Vérification logicielle et synthèse de programmes – TP2

Ce TP a pour but de programmer un vérificateur de programmes, qui utilisera le solveur SMT Z3 pour résoudre les obligations de preuves. Le premier exercice est la fondation du système de vérification, et est à faire avant de pouvoir poursuivre. Les exercices suivants sont des fonctionnalités supplémentaires. À vous de choisir lesquelles implémenter et dans quel ordre.

Pour chaque exercice, le dossier `examples` contient des exemples de programmes à vérifier.

1 Weakest pre-conditions et vérification

Le fichier `verifier.py` contient un analyseur syntaxique (programmé à l'aide de la bibliothèque Lark) et des classes pour modéliser un programme. Les expressions booléennes et arithmétiques sont directement représentées avec les structures de données de Z3.

Un programme est représenté par une instance de la classe abstraite `Instr`. Les différentes classes qui héritent de `Instr` correspondent aux différents types d'instructions (instruction vide, séquence d'instructions, conditionnelle, affectation).

Notez que pour l'instant, notre langage n'a pas de boucles et sa syntaxe n'autorise l'instruction `return` que comme dernière instruction d'une méthode (il n'y a donc pas de sauts dans le flot d'exécution). L'instruction `return` est par ailleurs traitée comme une affectation dans une variable fictive appelée "return", qui est utilisable dans les spécifications.

1. Pour chaque classe qui hérite de `Instr`, programmez la méthode `weakest_precondition` qui calcule la pré-condition la plus faible par rapport à une post-condition donnée en argument.
2. Codez la méthode `main` afin qu'elle fasse appel à Z3 pour vérifier que le programme analysé est correct par rapport à sa spécification.

2 Return

La grammaire fournie n'autorise l'instruction `return` que à la fin d'une méthode, afin d'éviter un saut dans le flot d'exécution. Pour cet exercice,

vous leverez cette limitation.

1. Ajoutez une nouvelle classe héritant de `Instr` pour modéliser l'instruction `return` (ou bien simplement une instruction de saut, auquel cas le `return` pourra être représenté par une affectation suivie d'un saut).
2. Comment modifier le calcul de la pré-condition la plus faible pour autoriser un saut à la fin de la méthode après un `return` ? Implémentez cette modification ?
3. Modifiez l'analyseur syntaxique afin que l'instruction `return` puisse être placée à n'importe quel endroit du programme.

3 Boucles

Pour cet exercice, vous ajouterez des boucles `while` au langage.

1. Ajoutez une classe héritant de `Instr` pour modéliser les boucles `while`. Cette classe devra posséder des attributs pour modéliser l'invariant (expression booléenne) et la mesure de terminaison (expression arithmétique) associée à cette boucle.
2. Implémentez la méthode `weakest_precondition` de cette classe. Cette méthode devra faire appel à Z3 pour vérifier que l'invariant est préservé par la boucle et garantit la post-condition. Dans le cas contraire, un message d'erreur sera affiché.
3. En plus des deux vérifications ci-dessus, ajoutez la vérification de la terminaison : on vérifiera que la mesure de terminaison décroît strictement à chaque itération, et qu'elle est bornée par une valeur minimale.

4 Quantificateurs

Pour cet exercice, vous ajouterez des quantificateurs au langage de spécification (pré-conditions, post-conditions, invariants de boucle).

1. Modifiez l'analyseur et le système de types pour autoriser les formules quantifiées dans les spécifications (on se limitera aux formules en forme prénexe, c.-à-d. où les quantificateurs sont en tête de formule).
2. Codez les méthodes pour que l'analyse syntaxique génère les expressions Z3 correspondantes.

5 Tableaux

Pour cet exercice, vous ajouterez des tableaux de booléens et d'entiers au langage.

1. Modifiez l'analyseur syntaxique et le système de typage pour permettre les programmes avec des tableaux d'entiers ou de booléens.
2. Utilisez la théorie des tableaux Z3 pour modéliser les expressions d'écriture et de lecture des tableaux.
3. Modifiez le calcul des pré-conditions afin de vérifier que les accès/écritures respectent les bornes des tableaux.

6 Appels de méthodes

Pour cet exercice, vous étendrez le langage pour permettre de déclarer plusieurs méthodes dans le même fichier, et permettre les appels de méthode parmi les instructions.

1. Modifiez l'analyseur syntaxique comme indiqué ci-dessus.
2. Implémentez la vérification des appels de fonctions. Cette vérification devra utiliser le principe de la "boîte noire", c.-à-d. utiliser uniquement la spécification de la méthode appelée (et pas son code).

7 Contre-exemples

Pour cet exercice, vous utiliserez les contre-exemples fournis par Z3 en cas d'échec de la vérification, afin d'améliorer les messages d'erreurs. Vous pourrez par exemple fournir des valeurs d'entrées telles que la méthode ne satisfait pas la spécification pour ces valeurs. Cette fonctionnalité peut être étendue à la vérification d'invariant et de terminaison, si vous avez implémenté les boucles.