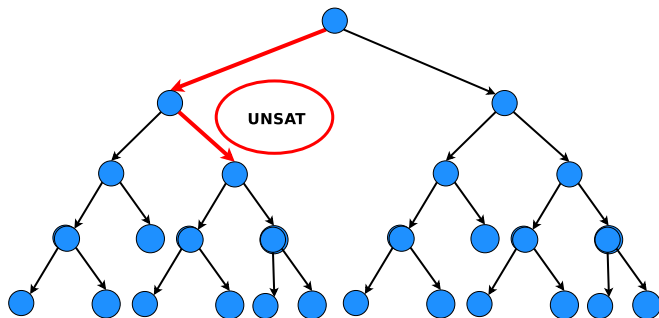
concret : $X=5$

backtrack + résolution, unsat

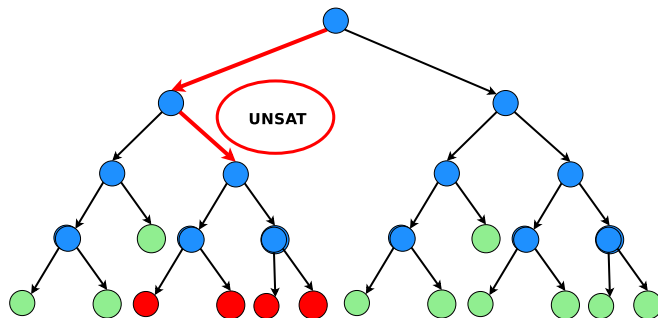
Suivre seulement des chemins faisables (2)



Suivre seulement des chemins faisables (2)



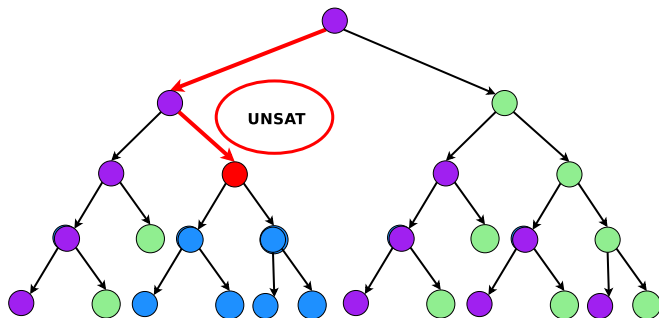
Suivre seulement des chemins faisables (2)



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

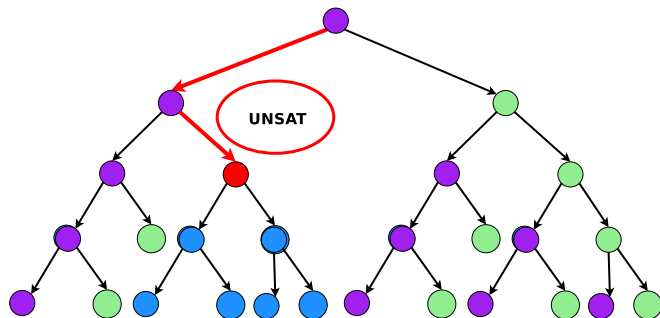
Suivre seulement des chemins faisables (2)



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 préfixe minimal infaisable
- + : toujours moins d'appels que les deux autres méthodes

Le mécanisme général de concrétisation peut être utilisé 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

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: | halt  $\rightarrow$  () /* end node */
3: | block  $i \rightarrow$  SEARCH( $n.next$ ,  $\pi \cdot n$ ,  $\Phi \wedge \text{SYMB}(i)$ , update( $C, i$ ))
4: | goto  $n' \rightarrow$  SEARCH( $n'$ ,  $\pi \cdot n, \Phi$ ,  $C$ )
5: | ite( $cond, in, tn$ )  $\rightarrow$ 
6:     Case eval( $cond, C$ ) of /* follow concrete branch */
7:     | true  $\rightarrow$ 
8:         SEARCH( $in$ ,  $\pi \cdot n$ ,  $\Phi \wedge \text{SYMB}(cond)$ ,  $C$ );
9:     try /* solve new branch first */
10:         $S_p :=$  SOLVE( $\Phi \wedge \neg cond$ ); Tests := Tests +  $\{(S_p, \pi.tn)\}$ 
11:         $C' :=$  UPDATE_C_FOR_BRANCHING( $S_p$ )
12:        SEARCH( $tn$ ,  $\pi \cdot n$ ,  $\Phi \wedge \neg \text{SYMB}(cond)$ ,  $C'$ ) /* branching */
13:    with unsat  $\rightarrow$  ()
14:    end try
15:    | false  $\rightarrow$  ..... /* symmetric case */
16:    end case
17: end case
```

Procédure concolique basique (2)

$\text{update}(C : \text{mem-conc}, i : \text{instr}) \rightarrow \text{mem-conc}$: m à j de l'état mémoire actuel

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

$\text{update_C_for_branching}(Sp : DT) \rightarrow \text{mem-conc}$: 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”).

