Test Structurel Aléatoire et Détection de Chemins Infaisables

Romain Aïssat, Marie-Claude Gaudel, Frédéric Voisin

TestClub, 29/03/2018

Test structurel : chemins infaisables

Test structurel :

- basé sur la notion de "critère de couverture" d'éléments structurels du programme
- en pratique : utilisé souvent pour valider ou compléter un jeu de test existant
- ici, utilisé pour de la génération de jeux de tests

Génération des jeux de test :

1. sélection d'un ensemble de chemins,
2. génération des valeurs de test **à partir des chemins faisables**.

Existence des chemins infaisables : impact négatif sur

- test structurel : chemins infaisables inutilisables,
- toute technique basée sur le CFG : analyse statique, optimisation de code, model checking, ...

Sélection aléatoire de chemins

Objectif : échantillon de n chemins parmi les N_l de longueur maximale l.

Sélection aléatoire des chemins : selon une distribution totale et uniforme sur les chemins de longueur maximale l, par marches aléatoires uniformes (Denise et al., *Coverage-biased random exploration of large models and application to testing*, 2011).

Avantages des marches aléatoires uniformes :

1. tous les chemins ont une même chance non-nulle d'être tirés,
2. diversité des ensembles de chemins,
3. on peut évaluer la "qualité d'un jeu de test" (nombre de tests à effectuer pour s'approcher de la couverture fixée avec une probabilité donnée)

Défaut majeur :

- pas de distinction entre faisables et infaisables durant les tirages,
- généralement : rapport faisables/infaisables **très** faible.
- empiriquement : la faisabilité diminue avec la longueur des chemins.

La librairie Rukia (Johan Oudinet)

Bibliothèque C++ disponible au-dessus de la librairie Boost

Disponible : http ://rukia.lri.fr/en/index.html

- conçue pour le tirage uniforme de chemins de longueur bornée dans de très grands graphes
- à base de méthodes de comptage : une version "récursive", une version "dichotomique"
- efficacité : chemins longs (plusieurs milliers de transitions) dans des graphes à plus de 10^9 sommets (Oudinet, *Approches combinatoires pour le test statistique à grande échelle*, 2010).
- méthode "détournée" pour permettre la couverture d'autres éléments structurels : arcs, sommets
- *possibilité d'intervenir sur la table de comptage (élimination de préfixes infaisables)*

Détection de chemins infaisables

Objectif : diminuer l'impact des infaisables dans le contexte du test structurel aléatoire.

Détection de chemins infaisables :

- **indécidable** : impossible de calculer l'ens. des faisables,
- $\rightarrow$ utilisation d'heuristiques.

Approches naïves :

1. prise en compte des infaisables *a posteriori* : inefficace (Gouraud, *Utilisation des structures combinatoires pour le test statistique*, 2004),
2. arbre des chemins faisables : impossible en pratique lorsqu'on considère des chemins longs.

Solution proposée :

- en amont de la phase de sélection,
- éliminer des chemins infaisables du CFG, par "dépliage" borné du CFG
- transformation du CFG initial :
  - plus grand, plus détaillé,
  - contenant moins de chemins infaisables.
- tirages dans le nouveau CFG.

Contrainte : conserver **tous** les chemins faisables, quelle que soit leur longueur

Opérateurs de transformation de graphes

Algorithme de transformation de graphes (Aïssat) : basé sur 5 mécanismes

1. exécution symbolique de tous les chemins,
2. résolution de contraintes (*Solveurs SMT*)
3. détection de subsomptions (*restreinte actuellement aux entrées de boucles*)
4. abstraction (+ renforcements pour les contrôler),
5. raffinement par contre-exemple.

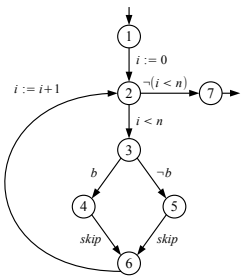
Entrées : CFG + pré-condition de départ

Sortie : nouveau CFG.

