Approche concolique pour la génération de tests à partir de code

Panorama du test automatique de programmes au CEA

Sébastien Bardin, Nicky Williams

prenom.nom@cea.fr

CEA-LIST, Laboratoire de Sûreté Logicielle

AFADL 2009

Panorama de l'exposé

BUT : présenter les activités et outils de test structurel du CEA

Deux outils au CEA LIST/LSL pour le test structurel

- PathCrawler pour les programmes C
- Osmose pour le code binaire
- ~~GaTeL pour Lustre / Scade~~

Points communs

- approche
  - symbolique : basé sur les contraintes
  - locale : un chemin à la fois + énumération de chemins
  - combinaison symbolique - concret
- but : orienté couverture du programme

Rappels sur la génération de tests

Génération automatique de tests : 2 problèmes

- génération de données de tests (DT)
- génération de l'oracle

Deux familles de techniques de génération de tests

- fonctionnelle : oracle ok, mais DT ? modèle formel ?
- structurelle : DT ok, mais oracle ?

Approches structurelles

On se concentre dans cet exposé sur les méthodes structurelles

... et plus précisément sur la génération de DT avec objectifs de couverture structurelle

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#)

Approches structurelles : panorama

Deux grandes familles de génération structurelle

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

orientées recherche : explorer l'espace des entrées possibles jusqu'à en trouver une satisfaisante, la recherche se base sur des heuristiques venues de l'optimisation

Deux familles d'objectifs de test

- couverture d'un élément précis du programme
- couverture complète d'une classe d'éléments du programme

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

Approches globales : tout le programme est transformé en une formule logique

- théories : quantificateurs ou points fixes pour les boucles

Approches locales : un seul chemin est considéré à la fois

- théories : sans quantificateur, juste conjonction
- résolution pour un chemin + énumération de chemins

Génération de tests structurels par contraintes (2)

Récapitulatif du test structurel par contraintes

	cible	couverture
local		PathCrawler
		Osmose
global	Inka	
	(GaTeL)	

Dans la suite : exécution concolique

- approche contrainte
- approche locale
- combinaison symbolique - concret
- orienté couverture du programme

Plan

Prédicat de chemin

π un chemin (fini) du CFG du programme
D l'espace des entrées du programme
$V \in D$ une entrée du programme

Prédicat de chemin

Un prédicat de chemin pour π est une formule logique φ_π interprétée sur D telle que si $V \models \varphi_\pi$ alors l'exécution du programme sur V suit le chemin π.

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

Prédicat de chemin (2)

Contexte

Concolique

Discussion

PathCrawler

Osmose

Conclusion

Plus formellement : $\pi = \xrightarrow{t_1} \xrightarrow{t_2} \ldots \xrightarrow{t_n}$

Alors

- prédicat de chemin le plus lâche :
  $\bar{\varphi_\pi} = wpre(t_1, wpre(t_2, \ldots wpre(t_n, \top)))$
- prédicat de chemin : φ_π tel que $\varphi_\pi \Rightarrow \bar{\varphi_\pi}$

Remarques

- sous-approximation de $\bar{\varphi_\pi}$: risque de râter des DT
- sur-approximation de $\bar{\varphi_\pi}$: DT incorrecte (ne suit pas π)

Prédicat de chemin, exemple

Loc	Instruction	Exécution symbolique
0	input(y,z)	Y_0, Z_0
1	y++	$Y_1 = Y_0 + 1$
2	x := y + 3	$X_2 = Y_1 + 3$
3	if (x < 2 * z) (branche True)	$X_2 < 2 \times Z_0$
4	if (x < z) (branche False)	$X_2 \geq Z_0$

Prédicat de chemin (entrées Y_0 et Z_0)

- $Y_1 = Y_0 + 1 \wedge X_2 = Y_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$
- (projection) $Y_0 + 4 < 2 \times Z_0 \wedge Y_0 + 4 \geq Z_0$

Framework : exécution symbolique

Génération de tests basée sur les chemins

1. choisir un chemin π du CFG
2. calculer le prédicat 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

Paramètres : théorie logique, énumération de chemins

Problèmes de l'exécution symbolique

Chemins infaisables

- 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

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

Comment assurer une sous-approximation ?