Plutôt 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

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

Prise en compte des préconditions de la fonction à tester

Pourquoi : éviter des tests non pertinents

- ex : algorithme de recherche dichotomique
- le tableau d'entrée généré doit être trié, sinon le test n'est pas représentatif

SOLUTION 1 : filtrer a posteriori

- générer les DT comme avant
- puis éliminer toutes les DT ne respectant pas la précondition
- PB : nécessite une précondition exécutable
- PB : ne fonctionne pas si précondition trop contrainte

SOLUTION 2 : générer à coup sûr des DT satisfaisant *Pre*

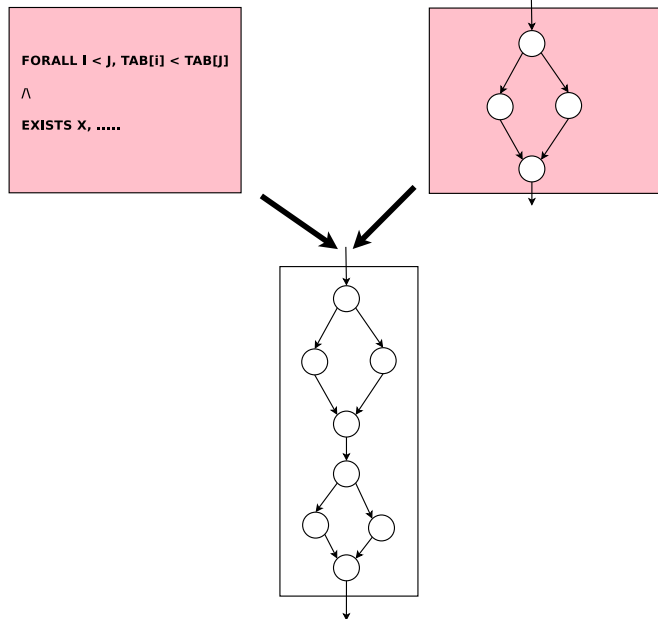
Approche 1 : gestion de la précondition au niveau logique

- on résoud des formules de la forme $\pi \wedge \text{PRECOND}$
- PB : les préconditions élaborées (ex : tableau trié) demandent des quantificateurs $\forall$

Approche 2 : gestion de la précondition au niveau du code

- on ajoute au début du programme une fonction `check_precond(args)`
- PB1 : les préconditions élaborées (ex : tableau trié) demandent des boucles (`#chemins ++`)
- PB2 : nécessite des préconditions exécutables

Préconditions complexes (3)



Remarque 1 : certaines préconditions simples sont aisées à intégrer, dans un cas comme dans l'autre

- exemple : $X \geq 0$, ou $X \neq \text{NULL}$

Remarque 2 : les préconditions peuvent aussi être un moyen donné à l'utilisateur pour diminuer l'espace de recherche de la DSE

- exemple : restreindre arbitrairement les valeurs possibles de l'input X

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

- rapport de tests plus informatifs (dt, chemin, **verdict**)
- idéal : générer directement des tests fautifs

On retrouve les mêmes problèmes que pour la précondition

- a posteriori : tests générés sans oracle, puis verdict ensuite
- a priori : tests générés *contre* l'oracle

Quel format pour l'oracle ?

- code ou formule : même pb que pour la précondition
- des oracles partiels simples (runtime errors) peuvent être légers et intéressants du point de vue guidage de la génération

En pratique :

- la prise en compte de la précondition à la génération est essentielle (correction des tests)
 - filtrage a posteriori pas satisfaisant
- pour l'oracle, la prise en compte à la génération est lourde et pas forcément couronnée de succès
 - ▶ filtrage a posteriori si le but est de considérer un oracle complet
 - ▶ intégration a priori envisageable avec un oracle partiel et léger

Sortir les tests dans un format réutilisable

- exporter vers autres outils
 - ▶ JUnit, calcul de couverture, sélection / minimisation, etc.
- utilisation conjointe de DT issues d'autres méthodes / outils
 - ▶ tests issus de méthodes manuelles ou orientées modèles
 - ▶ méthodes automatiques simples (random testing, interface-based testing)

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

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.)

PathCrawler online

[http ://pathcrawler-online.com/](http://pathcrawler-online.com/)

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

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 : traçabilité 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

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

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

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

Chaque instruction doit être traduite vers une formule

- le langage des formules peut être très riche

Les contraintes doivent être résolues automatiquement

- satisfiabilité décidable + résolution efficace *en pratique*
- beaucoup moins de liberté !!

Remarques

- compromis expressivité VS décidabilité / complexité
- si théorie pas assez expressive : approximations [concrétisation]

Une théorie logique T

- un ensemble prédéfini de symboles de fonctions et prédicats [en général au moins =]
- une sémantique implicite [domaine des variables, “sens” des fonctions et prédicats]
- des axiomes imposant la sémantique implicite
- on note $T \models \varphi$ pour dire que la formule φ est valide dans la théorie T

Un fragment logique

- on limite les connecteurs logiques
- typiquement : pas de quantificateur, pas de $\vee$

À propos des théories utilisées (2)

Avantage d'être sur un chemin

- fragment simple : pas de quantificateur, 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 [Simplex (non polynomial)])
- $(\mathbb{N}, +, \times k)$ (arithmétique linéaire, NP-complet [Omega test])
- $\mathcal{B}$ (booléens, NP-complet [DPLL ou BDDs])
- $(\mathbb{N}_{\leq}, +, \times)$ (arithmétique bornée non linéaire, NP-complet [CP(FD)])
- BV (bitvecteurs, NP-complet [bitblasting])
- $FLOAT$ (arithmétique flottante, NP-complet)
- types algébriques (algèbre libre, P [unification])

Exemple de théorie : EUF

Théorie des fonctions non interprétées (EUF)

signature : $\langle =, \neq, x_1, \dots, x_n, f_1(\dots), \dots, f_m(\dots) \rangle$

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)*

Résolution très efficace

- algorithme de *congruence closure* (cf Wikipedia)
- polynomial

Variantes

- (AC) axiomes d'associativité / commutativité sur certains symboles de fonction
- (free-algebra) algèbre de types libres

Théorie des tableaux

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

sémantique :

- $load : ARRAY \times I \mapsto E$
- $store : ARRAY \times I \times E \mapsto ARRAY$
- 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.