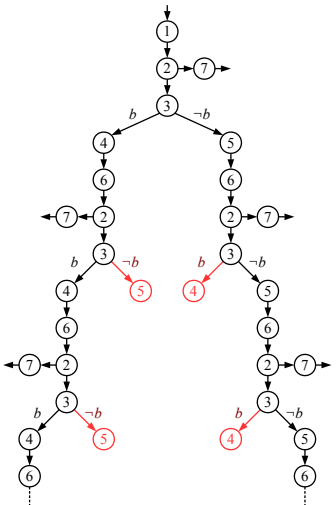
Inspirations :

- exécution symbolique + boucles non-bornées :
  - Jaffar et al., *Unbounded symbolic execution for program verification*, 2012,
  - McMillan, *Lazy annotation for program testing and verification*, 2010.
- systèmes à la CEGAR : "abstract-check-refine"
  - Henzinger et al., *Lazy Abstraction*, 2002,
  - Clarke et al., *SATABS : SAT-Based predicate abstraction for ANSI-C*, 2005,
  - Ivancic et al., *F-Soft : Software verification platform*, 2005,
  - Beyer et al., *The software model checker Blast*, 2007,
  - Grebenshchikov et al., *Synthesizing software verifiers from proof rules*, 2012.

Exécution symbolique et boucles non-bornées

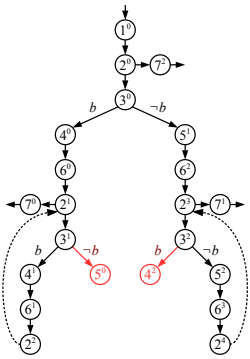


CFG de `foo`
(b ne change pas dans la boucle).

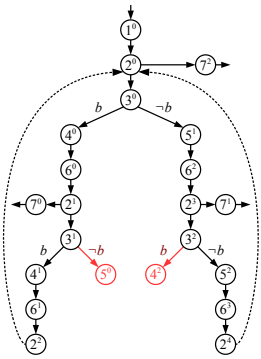


Arbre d'exécution symbolique infini de `foo`.

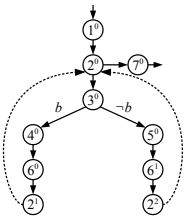
Subsomption : ça ne vient pas spontanément



Graphe d'exécution symbolique idéal.
Ne contient plus d'infaisables

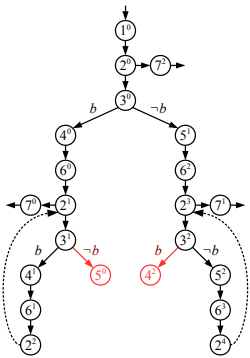


Liens de subsomption «imprécis».
Contient encore des infaisables.

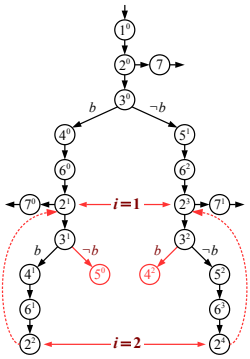


Liens de subsomption triviaux.
Equivalent au graphe initial.

Subsomption : ça ne vient pas spontanément



Graphe d'exécution symbolique idéal.



Pas de subsomptions «naturelles»
(il faut abstraire i).

Forcer les subsomptions : abstraction

Définition (Abstraction)

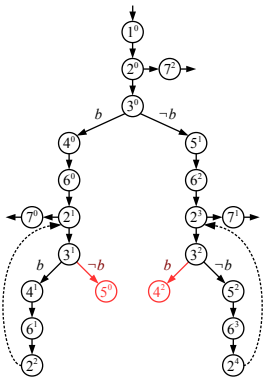
Soient c et c_a deux configurations. c_a est une abstraction de c si $c \sqsubseteq c_a$.

Calcul des abstractions :

- affaiblissement de l'influence du prédicat de chemin,
- synthèse d'invariants («locaux» et faibles),
- plusieurs méthodes/résultats
- «meilleure» abstraction : impossible en pratique,
- perte d'informations, donc de pouvoir de détection de chemins infaisables.

Contrôle des abstractions :

- raffinement par contre-exemples,
- contre-exemples : infaisables rendus faisables par abstraction,
- calcul de conditions limitantes,
- peut empêcher la terminaison : dépliage infini.



Graphe d'exécution symbolique idéal.