L'exécution concolique apporte des solutions à ces problèmes

Combinaison d'exécutions symboliques et concrètes

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

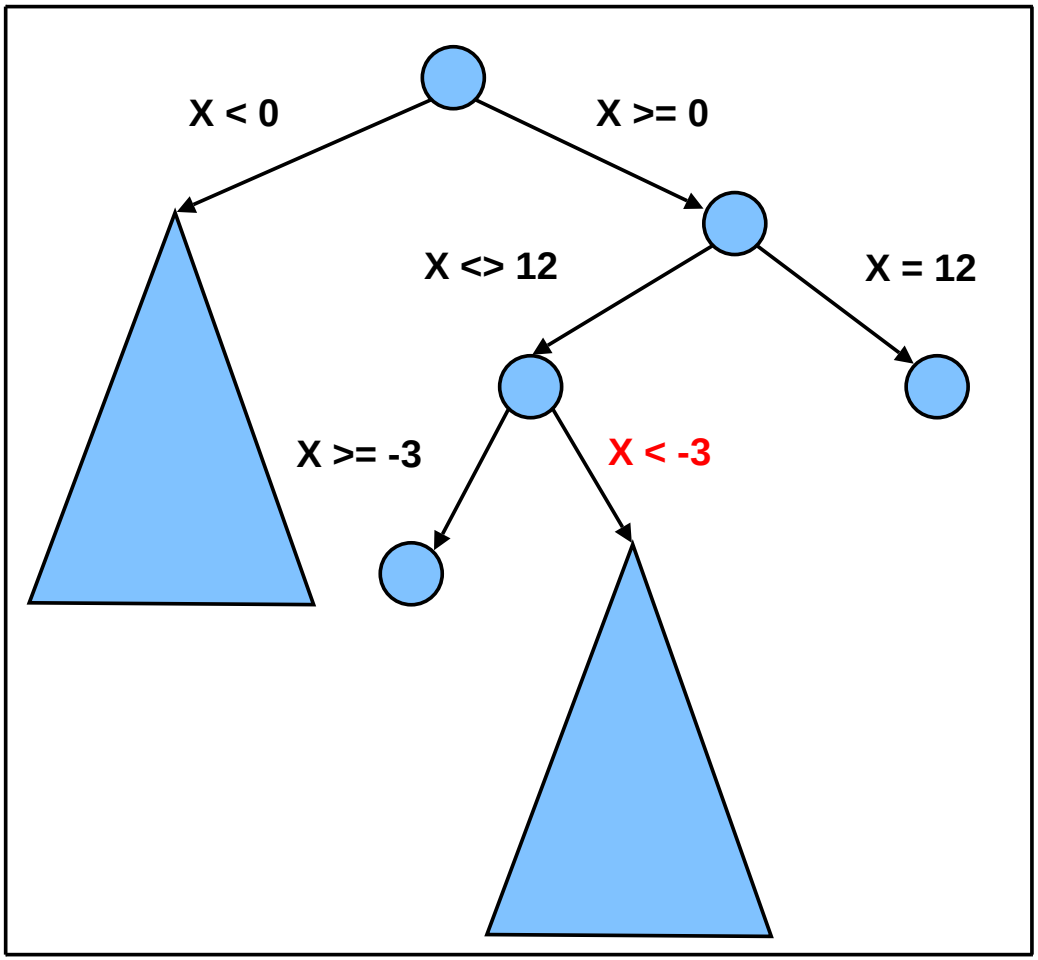
Exécution concrète : collecte des infos pour aider le raisonnement symbolique