Résolution : EUF + case-split, problème NP-complet

Quelles théories en pratique ?

expressivité ↗ : moins d'échecs mais résolution sur un chemin + chère

- ex : BV + Array (NP-complet)

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

- ex : Difference + EUF (Polynomial)

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, $(\mathbb{N}_{\leq}, +, \times)$

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

Traduction 1 :

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

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

Traduction 1 :

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

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \dots\}$

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

Traduction 1 :

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

Modèle mémoire sous-jacent : ensemble de variables $\{A, B, X, \dots\}$

Bien mais ne pourra prendre en compte les pointeurs

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

Traduction 2 : ajout d'un état mémoire M

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

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

Traduction 2 : ajout d'un état mémoire M

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

Modèle mémoire sous-jacent : $map \{Addr_1 \mapsto A, Addr_2 \mapsto B, \dots\}$

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

Traduction 2 : ajout d'un état mémoire M

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

Modèle mémoire sous-jacent : $\text{map } \{Addr_1 \mapsto A, Addr_2 \mapsto B, \dots\}$

ok pour les pointeurs, mais on ne peut écrire au milieu de x
(respect du typage, modèle mémoire à la Java)

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

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

Traduction 3 : encodage de M au 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 = tmpA+tmpB
in
  store(
    store(
      store(M, addr(X), nX[0]),
      addr(X) + 1, nX[1]),
    addr(X) + 2, nX[2])
```

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

Traduction 3 : encodage de M au 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 = tmpA+tmpB
in
  store(
    store(
      store(M, addr(X), nX[0]),
      addr(X) + 1, nX[1]),
      addr(X) + 2, nX[2])
```

ok pour du C ... mais la formule est complexe

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

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

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

Diminuer le coût moyen d'un appel au solveur

- on se place dans le cas où le solveur est utilisé en boîte noire
- on peut cependant quand même améliorer ses performances

Diminuer le nombre d'appels au solveur

- via des parcours de chemin plus malins que la DFS

- système de cache
- réutilisation des solutions précédentes
- réutilisation de coeurs insatisfiables
- séparation des sous-formules indépendantes
- simplification de formule
- solveur “léger”

Cache de formules : pour certaines φ' déjà résolues, on garde dans un cache $\mathcal{C}$: $\varphi' \mapsto \text{SAT}$ (et une solution) ou $\varphi' \mapsto \text{UNSAT}$.

Soit φ à résoudre, à chaque appel du solveur on fait :

- si $\exists \varphi' \in \mathcal{C}$ tq $\varphi \Rightarrow \varphi'$ et φ' UNSAT, alors φ UNSAT
- sinon si $\exists \varphi' \in \mathcal{C}$ tq $\varphi' \Rightarrow \varphi$ et φ' SAT, alors φ SAT (et même solution)
- sinon $\text{SOLVE}(\varphi)$

Remarque : calculer $\Rightarrow$ est coûteux, on approxime $A \Rightarrow B$ par $B \preceq A$ (sous-terme syntaxique)

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

* Supposons que l'on a déjà résolu

$$X = Y + 3 \wedge X \leq 5 \wedge B \geq 0$$

solution trouvée : $X = 6$, $Y = 8$, $B = 0$

* 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 (6 et 8), et on résoud
seulement $B \geq 0 \wedge B + 12 \leq Z$

Remarque : peut être simulé en utilisant **séparation des sous-formules indépendantes** (cf après) et **utilisation du cache**

Solution tout de même intéressante

- plus facile à mettre en oeuvre qu'un cache de calcul (mais moins général)
- même si le cache est dispo, économise la recherche dans le cache

Soit φ une formule

En cas de UNSAT, les solveurs modernes de type SMT retournent un coeur insatisfiable

- une formule ϕ tq ϕ UNSAT et $\varphi \Rightarrow \phi$
- ϕ est un “témoin” minimal d’insatisfiabilité de φ
- normalement ϕ plus simple que φ

Ex : $X < Y \wedge X > Y$ est un coeur insat. de

$X < Y \wedge X + 10 = 100 \times Z \wedge Z = B - A \wedge X > Y$

Comment utiliser cette information

- dans le cache, utiliser ϕ plutôt que φ
- autre ? lancer le solver en lui ajoutant comme axiome $\neg\phi$?

Séparation de formules : Soit $\vec{V}$, $\vec{V}_1$, $\vec{V}_2$ des ensembles de variables. Si $\varphi(\vec{V})$ peut se décomposer en $\varphi_1(\vec{V}_1) \wedge \varphi_2(\vec{V}_2)$ et $\vec{V}_1 \cap \vec{V}_2 = \emptyset$, alors :

- si $\text{SOLVE}(\varphi_1)$ retourne UNSAT alors UNSAT
- sinon si $\text{SOLVE}(\varphi_2)$ retourne UNSAT alors UNSAT
- sinon SAT, et une solution = $\text{solution}(\vec{V}_1) \cup \text{solution}(\vec{V}_2)$

Rmq 1 : dès que SOLVE a une complexité supérieure à linéaire, on gagne à séparer la formule

Rmq 2 : (pratique, cas UNSAT) dans quel ordre résoudre les φ_i ?

