Univesité de Bordeaux
Sciences & Technologie
351 Cours de la Liberation
33400 Talence

- Projet Vérification Logiciel -

Software Verification Project

Amelie GUEMON

Master 2 CSI — Cryptologie & Sécurité Informatique

18 décembre 2016

1 Semantics

1.1 Implémentation choisie

Le module *Semantics* est découpé en plusieurs fonctions :

- **matching_expression** pour parser les *expressions* ;
- **matching_predicate** pour parser les *predicates* ;
- **assign**, **keepargs** & **keepargs_except** pour la conservation des valeurs des variables ;
- **formula**, point d'entrée du module *Semantics* qui se chargera des *assignations*, des *skips* et des *guards*.

L'implémentation de la division par 0 aura entraîné une refonte de l'architecture du module. En effet, celle-ci crée une restriction supplémentaire à prendre en compte : les opérations en dénominateur doivent être différentes de 0.
Pour ne pas perdre d'informations, les retours de fonctions ne seront plus de la forme *Z3.Expr.expr* mais des tuples de la forme (*Z3.Expr.expr*, *Z3.Expr.expr list*). Le deuxième paramètre représentera les contraintes sur les divisions.

Aussi, concernant la division, on peut noter que les simplifications (comme le produit en croix pour les divisions) lors de la construction de la séquence d'opérations ne sont pas implémentées.

1.2 Tests ajoutés

Les tests ajoutés pour la partie *Semantics* sont présents dans le fichier *Test_Semantics.ml*. Ceux-ci sont basés sur ceux déjà présents et ne font que tester les combinaisons d'*Opérations*. Les tests sont effectués automatiquement à l'aide de la commande **make test** et, afin des les distinguer de ceux déjà fournis, leur nom est précédé du terme *"supp_"*.

Quelques tests implémentés pour les *assign* :

$$z = \frac{x+3}{y},\ z = \frac{x+3}{\frac{y}{z-4}}$$

Ainsi que pour les *guard* :

$$\frac{x+5}{z} > \frac{y}{z},\ \frac{x-5}{z} < \frac{y}{z+y}$$

Bien évidemment, aucun nouveau test n'a été effectué pour la partie *skip*.

2 Bounded Model Checking

2.1 Implémentation choisie

Au vu des tests effectués, l'implémentation effectuée semble fonctionner, bien que celle-ci ne respecte totalement le paradigme fonctionnel : pour y remédier, il faudrait restructurer la fonction *dfs*.

Le module se compose donc de 2 fonctions : *search* et *dfs*. *Search* est la fonction principale et le point d'entrée de ce module. Il ne prend en entrée que l'automate et la profondeur maximale désirée pour le parcourir. Quand à la fonction *dfs*, celle-ci représente le coeur du module : c'est dans cette fonction que tout va se passer.

La fonction *dfs* peut donc se découper en plusieurs parties :

1. Test de la profondeur actuel dans l'automate entier ;
2. Définition du noeud actuel ;
3. Définition du *result* actuel, en fonction du type de node et de la réponse donnée par la résolution de la formule dans le solveur ;
4. Partie récursive de la fonction. Va appliquer une fonction (définie à la volée) à l'ensemble des couples (transitions, noeud suivant) succédant le noeud actuel, et ce, à l'aide d'un **List.fold_left**. Cette fonction, en fonction du résultat du sous-arbre précédemment parcouru, empilera la transition du tuple sur lequel elle s'applique puis appellera récursivement *dfs* afin de connaître le type de *result* des sous-arbres issus de la node étudiée et d'en renvoyer un résultat associé.

Afin de garder trace du chemin effectué à chacun des parcours, une liste est passée à chaque fois en paramètre. Celle-ci contiendra toujours, en en-tête, un couple représentant la dernière transition effectuée et le noeud actuel. Aussi, il est important de différencier les **Empty true** des **Empty false** : c'est pour cela que l'astuce d'un & booléen sera utilisé.

2.2 Tests ajoutés

Les tests ajoutés se situent dans le dossier **examples**, avec ceux déjà fournis. Le script **.test_all** applique l'exécutable sur tous les tests présents dans ce dossier. Mais les résultats n'ayant pas été rentrés à la main, on ne peut vérifier, à la volée, la véracité des résultats obtenus sur chacun des tests. Il reste donc la vérification automatique pour chacun des tests à implémenter.