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

Deux utilisations typiques

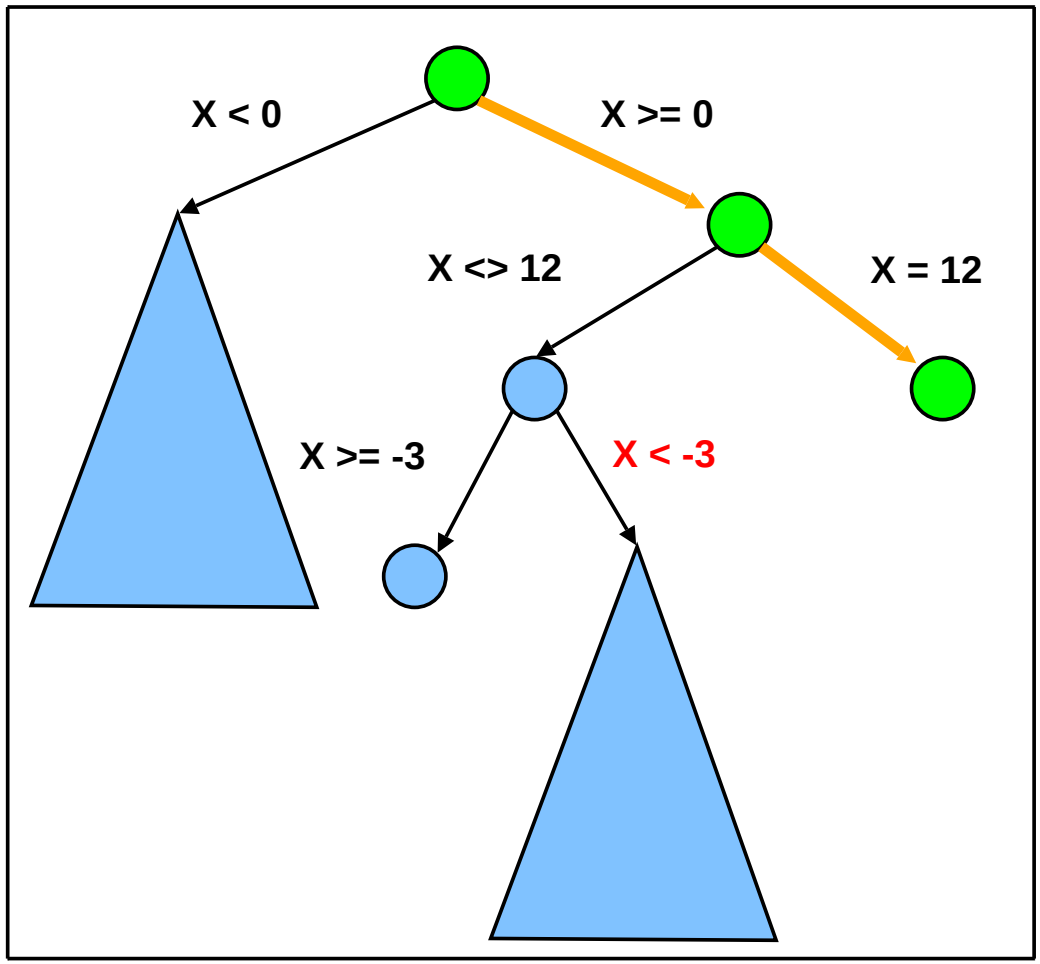
- suivre uniquement des chemins faisables à moindre coût
  toujours suivre une exécution concrète + résoudre au plus tôt
- approximation de constructions du langage "difficiles"
  concrétisation d'une partie des entrées/sorties
  approximations correctes ou non [choix]

Suivre seulement des chemins faisables



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

Suivre seulement des chemins faisables



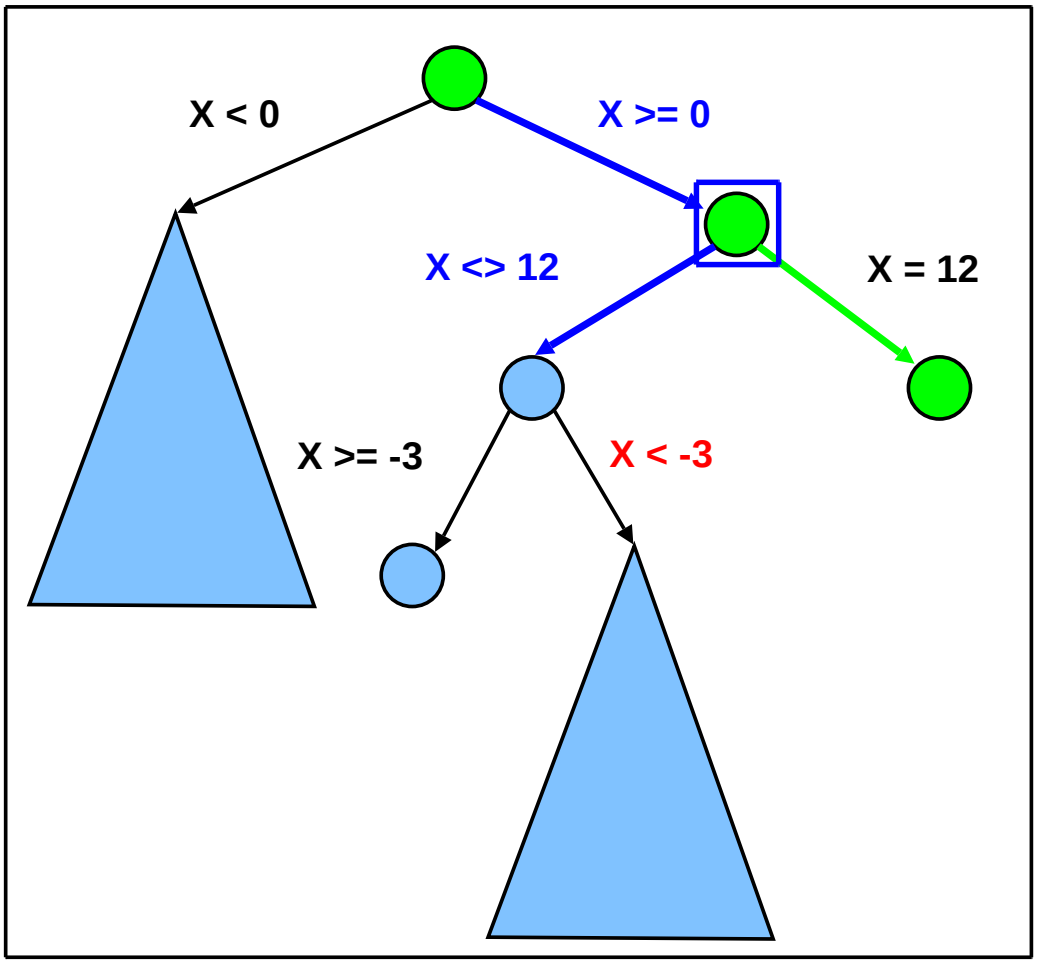
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

