Projet de Vérification Logicielle

Léo BARRÉ, Elliott BLOT

Décembre 2016

Introduction

Le but de ce projet et de réaliser un programme permettant de vérifier si un programme ce termine correctement. Pour cela nous allons explorer le CFA pour obtenir une formule, qui sera ensuite vérifié par le module $z3$, qui est un SMT-solveur (pour vérifier si on peut atteindre un mauvais état). Pour cela, nous avons du implémenter deux modules, Semantics.ml et BoundedModelChecking.ml que nous allons présenter dans la suite de ce rapport.

Semantics.ml

L'objectif de ce module est de transformer une opération, composé assignations, d'inégalités, d'égalités, et de skips, en une entrée valide de $z3$ pour pouvoir être testée plus tard dans le programme.

Le fichier a été, bien sur, séparé en plusieurs fonctions:

- *expr*, une fonction récursive, qui s'occupe de transformer les expressions en éléments de z3. Elle est récursive simplement car une expression peut contenir d'autre expressions.
- *guard*, qui s'occupe de toute les égalité et inégalité (et appelle *expr* pour les expression).
- *op*, la fonction principale, qui s'occupe de transformer les opérations en éléments de z3.

Quant à la variable div_plus, elle n'est là que pour traiter le cas particulier des divisions. En effet, celles-ci requièrent également un test pour vérifier que le dénominateur n'est pas nul, en plus de implémentassions normale de la division elle-même, et celui-ci doit être placé à l'origine.

BoundedModelChecking.ml

Ce module est le module principale, c'est à dire celui qui contient l'algorithme permettant d'effectuer la vérification.

Le principe de ce type de vérification est d'essayer tous les chemins possibles dans la représentation automate du programme, puis de vérifier si l'un d'entre eux nous mène jusqu'au noeud incriminé.

Pour ça nous avons donc eu l'idée de créer une fonction *_search*, récursive, qui s'occuperait de visiter tous les noeuds successeurs à partir d'un cité en arguments.

La fonction *search*, elle, se contenterait de faire référence à celle-ci, en initialisant au premier noeud de l'automate.

Le fonctionnement de *_search* est assez simple :

1. On vérifie que le chemin est atteignable avec un SMT-solver.
2. Si ça ne l'est pas, on renvoie un $true$.
3. Sinon, si on se trouve sur le mauvais noeud, on renvoie un chemin vide.
4. Sinon, on vérifie si on a atteint *bound*, auquel cas on renvoie $false$.
5. Sinon, on itère au niveau des successeurs du noeud actuel des appels à $\_search$ et on récolte leurs résultats
6. Si l'un d'entre eux trouve un chemin faisable, on renvoie celui-ci, auquel on ajoute le chemin pour atteindre ledit successeur,
7. Sinon, si l'un d'entre se heurte à la limite *bound*, on renvoie $false$
8. Sinon $true$

Nous avons également ajouté 3 tests pour vérifier quelques points :

- $initial\_is\_final.aut$, dont le but est de voir la réaction du programme si le noeud initial est également le noeud final
- $problem\_at\_10.aut$, qui donne un chemin de taille exactement 10 (grâce au parcours en profondeur)
- $no\_detection\_strangely.aut$, qui devrait visiblement déclencher une détection de chemin faisable, mais ne le fait pas bizarrement (c'est une simple incrémentation d'une variable, et un test vérifiant s'il dépasse une constante inférieure à 10)