Rmq 3 : plus les formules sont petites, mieux le cache fonctionne

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)

Exemple de chemin :

```
y := y+1; x := a+b; assume(y < 10); return x
```

- prédicat de chemin : $Y_1 = Y_0 + 1 \wedge X_1 = A_0 + B_0 \wedge Y_1 < 10$
- avec slicing : $Y_1 = Y_0 + 1 \wedge Y_1 < 10$

Remarque : si le langage de formules permet la définition de termes, alors le slicing se fait aussi bien niveau chemin que niveau formule

chemin `y := y+1; x := a+b; assume(y < 10); return x`

- prédicat en avant :

`let Y1 ::= Y0 + 1 in let X1 ::= A0 + B0 in Y1 < 10`

- le terme X_1 n'est jamais utilisé, on l'enlève

- formule simplifiée : `let Y1 ::= Y0 + 1 in Y1 < 10`

propagation de constantes : règles de la forme

- (élim de var) $X = 5$: remplacer X par 5 dans toute la formule, se souvenir de $X = 5$
- (élim de déf) $\text{let } X : := 5$: remplacer X par 5 dans toute la formule, éliminer X
- (calcul de terme) $5 + 3$: faire le calcul, remplacer par 8
- (calcul partiel de terme)
 - $X + 0$: remplacer par X
 - $X \times 0$: remplacer par 0
 - $X \times 1$: remplacer par X
 - ...

propagation d'égalités

- si $X = Y$, garder seulement X ou seulement Y dans la formule
- on peut étendre aux opérateurs : $X = A + B \wedge Y = A + B$
alors $X = Y$

unification des sous-termes identiques (introduction de défs)

- transformer $X = A + B + 1 \wedge Y = A + B + 2 \wedge B \leq Z$ en
let $T := A + B$ in $X = T + 1 \wedge Y = T + 2 \wedge B \leq Z$

On peut normaliser les opérateurs AC pour trouver plus d'égalités

- AC : associatif - commutatif
- fonctionne pour propagation des égalités et unification de sous-termes identiques
- ex : réécrire les additions en ordonnant les opérandes selon ordre lexicographique
- ex : $X = A + B \wedge Y = B + A$, on normalise en
 $X = A + B \wedge Y = A + B$, on déduit que $X = Y$

reconnaissance et utilisation des variables “proxy”

- sur une formule de type $X = A + B \wedge X + 3 \leq 100$
- X est un proxy pour A et B : A et B ne sont pas utilisés ailleurs, et $X = A + B$ est satisfaisable pour toute valeur de X
- dans ce cas : on enlève A et B de la formule (on résout avec X), et on résout à part $X = A + B$
- ici on peut même imposer directement par exemple $A = X$ et $B = 0$

Remarques

- ne marche pas si A ou B ont d'autres contraintes
- ce n'est pas un cas particulier de formules indépendantes

On utilise déjà un solveur peu coûteux mais incomplet

- répond UNSAT ou MAYBE

Le solveur léger est lancé avant le solveur complet

- on gagne du temps si la formule est UNSAT

Comment obtenir le solveur léger ?

- le faire soi-même
- utiliser un solveur existant avec juste des théories simples
- un preprocessing élaboré peut servir de solveur léger

L'efficacité de ces optims dépend du type de logiciel

- pour le code de type “parseur simple”, les systèmes de cache et de séparation de sous-formules fonctionnent très bien

Les optimisations se combinent bien

- plus on simplifie, plus on peut séparer les formules
- plus les formules sont simples / petites, plus le cache fonctionne
- ...

Deux familles d'optimisation

- modifier l'ordre de parcours des chemins pour couvrir plus vite
- diminuer le nombre de chemins à considérer

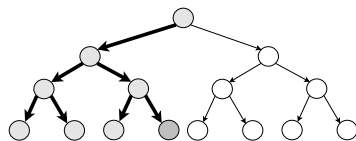
Technique usuelle : parcours en profondeur (DFS)

Avantage classique de DFS