backtrack + résolution, unsat

Suivre seulement des chemins faisables



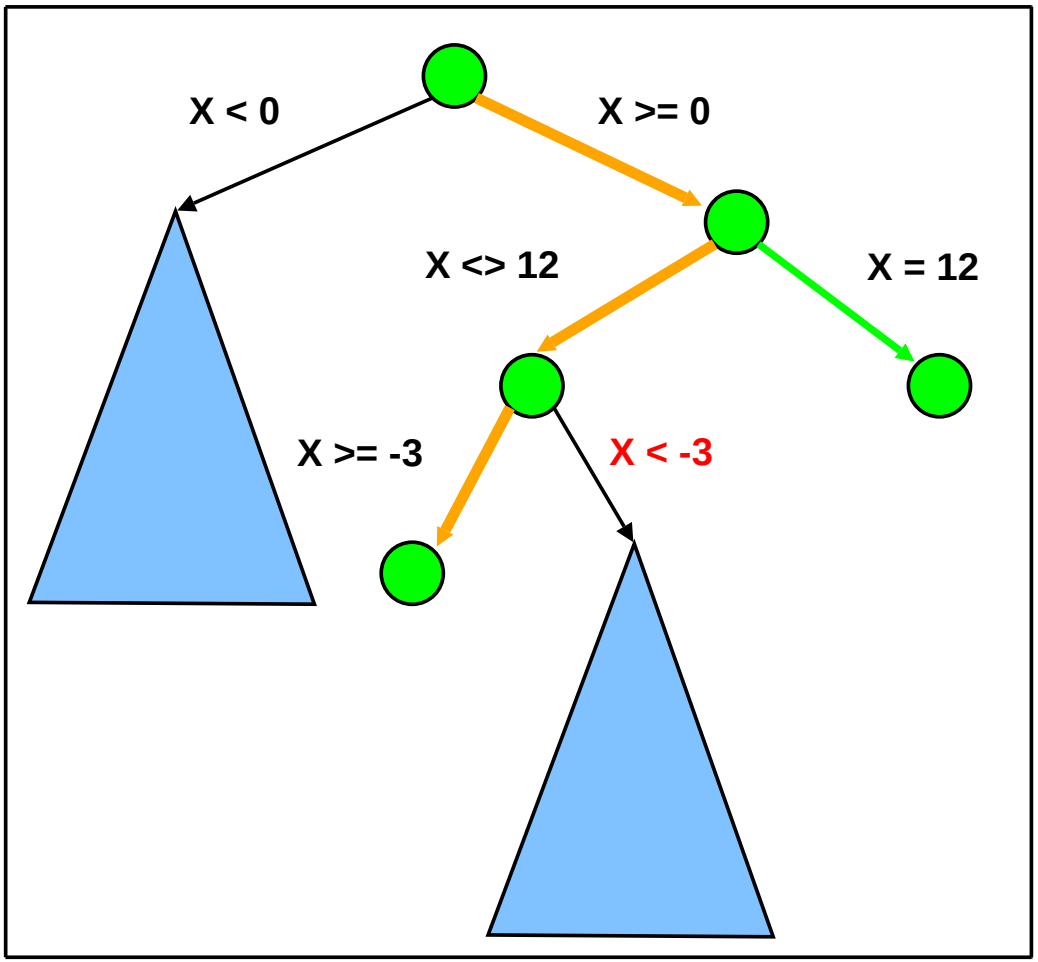
concret : X=12

backtrack + résolution, solution X = 5

concret : X=5

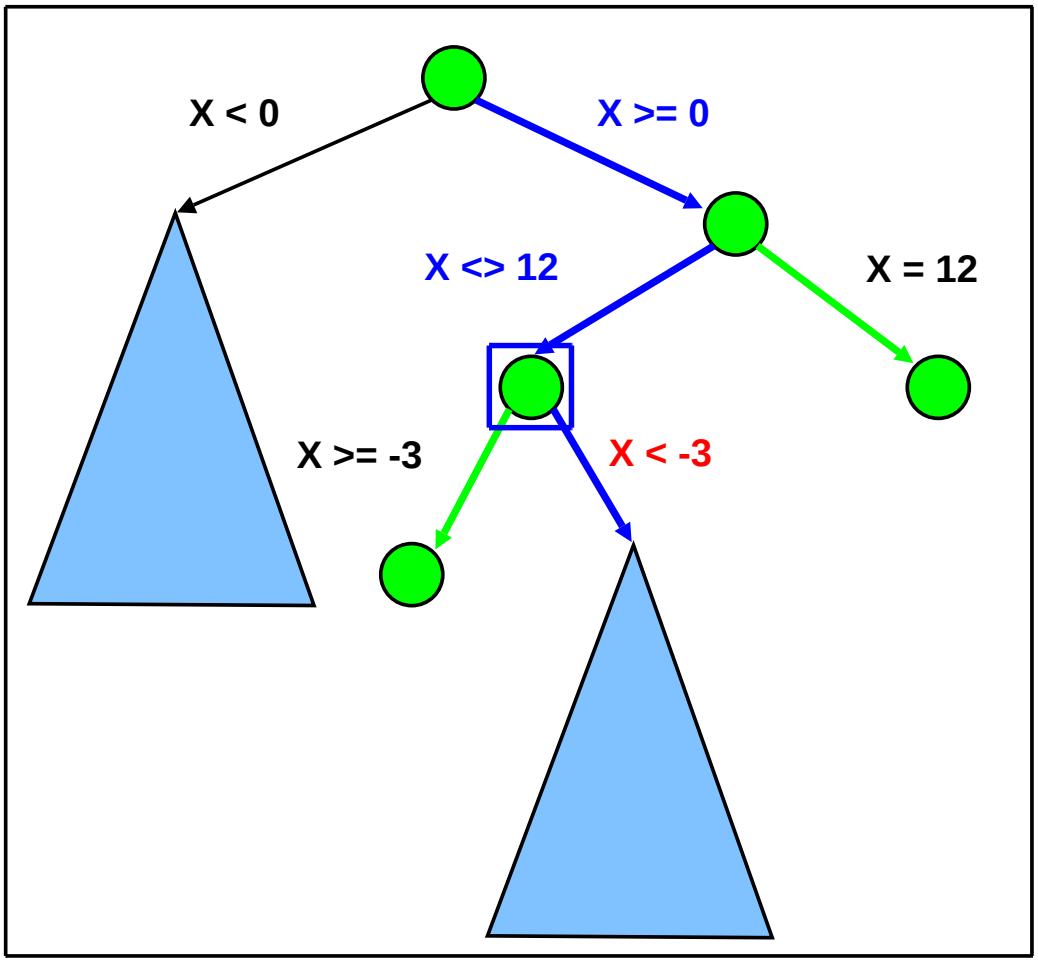
backtrack + résolution, unsat

Suivre seulement des chemins faisables



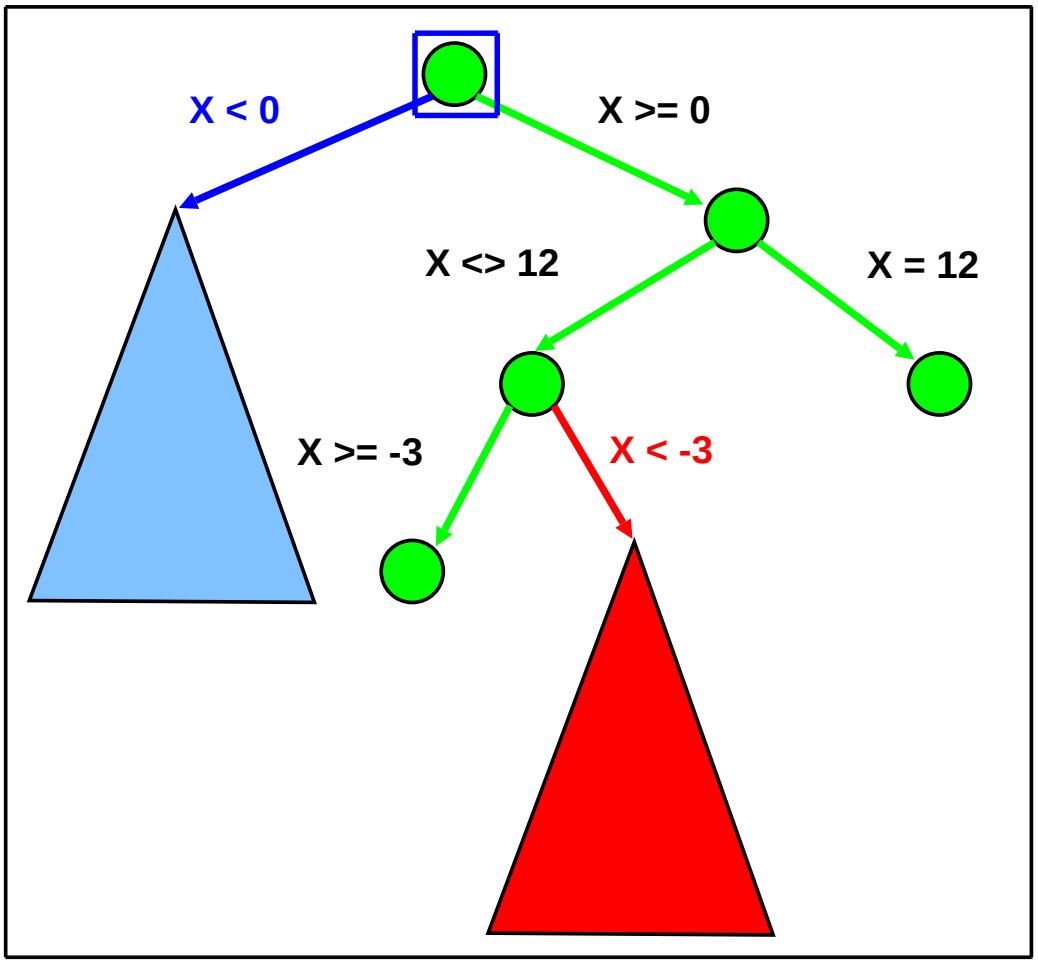
concret : X=12
backtrack + résolution, solution X = 5
concret : X=5
backtrack + résolution, unsat

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

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

Plan

Théories utilisées (1)

Fragment restreint : QF, seulement conjonction

Théories les plus utilisées

- arithmétique linéaire
- arithmétique (non linéaire) bornée
- vecteurs de bits

Ajouts utiles : tableaux, fonctions non interprétées

Théories utilisées (2)

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 2004-2009 : théories de + en + puissantes

Énumération de chemins (1)

PB1 : (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 (2)