Forcer les subsomptions : abstraction

Définition (Abstraction)

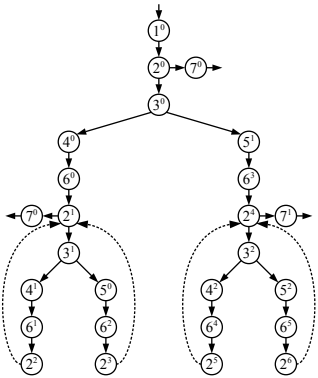
Soient c et c_a deux configurations. c_a est une abstraction de c si $c \sqsubseteq c_a$.

Calcul des abstractions :

- affaiblissement de l'influence du prédicat de chemin,
- synthèse d'invariants («locaux» et faibles),
- plusieurs méthodes/résultats
- «meilleure» abstraction : impossible en pratique,
- perte d'informations, donc de pouvoir de détection de chemins infaisables.

Contrôle des abstractions :

- raffinement par contre-exemples,
- contre-exemples : infaisables rendus faisables par abstraction,
- calcul de conditions limitantes,
- peut empêcher la terminaison : dépliage infini.



Abstractions triviales (abstraction de i **et** b).

- difficulté de formuler les théorèmes voulus,
- complexité/niveau de détails de la preuve.

Objectifs :

- monotonie de l'exécution symbolique : «inutile de construire les desc. d'un subsumé»,
- correction de l'approche : «chemins du nouveau CFG réalisent les mêmes calculs»,
- préservation des faisables : «aucun comportement perdu».

Des idées de la preuve se retrouvent dans le prototype

Il reste du travail à faire dans la formalisation si on veut étendre le langage d'entrée

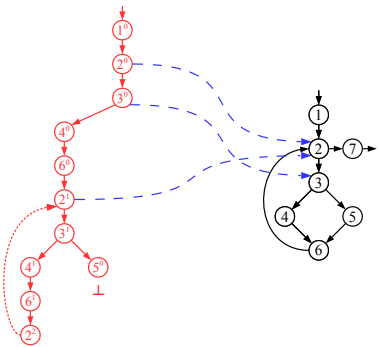
Formalisation : opérateurs de transformation de graphes

Graphes rouges-noirs :

- partie noire : CFG d'entrée,
- partie rouge : copie par dépliage partiel de la partie noire décoré par des :
  - configurations,
  - liens de subsomptions,
  - marques d'infaisabilité,
  - conditions limitantes,
- structure spécifique à nos travaux.

5 opérations élémentaires :

1. ajout d'une arête rouge,
2. marquage d'un sommet rouge infaisable,
3. ajout d'un lien subsomption,
4. abstraction,
5. renforcement par condition limitante.



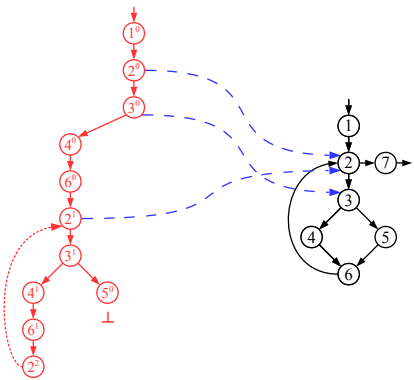
Un graphe rouge-noir.

Heuristiques non modélisées : formalisation valable pour toute une famille d'algorithmes.

CFG final

Applications :

1. Quid si plus de connexion entre parties rouge et noire ?
2. De même, si on tire des chemins uniquement dans la partie rouge ?
3. Mais rien n'empêche de tirer des chemins de longueur arbitraire : aucun chemin faisable du CFG de départ n'a été perdu.
4. : à voir lien avec la "qualité de test" calculée ?



Un graphe rouge-noir.

Théorèmes de correction

Théorème (Correction)

Les chemins rouges-noirs d'un graphe rouge-noir rb sont des chemins noirs de rb.

Théorème (Préservation des chemins faisables)

Les chemins noirs faisables depuis la configuration initiale d'un graphe rouge-noir rb sont des chemins rouge-noirs de rb.

Théorème (Relation entre sommets de la partie rouge)

Soit p un chemin rouge d'un graphe rouge-noir rb. La configuration en queue de p subsume la configuration obtenue par exécution symbolique de la trace de p depuis la configuration à la racine de rb.