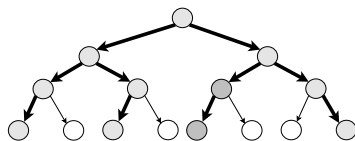
- un seul contexte ouvert à la fois (mémoire)
- simple à implanter en récursif

Problème de la DFS pour la génération de tests

- si #DT limité, la DFS se concentre sur une portion très restreinte du code



(a) DFS



(b) BFS

Couverture pour 4 tests générés

Couverture des chemins par DSE : tous les parcours de chemins se valent

Couverture des branches (budget limité ou non) : tous les parcours de chemins ne se valent pas

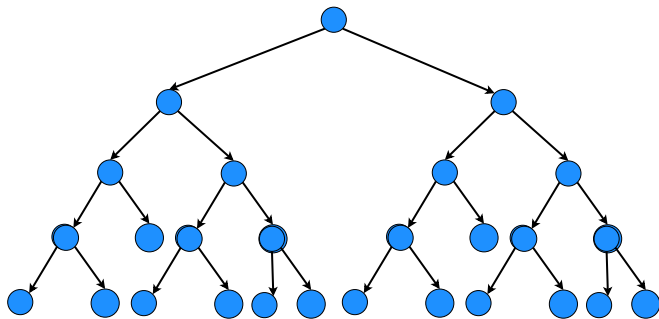
- DFS est souvent très mauvais

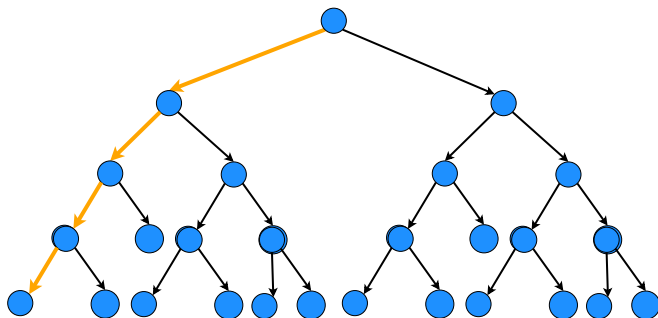
Quelques solutions

- parcours hybride (CUTE) : DFS + aléatoire
[simple, meilleur que dfs]
- *fitness guided* [EXE, SAGE, PEX] : les préfixes actifs sont évaluées, celui de plus haut score est étendu
[mécanisme très versatile]

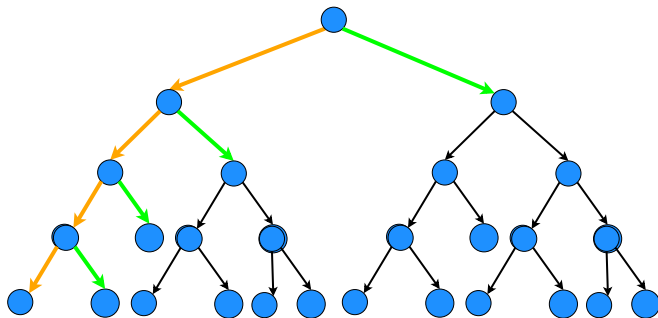
Ingrédients de l'exécution symbolique "Fitness-guided"

- **chemin actif** : chemin non couvert, dont le plus long (strict) préfix est couvert [on dit aussi préfix actif]
- notion de **score** d'un chemin actif
- à chaque étape : **sélection** + **extension** +
 - ▶ choisir le chemin actif ayant le meilleur score
 - ▶ "étendre" ce chemin : résolution + exécution
 - ▶ ajouter les nouveaux chemins actifs créés par cette exécution

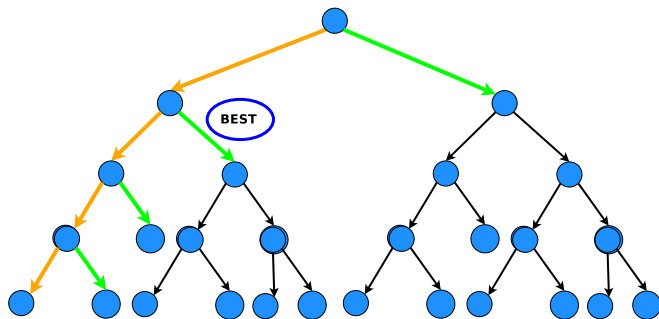




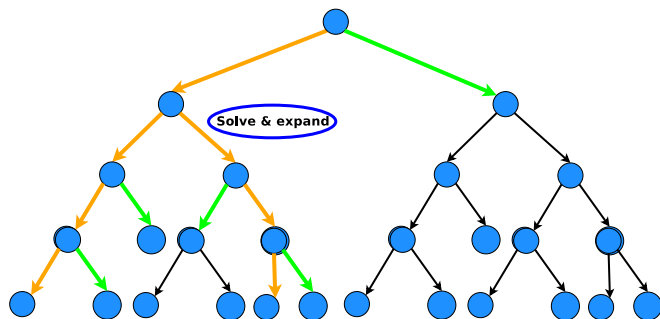
Parcours fitnex-guided (2)



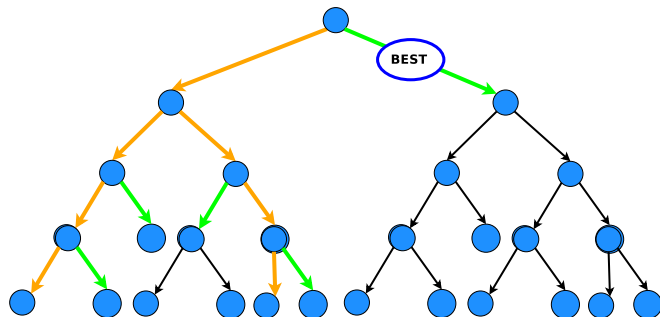
Parcours fitnex-guided (2)

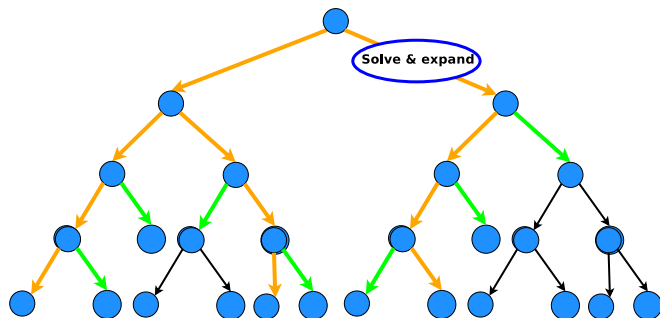


Parcours fitnex-guided (2)



Parcours fitnex-guided (2)





Nous supposons disponible les deux fonctions de base suivantes :

- `get_initial_value` : $\varepsilon \mapsto \text{inputData}$: *fournit une donnée d'entrée concrète initiale (constante ou aléatoire).*
- `get_new_paths` : $\text{inputData} \mapsto \text{set}\langle \text{path} \rangle$: *lance une exécution concrète à partir de valeurs d'entrées, observe le chemin π suivi à l'exécution et retourne les préfixes actifs de π qui n'ont pas encore été collectés. Une implémentation réelle demande un type plus complexe, prenant en compte l'historique des préfixes actifs collectés jusque là.*

L'algorithme générique nécessite un type abstrait VAL de score, et les deux fonctions abstraites suivantes :

- `score` : $\text{path} \mapsto \text{VAL}$: *évaluation d'un préfixe*
- `compare` : $\text{VAL} \times \text{VAL} \mapsto \{<, =, >\}$: *comparaison à partir du score*

La fonction suivante est déduite facilement :

- `get_best` : $\text{set}\langle \text{path} \rangle \mapsto \text{path}$: *utilise compare*

input : un programme P

output : RES : ensemble de couples $(\pi - dt)$, tq pour chaque couple, P(dt) couvre π et l'ensemble des dt couvre toutes les branches faisables de P

```
1: RES :=  $\emptyset$ 
2:  $v_0 := \text{get\_initial\_value}$  /* arbitrary initial concrete value */
3:  $H := \text{get\_new\_paths}(v_0)$  /* get all active prefixes from an execution */
4: While still uncovered branches or paths do
5:    $\pi := \text{get\_best}(H)$ 
6:    $H := H \setminus \{\pi\}$ 
7:   case SOLVE(SYMB( $\pi$ )) of
8:     | UNSAT  $\rightarrow$  nop
9:     | SAT( $v$ )  $\rightarrow$ 
10:       RES := RES  $\cup \{(\pi, v)\}$ ;
11:        $H := H \cup \text{get\_new\_paths}(v)$ 
12:   end case
13: end while
14: return RES
```

Quelle fonction de Score ?