En ce qui concerne les tests en eux-mêmes, les noms des tests supplémentaires sont précédés de *supp_*. Voici une liste de ceux qui ont été ajoutés :

- **final_init** : représente un automate n'ayant qu'un seul état, initial et final, sans aucune transition ;
- **loop_no_sat** : représente présentant une initialisation puis une boucle sur lui-même sans aucune opération (*skip*) et un test impossible à réaliser pour atteindre l'état final ;
- **neg** : représente un automate décrémentant infiniment dont la valeur x commence négativement et où le test pour atteindre l'état final est une égalité à 0 ;
- **no_final** : représente un automate où l'état final n'est pas atteignable : aucune transition ne le rejoint ;
- **over_bound** : représente un automate nécessitant 11 étapes afin d'atteindre l'état final ;
- **simple_final_loop** : représente un automate simple mais avec une boucle sur l'état final ;
- **under_bound** : représente un automate nécessitant 10 étapes afin d'atteindre l'état final.

Vous pouvez retrouver les représentations de ces graphes en annexes.

Certains de ces tests ont des utilités spécifiques.
Ainsi, les tests **over_bound** et **under_bound** permettent de tester précisément la partie de détection de la profondeur maximale. Il est important de noter que bien que la valeur de x soit initialisée à 8, il faut compter 8 décrémentation, 1 transition pour sortir de l'état initial et 1 autre pour arriver à l'état final.
De même, le test **simple_final_loop** permet de vérifier que le chemin renvoyé est bien le premier chemin trouvé, et ce, grâce à la boucle sur l'état final.
Enfin, le test **final_init** permet de vérifier que même dans le cas particulier d'un état initial et final à la fois, on retrouve bien un chemin (vide). Tout comme le test **no_final** qui ne peut renvoyer de chemin car l'état final n'est pas atteignable. Ce dernier aura aussi été utilisé pour vérifier que le bdm essaie bien tous les chemins possibles, en affichant les diffèrents états du solveur à chacune des étapes avec *"Format.printf ¨[Solver state %s].(Solver.to_string solv)"*.

3 Annexes

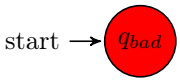

FIGURE 1 – supp_final_init

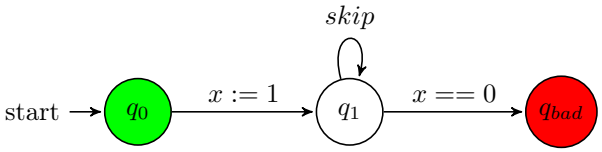

FIGURE 2 – supp_loop_not_sat

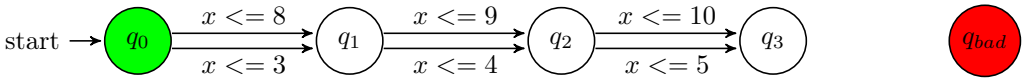

FIGURE 3 – supp_no_final

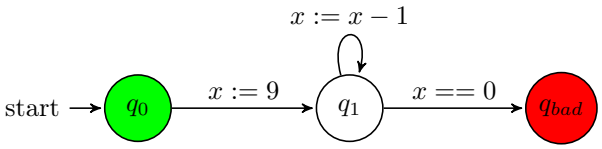

FIGURE 4 – supp_over_bound

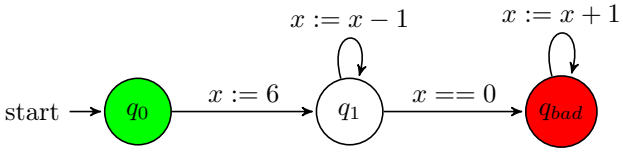

FIGURE 5 – supp_simple_final_loop

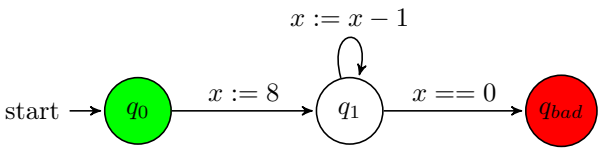

FIGURE 6 – supp_under_bound