De la formalisation à une implémentation

Implémentation :

- sous forme d'un prototype (Ocaml, 3k loc),
- résolution de contraintes : solveur SMT (Z3).

Basée sur la formalisation : graphe rouge-noir + 5 opérations élémentaires.

Mais plusieurs choix à faire (heuristiques) :

- ordonnancement des 5 opérations élémentaires,
- sélection des liens de subsomption,
- calcul des abstractions,
- sélection des contre-exemples,
- calcul des conditions limitantes.

Publication : *Software Quality, Reliability and Security 16*.

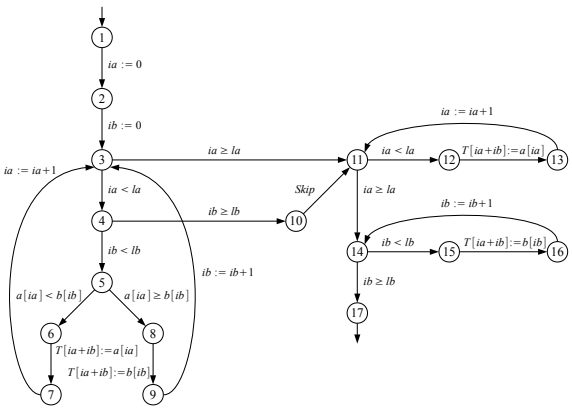
Fusion de tableaux triés

Version limité des tableaux : sans chevauchement avec d'autres variables.

En pratique : version légèrement simplifiée du programme

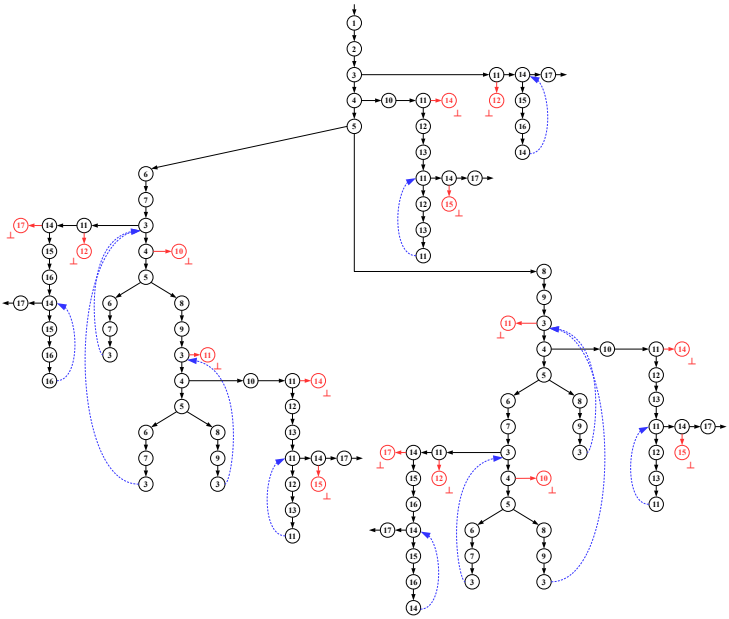
```
Fonction fusion(int[] a, int[] b, int la, int lb, int[] T)
1   ia = 0;
2   ib = 0;
3   tant que ia < la faire
4       si ib < lb alors
5           si a[ia] < b[ib] alors
6               T[ia + ib] ← a[ia];
7               ia ← ia + 1;
            sinon
8               T[ia + ib] ← b[ib];
9               ib ← ib + 1;
        sinon
10          break;
11  tant que ia < la faire
12      T[ia + ib] ← a[ia];
13      ia ← ia + 1;
14  tant que ib < lb faire
15      T[ia + ib] ← b[ib];
16      ib ← ib + 1;
```

Programme `fusion`.



Graphe de flot de contrôle de `fusion`.

Fusion de tableaux triés : son graphe rouge-noir final



Graphe $\mathcal{G}'$ pour $a = 1$ et $k = 2$. Ne contient plus de chemins infaisables.

Fusion de tableaux triés : son graphe rouge-noir final

Paramètres :

- a : méthode d'abstraction (1 : suppr. de contraintes, 2 : màj de variables),
- r : raffinements (✓ : activés),
- k : comparaison des k-faisables.