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
- ...

Quelles fonctions de Score (2) ?

Quelques exemples d'heuristiques

minCallDepth-dfs

- les chemins dont le noeud final a la plus petite profondeur d'appel sont prioritaires
- puis dfs sur ces préfixes

hybrid dfs

- alterner k_1 test aléatoire puis k_2 étapes de dfs à partir du dernier chemin aléatoire

Best first

- le prochain chemin est celui ayant le gain le plus élevé
- gain, ex1 : nb de nouvelles instructions couvertes à coup sûr
- gain, ex2 : nb d'instructions successeurs non encore couvertes

Élimination de chemins redondants : certains chemins sont redondants pour le critère de couverture choisi, on peut les éviter

- techniques souvent complètes

Gestion des appels de fonction : cause principale de l'explosion du nombre de chemins

- techniques souvent incomplètes

Couper les chemins qui ne peuvent atteindre de nouvelles instructions

- pour chaque chemin actif, calcul des instructions accessibles à partir de l'état actif
- on stop le chemin si le préfixe ne peut atteindre de cible non couverte
- calcul des accessibles peut être fait très efficacement
- technique complète vis à vis de la couverture d'instructions

Technique complète vis à vis de la couverture d'instructions

- on ne perd rien

Technique puissante : (DFS + optim) meilleure que bcp de parcours avancés

Couper les chemins amenant à un état symbolique déjà couvert : si on a deux préfixes π et σ tq $\phi_\pi \Rightarrow \phi_\sigma$, alors on garde seulement le préfixe σ

- technique complète vis à vis de la couverture d'instructions
- potentiellement très coûteuse, demande de vérifier $\Rightarrow$
- on peut utiliser un calcul approché de $\Rightarrow$ via $\preceq$

Technique complète vis à vis de la couverture d'instructions

Reste un problème ouvert

Quelques solutions partielles

- 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)

Couper l'exploration à une certaine profondeur

- simple à mettre en oeuvre !!
- pas complet

Variantes

- concrétiser : prédicat de chemin correct et simple, mais très (trop ?) contraint
- remplacer l'appel de fonction par $\top$: formule simple mais incorrecte (!)
- rentrer dans la fonction mais empêcher l'énumération de chemin : correct, mais formule compliquée

Résumés de fonction de type $Pre(\vec{In}) \wedge Post(\vec{In}, \vec{Out})$

C'est le cas idéal

- pas besoin d'entrer dans la fonction appelée pour énumérer les chemins
- pas besoin d'entrer dans la fonction appelée pour générer des contraintes

Attention

- le solveur doit pouvoir gérer la spéc
- la spéc doit être fonctionnelle, et suffisamment précise pour permettre de déduire des valeurs (pas de surapproximation)
- qui donne la spéc ?

Alternatives pour les résumés de fonction

- résumés en sous-approximation
- utilisation incrémentale de résumés en surapproximation

Résumés en sous-approximation de type $\forall \vec{in} = c_i \Rightarrow \vec{out} = o_i$

- correct, et peut être construit pendant l'exploration
- rappelle la logique des “stubs” en test usuel
- limite le raisonnement symbolique, ajoute des $\forall$

Utilisation incrémentale de résumés en surapproximation [technique correcte]

- résoudre sans “entrer dans les fonctions”, avec le résumé.