PB2 : certains chemins sont redondants pour le critère de couverture choisi

Quelques solutions

- couper les chemins qui ne peuvent atteindre de nouvelles instructions (Osmose)
- couper les chemins "ressemblant" à un chemin couvert (Exe - RWset)
- voir PathCrawler

Énumération de chemins (3)

PB3 : explosion du #chemins due aux appels de fonction

Quelques solutions

- couper l'exploration à une certaine profondeur (Osmose, Java PathFinder)
- gestion paresseuse des fonctions (Sage)
- construction de résumés de fonctions (Dart)
- spécifications de fonctions (PathCrawler)

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

Bilan (subjectif) sur l'approche concolique

Points forts pour une utilisation industrielle

- efficace
- totalement automatisée si oracle automatique
- robuste aux "vrais" programmes
- correcte, résultats facilement vérifiables
- incrémental, s'insère dans pratiques de test existantes

Puissance maximale si couplée avec un langage de contrat

Quels domaines d'utilisation ?

- pb pour la certification : traçabilite DT - exigences
- ok pour tout le reste :-) [Pex livré dans Visual C#]

Plan

Génération automatique de DT de fonctions C/C++

Développé par le CEA depuis 2002

Entrées :

- Code source compilable (y compris fonctions appelées)
- Définition du contexte de test
- Programme oracle (facultatif)

Sorties :

- Données de test
- Chemin couvert par chaque test et chemins partiels infaisables
- Verdict (si oracle)

Vise 100% chemins faisables ou 100% branches atteignables

Limites à la couverture 100%

- Pour chaque chemin partiel, résolution de contraintes potentiellement NP complet
- Suffisamment efficace en pratique, mais résolution parfois coûteuse pour
  - flottants
  - éléments de structures de données à indice variable
  - contraintes non-linéaires
  - ... (difficile à caractériser)

Mis en place **timeout** : suppose chemin partiel infaisable

Aussi : constructions pas encore traitées (concrétisation ?)