Colonnes

- $|S|$: nombre de sommets,
- $|A|$: nombre de transitions,
- t : temps de construction de $\mathcal{G}'$ en secondes,
- l : longueur des chemins,
- C : nombre de chemins,
- CF : nombre de chemins faisables.

	a	r	k	$\lvert S\rvert$	$\lvert A\rvert$	t	$l \leq 30$		$l \leq 50$		$l \leq 100$	
							C	CF	C	CF	C	CF
$\mathcal{G}$				17	21		593	82	11 728	1 351	12 389 030	1 385 616
$\mathcal{G}'$	1	✓	0	23	26	3	210		3 694		3 819 743	
	2	✓				2.9						
	1	✓	2	**85**	**96**	**15.7**	**82**		**1 351**		**1 385 616**	
	2	✓		**170**	**178/19**	**117.7**	**82**		**2 728**		**11 593 094**	
	2			**127**	**144**	**9.3**	**131**		**2 031**		**1 980 403**	

Tri à bulles : chemins faisables vs infaisables

Tri à bulles : deux boucles imbriquées.

Propriété des chemins faisables : même nombre d'itérations de la boucle interne à chaque passage dans la boucle externe.

Conséquence : l'ensemble des chemins faisables pour le tri à bulles ne forme pas un langage rationnel.

Cas favorable : réduction du nombre de chemins par un facteur 8, 22, 270, pour $l = 30$, 50 et 100.
Lookahead : lié à la taille du corps de boucle.

	a	r	k	$\lvert S\rvert$	$\lvert A\rvert$	t	$l=30$		$l=50$		$l=100$
							C	CF	C	CF	C
$\mathcal{G}$				12	14		494		76 625		2.4×10^{10}
$\mathcal{G}'$	1	✓	0	16	18	2.7	493		76 624		2.4×10^{10}
	2	✓				3.7					
	1	✓	2	40	48	7.2	337		52 171		1.7×10^{10}
	2	✓		22	25	6.7	60	13	3 530	82	1×10^{8}
	1	✓	8	73	84	48	121		9 569		6×10^{8}
	2	✓	4	180	193/13	116.6	26		3 316		9.2×10^{8}
				35	39	7	60		3 530		1×10^{8}
			8	**76**	**85**	**23.9**	**60**		**3 408**		$\mathbf{8.8 \times 10^{7}}$

Substring : chemins faisables vs infaisables

Substring : deux boucles imbriquées.

Langage des chemins faisables : non-rationnel.

Cas favorable : réduction du nombre de chemins par un facteur 14, 90, 7000, pour $l = 30$, 50 et 100.

	a	r	k	$\lvert S\rvert$	$\lvert A\rvert$	t	$l \leq 30$		$l \leq 50$		$l \leq 100$
							C	CF	C	CF	C
$\mathcal{G}$				12	15		789	57	85 598	854	1×10^{10}
$\mathcal{G}'$	1	✓	0	14	15	1.1	88		1 660		2 612 181
	2	✓				1.1					
	1	✓	11	42	46	16.5	80		1 520		2 398 239
	2	✓		92	102	63.3	61		1 125		1 765 110
	1	✓	14	118	130	153.7	71		1 342		2 123 382
	2	✓		234	262	317.4	57		949		1 468 171

Perspectives (à moyen terme)

Recherche d'heuristiques plus fines

- définitions alternatives pour la subsomption,
- méthodes d'abstraction,
- calcul des conditions limitantes,
- apprentissage à partir des conditions limitantes.

Extension du langage d'entrée : formalisation en Isabelle et implémentation

- utilisation d'un modèle mémoire pour traiter les pointeurs,
- appels de fonctions.

Analyse de flot de données/interprétation abstraite : à investiguer.

Découverte d'invariants locaux : à investiguer.