- si UNSAT, stop
- sinon, la DT suit le préfixe ? oui OK
- sinon, “déplier” les appels de fonction et résoudre

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

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)

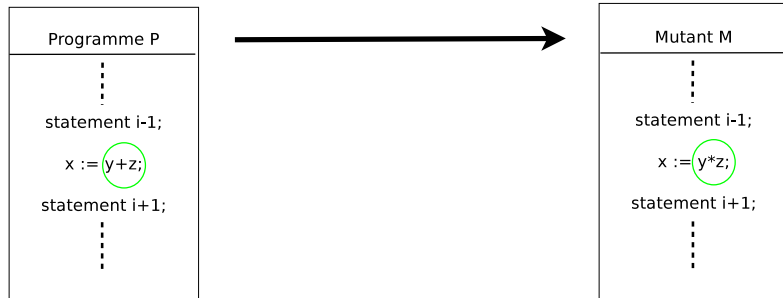
■ Idée :

- ▶ Transformer les mutants (faibles) en prédicats/labels dans le programme

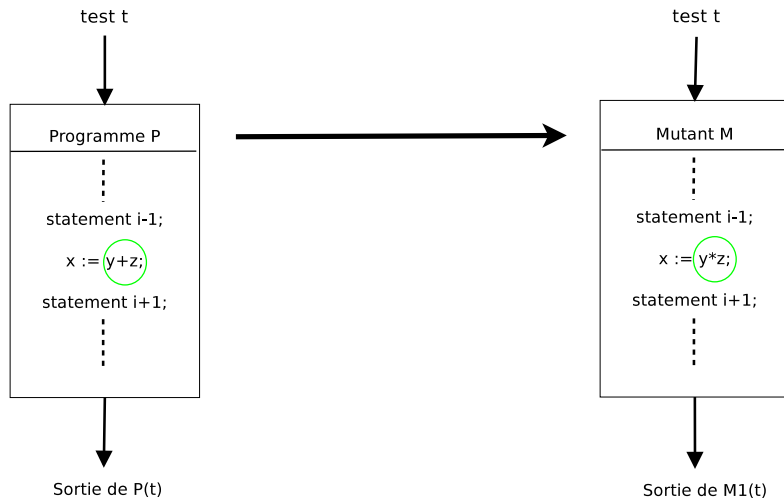
■ Pourquoi :

- ▶ Bénéficier des avantages des deux familles de critères
- ▶ Automatiser efficacement la couverture de mutants
- ▶ Étendre l'exécution concolique à des critères de test avancés

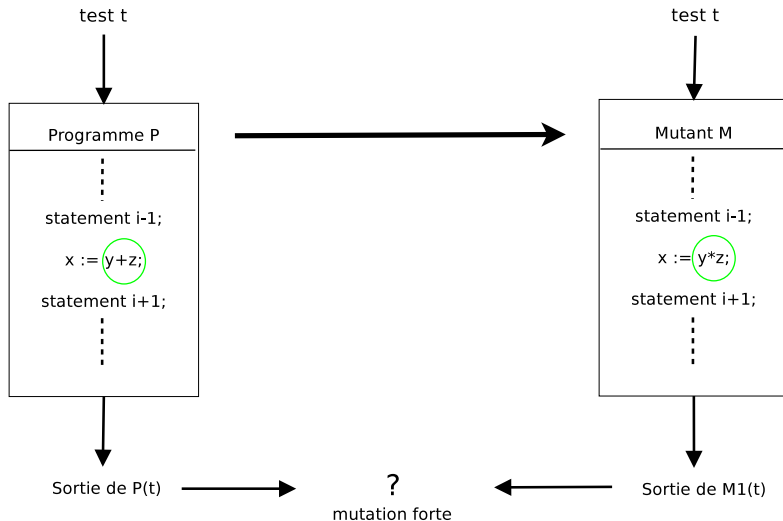
Mutations fortes



Mutations fortes



Mutations fortes



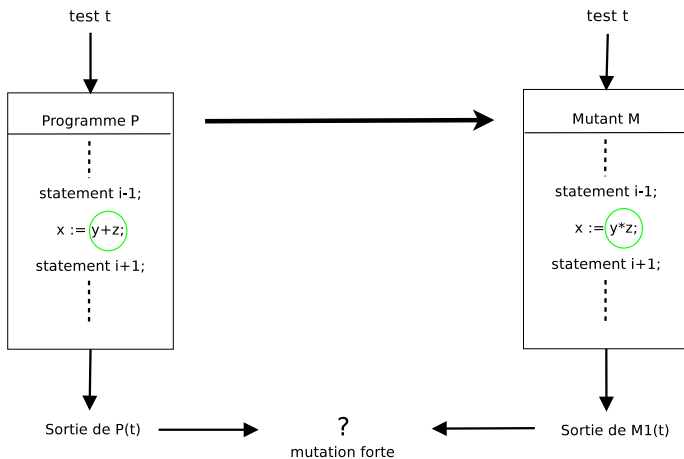
Intérêt

- 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

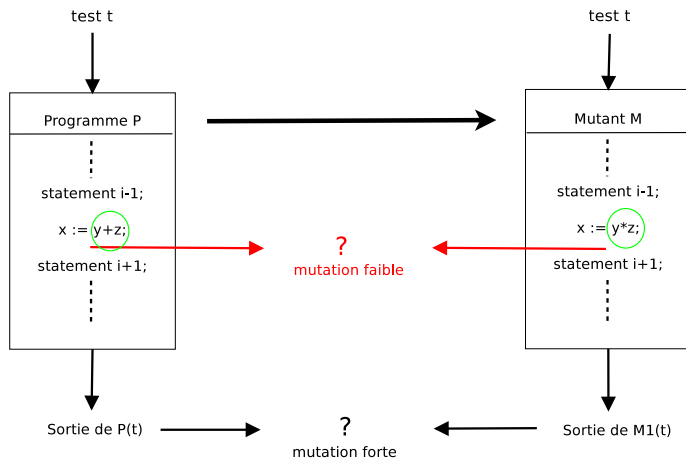
Difficile à automatiser

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

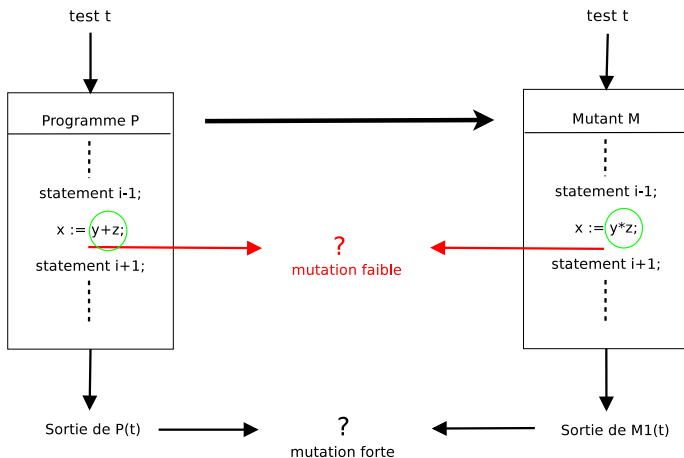
Mutations faibles



Mutations faibles

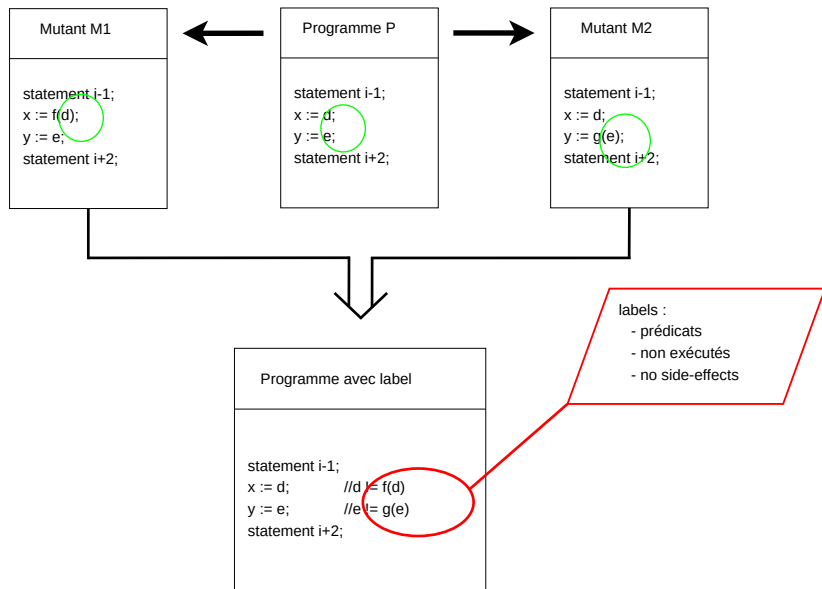


Mutations faibles



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

Des mutants faibles aux 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 :

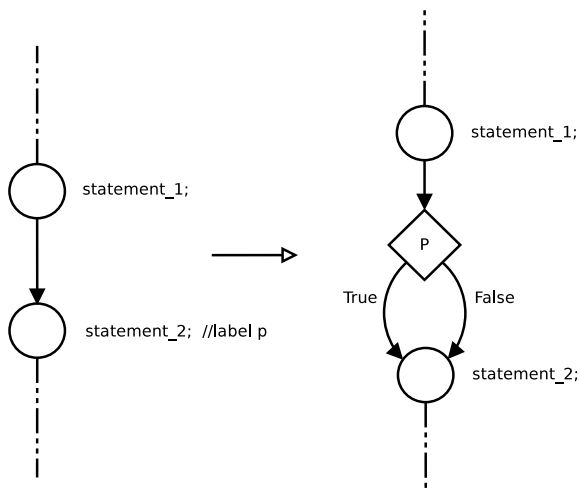