Paramètres d'entrée d'une fonction C

Un sous-ensemble de

- paramètres formels et var. globales de type basique
- champs, éléments et déréférences accessibles via les autres paramètres formels et var. globales
- dimensions variables des structures de données en entrée
- données lues (d'un fichier) pendant l'exécution

Par défaut : tout

mais l'utilisateur peut en éliminer

Contexte de test

- paramètres effectifs d'entrée
- intervalle des valeurs de chaque paramètre effectif d'entrée
- intervalle des dimensions des structures de données
- la précondition sur les valeurs des paramètres effectifs en entrée pour
  - éviter les erreurs à l'exécution (ex. relation dimension de tableau et nombre éléments à traiter)
  - garantir un calcul correct (ex. entrée ≥ 0 pour $\sqrt{}$)
  - produire des tests pertinents, réalistes
- les paramètres de la stratégie de test

Comment exprimer et traiter la précondition ?

- en C
  - codée par l'utilisateur
  - retourne 0 ou 1
  - précondition directement traitée par PathCrawler
  - MAIS risque d'augmenter le #chemins
- directement en contraintes
  - difficulté : gestion de ces contraintes par PathCrawler
  - on a vite besoin de fragments + expressifs (disjonction, quantificateurs)

Stratégie de test

Explosion combinatoire de chemins à cause de

- boucles à nombre d'itérations variable :
  ne générer des tests que pour les chemins avec k itérations de ces boucles, où k est fixé par l'utilisateur (mais si un chemin n'est faisable qu'avec $k+1$ itérations ?)
- appels de fonction (test unitaire) : ne couvrir que les chemins de l'appelant
- trop de conditions indépendantes (if dans boucle, suite de if, etc.) : couverture branches

Problème : minimiser l'exploration de l'arbre des chemins (PB2)

Problèmes du traitement du C

- sémantique dépend compilation, machine cible : devraient êtres paramètres du contexte
- sémantique compliquée : partiellement simplifiée par CIL, bibliothèque open-source utilisée pour l'instrumentation du code source afin de récupérer la trace du chemin et la sémantique des blocs d'instructions
- flottants
- modèle mémoire
  - tableaux, pointeurs
  - alias
  - opérations sur les bits : traitées
  - casts de pointeur : non traitées
- appels de fonction bibliothèque

Traitement des flottants

- PathCrawler utilise le solveur de contraintes COLIBRI commun aux outils Osmose et GATeL
- basé sur la Programmation Logique par Contraintes
- traite domaines finis : entiers, flottants, etc.

- opérations arithmétiques des flottantes ont peu de bonnes propriétés (associativité, distributivité, etc.)
- mauvais comportements : absorption $X + \varepsilon = X$, annulation $X + \varepsilon - X = 0$
- solveur pour réels pas correct sur flottants

- COLIBRI traite arithmétique sur `double` IEEE 754
- mais sémantique sensible à la plateforme
- fonctions trigonométriques pas encore implantées

Traitement des tableaux, pointeurs

- représentation exacte tableaux
- contraintes spécialisées en cas d'indice variable
- traitement paresseux en cas de dimension variable :
  0 par défaut, incrément minimal lors référence élément du tableau

- représentation exacte déréférences (à plusieurs niveaux) pointeurs
- pointeurs représentés comme adresses tableaux
- pointeur NULL $\approx$ adresse tableau dimension 0

- suppose séparation tableaux et pointeurs en entrée
- sauf exceptions définies par la précondition
- alias locaux traités

Rappel calcul prédicat de chemin

Méthode de base : traverse chemin d'exécution

affectations : mettre à jour état symbolique, valeur des variables locales en termes des entrées

conditions : remplacer variables locales par valeur symbolique et ajouter au prédicat de chemin

Ex. chemin : `x >= 0 ; x = x + 3 ; x == 15...`

instruction	prédicat	état symbolique
`x >= 0`	$X \geq 0$	
`x = x + 3`		$x = X + 3$
`x = = 15`	$(X + 3) = 15$	

Alias local : 1er exemple

alias : noms différents pour même location mémoire

```
void f(int x, int tab[]){
  int *pt ;
  pt=&tab[0] ;
  *(pt+4)=5 ;                          /* tab[4] modifié ici */
  if (tab[4] < x)          /* pas TAB[4] < X mais 5 < X */
```

ne pose pas de problème : car état symbolique inclut pointeurs

mais traitement plus compliqué si
comparaison des noms dépend valeurs des entrées

Alias local : 2ème exemple

```
void f(int x, int y, int z, int tab[]){
  tab[2] = x;
  tab[x + 2] = z;
  if (tab[y] > 5)      /* vrai si ... */
```


	$(2 + X) = Y$		$\wedge\ Z > 5$
$\vee$	$(2 + X) \neq Y$	$\wedge\ Y = 2$	$\wedge\ X > 5$
$\vee$	$(2 + X) \neq Y$	$\wedge\ Y \neq 2$	$\wedge\ TAB[Y] > 5$
	relation d'alias		condition chemin