Extension au test d'intégration : à investiguer ("qualité de test" globale en fonction de celle des composants")

Perspectives (à plus court terme)

- validation expérimentale sur d'autres exemples
- meilleure prédictibilité du paramétrage des heuristiques
- consolidation et optimisation
- élimination au vol d'infaisables dans la table de comptage de Rukia
- intégration à Frama-C (faciliter les expérimentations / utiliser les outils d'analyse statique / modèles mémoire disponibles)
- extensions au langage d'entrée actuel

Validation expérimentale

- Suite de tests “SIEMENS” : ensemble de programmes avec leurs tests dont deux semblent plus intéressants pour nous
  - *tcas* : programme sans boucle, nombreux opérateurs logiques “paresseux”
  - *replace* : expressions régulières. Pas encore traitable (appels récursifs, pointeurs et paramètres passés par référence)
- Validation du tirage aléatoire uniforme dans Rukia
- Exemples tirés de PathCrawler (CEA, approche de type “concolic”)
- Autres benchmarks : à discuter.
  - Beaucoup d’exemples intéressants ne sont pas encore traitables.
  - La “confusion”en C entre tableaux et pointeurs ne facilite pas non plus les choses
  - Tous les exemples ne sont pas forcément intéressants pour notre approche :
    Partie algorithmique très simple : peu de chemins infaisables
    Découpage en de nombreuses petites fonctions : peu de test unitaire.

Tcas

- La documentation indique au moins un chemin infaisable
- Pas de boucle, la plupart des fonctions se résument à une expression. Une partie initialisation (arguments passés au programme) puis appel à la "fonction d'intérêt"
- Nombreux opérateurs logiques paresseux dans les fonctions : combinatoire intéressante
- Les fonctions ont été "in-linées" dans la fonction d'intérêt
- le programme est traité par le pré-processeur de Frama-C puis traduit (à la main) dans le langage d'entrée du prototype

Résultats :

- Traité par évaluation symbolique pure : pas de subsomption ou abstraction
- **Avec un dépliage optimal** ($l = 47$, longueur maximale d'un chemin)
  - graphe entièrement "rouge" (1675 arcs, 1527 sommets). Plus aucun chemin infaisable
  - 123 chemins faisables sur un total de 179720 ($l = 47$) ou 181512 ($l = 50$) dans le graphe intial
  - tirage uniforme de 60 chemins ($l = 50$) dans le graphe initial : 0 ou 1 chemin faisable
- **Avec un dépliage non optimal :** $l = 40$ et tirage de chemins de longueur 50 :
  - la partie noire ne disparait pas ; 386 chemins dont 123 faisables
  - 3 tirage de 60 chemins dans le graphe rouge-noir : 38, 42, 38 infaisables.

“binary search” borné

- Extrait de la galerie “PathCrawler on line”
- Recherche dichotomique dans laquelle on ne s’arrête pas dès qu’on a trouvé l’élément
- Nombreux chemins infaisables
- Tableau avec une dimension fixée à 10 dans la librairie
- pas forcément utilisé avec la précondition que le tableau est trié !
  Les expériences ici sont faites sans la précondition.
- L’ajout de préconditions diminue le nombre de chemins faisables (PathCrawler : de 82 dont 23 faisables à 76 dont 20 faisables).
- un chemin partiel de PathCrawler peut correspondre à plusieurs chemins chez nous (en cas de préfixe infaisable)

Résultats :

- Avec un dépliage optimal ($l = 27$)
  - déplié automatiquement en un arbre d’exécution symbolique
  - Graphe purement “rouge” (23 chemins, 287 sommets, 202 arcs)
  - 159 chemins dans le graphe initial (PathCrawler : 59 préfixes infaisables)
  - graphe initial : sur 23 chemins tirés, entre 19 et 20 infaisables

L'exemple "binary search" non borné

- le même que précédemment mais sans fixer la dimension du tableau
- On remplace la valeur de la borne par l'assertion $low < high$
- Testé avec un dépliage de 30

Résultats :

- la partie noire ne disparait pas entièrement
- pas (actuellement) possible de conserver l'information que si $found$ passe à $true$, sa valeur ne changera plus.
  - on tire des chemins de longueur 30 :

    Graphe rouge-noir : 1026 arcs, 800 sommets, 53 chemins de longueur 30, tous faisables

    Graphe initial : 271 chemins de longueur 30, 83 infaisables sur 100 tirés.
  - on tire des chemins de longueur 50 :

    Graphe rouge-noir : 1026 arcs, 800 sommets, 8148 chemins de longueur 50, 66 infaisables sur 100 tirés.

    Graphe initial : 21247 chemins de longueur 50, 83 infaisables sur 100 tirés.