- réutilisation d'outils de vérification
- automatisation efficace

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



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

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

Pour un jeu de 100 tests

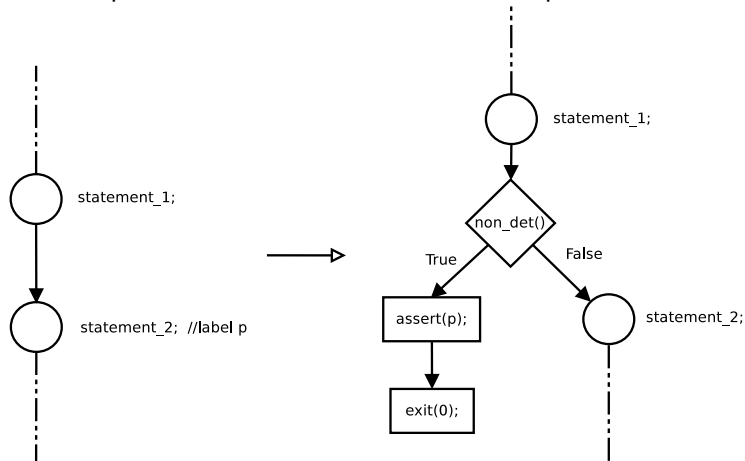
		Emma		MuJava
		programme initial	programme avec labels	mutants
TCas	124 LOC 111 labels	0,03 s	0,03 s	5 s
Replace	436 LOC 607 labels	0,05 s	0,10 s	40 s
Jtopas	5400 LOC 610 labels	3,22 s	11,72 s	1 400 s

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

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

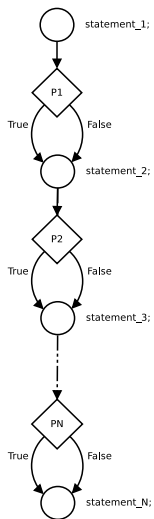
- nombre exponentiel de chemins (cf ci après)
- les chemins ajoutés sont complexes

Chaque chemin d'exécution contient au plus un label



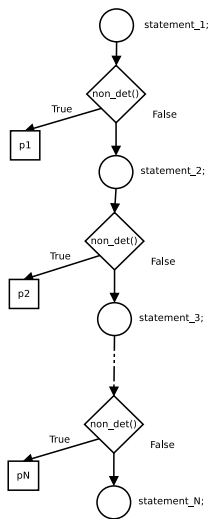
Comparaison des 2 instrumentations

Instrumentation
naïve



2^N chemins

Instrumentation
fine



$N+1$ chemins

- 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

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

PC_i⁺⁺ : couverture maximale + temps raisonnable

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

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