Les alias transforment le prédicat de chemin en arbre de disjonctions

Pour minimiser #tests, traitement paresseux des alias

Traitement des appels de fonction bibliothèque

- traiter la spécification de la fonction au lieu du code source
- traduction spécification en contraintes
- domaines fonctionnels au lieu chemins faisables d'exécution
- exploration paresseuse pour ne couvrir que les chemins de l'appelant
- stratégie boîte grise

Nouvelle application : WCET

- génération de cas de test pour mesure pire temps d'exécution (WCET)
- complémentarité outils analyse statique donnant sur-approximation
- pas besoin modèles de l'architecture des microprocesseurs (nombreux, en évolution constante, protégés par secret industriel)
- supposer (ou s'assurer) 1 chemin = 1 temps d'exécution
- mesurer temps d'exécution effectif des chemins sur la cible
- trop de tests :
  - ordres partiels entre chemins similaires correspondant (sous conditions) à la relation des temps d'exécution :
    - branche vide de if-then-else plus rapide que branche pleine
    - boucle avec n itérations identiques plus rapide que même boucle avec $n+1$ itérations identiques aux autres
  - modifier la stratégie pour minimiser les tests de chemins non-maximaux dans ordre partiel (PB2)
  - mais garantit couverture tous les chemins maximaux $\Rightarrow$ mesure temps d'exécution le plus long

Feuille de route

- PB2 : optimisation exploration arbre des chemins
  - améliorer traitement code source fonctions appelées
  - améliorer traitement boucles
  - ...
- meilleure modèle mémoire (cast de pointeurs)
- traitement précondition en langage de spécification
- complémentarité test et analyse statique code C
- couverture structurelle de domaines fonctionnels
- test structurel des systèmes cycliques réactifs
- test structurel de modèles systemC
- mise en ligne comme serveur de tests (dans 1 an)

Plan

Osmose en bref

Génération automatique de DT à partir de codes binaires bruts

Coopération CEA-EDF depuis 2006

- besoin : analyse de vieux programmes ou de COTS

Entrées : code binaire, nom de l'architecture, environnement, objectif de test

Sorties : jeu de tests, couverture atteinte, modèle du programme

Motivations de l'analyse de code binaire

Pas de code source de haut niveau disponible

- Composants sur étagère
- Programmes anciens toujours en utilisation
- Certification par un tiers
- Code mobile (applets, virus, etc.)

Se prémunir contre les incertitudes liées à la compilation

- instructions à la sémantique mal définie
- les compilateurs peuvent être bogués
- impact des optimisations (sécurité, sûreté)

Propriétés non fonctionnelles

- pire temps d'exécution, taille de pile, etc.

Difficultés techniques

(1) Sémantique bas niveau des données

- arithmétique machine + flags
- instructions sur les séquences de bits

(2) Sémantique bas niveau du contrôle

- mécanismes bas niveau pour prochaine instruction et fonctions
- pas de distinction claire entre contrôle et données

(3) Diversité des jeux d'instruction et architectures

Conséquence de (2) : pas de flot de contrôle a priori

- transitions, boucles, fonctions doivent être retrouvées
- problème de reconstruction de modèles (IR recovery)
- problème transversal

Technologies clés

Support multi-architecture

- Machine Binaire Abstraite (MBA)
- Deux pbs distincts : retrouver le MBA, analyser le MBA

Exécution concolique

- sémantique bas niveau sur les données (théorie BV)
- sémantique bas niveau sur le contrôle (`PC` et `SP`)
- reconstruction de modèle en dynamique et en symbolique

Solveur sur la théorie des vecteurs de bits (BV)

- basé sur la CLP et COLIBRI
- arith. modulaire + flags, opérations bit à bit, etc.

Feuille de route de Osmose

État actuel

- multi-architecture : PPC 32 bit, Motorola 6800, Intel 8051
- solveur propre
- plusieurs optimisations pour réduire #chemins
- quelques expérimentations sur code industriel
- publications : [ICST 2008] [ICST 2009]

Travaux en cours / à venir

- solveur : passerelle vers des solveurs classiques (STP)
- solveur : révision complète de notre solveur
- révision des MBAs
- entrées-sorties / interfaces
- ANR BinCoA (2009-2012)
- passage en license libre ?

Conclusion : bilan approche concolique (bis)

Points forts pour une utilisation industrielle

- efficace
- totalement automatisée si oracle automatique
- robuste aux “vrais” programmes
- correcte, résultats facilement vérifiables
- incrémental, s’insère dans pratiques de test existantes

Puissance maximale si couplée avec un langage de contrat

Quels domaines d’utilisation ?

- pb pour la certification : traçabilite DT - exigences
- ok pour tout le reste :-) [Pex livré dans Visual C#]