Titre

Auteur :
Maxime CLEMENT

18 Janvier 2016

Table des matières

Introduction

La réparation automatique de bugs est un domaine en pleine expansion. En effet, le nombre de programme évolue de plus en plus chaques jours et par corrélation le nombre de bugs également. Il est envisageable dans les prochaines années que les développeurs ne soit pas assez nombreux pour résoudre tous ces bugs. C'est ici qu'intervient la réparation automatique de bugs, consistant à utiliser des programmes pour réparer d'autres programmes afin de remplacer ou guider ces développeurs.

Des approches déjà existantes[1][2] ont déjà fait leurs preuves. Elles permettent effectivement de réparer des bugs, mais leurs comportement n'est pas déterministe. Elles appliquent des transformations sur le code jusqu'à faire passer les tests. Est-il possible de trouver un comportement déterministe dans le but de réparer des bugs ?

L'objectif de ce rapport est de montrer qu'un tel comportement est possible.

La solution proposée ici est grandement inspirée par un outils appelé Nopol[3]. Cette approche utilise les entrées / sorties des tests pour définir des contraintes, et essaye de les résoudre à l'aide de solver SMT[1].

Evaluation ?????

[1] `https://fr.wikipedia.org/wiki/Satisfiability_modulo_theories`

Travail technique

1.1 But

Le but est d'utiliser les suites de tests d'un projet comme spécifications de celui-ci. Ces spécifications contiennent deux types de valeurs très importante. Premièrement les variables accessibles lors d'un comportement exécuté par l'application : les entrées. Ensuite la valeur attendu de ce comportement : la sortie. À la manière de Nopol[3], le but est de trouver une relation entre les entrées et la sortie à l'aide de solver SMT. Cette relation représentera le comportement nominal de l'application.

1.2 SMT

Un problème SMT est un problème dit de *satisfiabilité modulo théories*. DEFINIR

1.3 Overview

Premièrement, la phase de collecte de données est effectuée pour obtenir les entrées et la sortie pour chaque tests. Ensuite, cette approche va généré un problème SMT basé sur la même implémentation de Nopol[3]. Un solver sera alors appelé pour essayer de résoudre se problème. Si une solution existe, il est nécessaire de transformer la solution en code adéquate afin de générer un patch. Pour finir, les tests seront lancer à nouveau pour verifier la véracité du patch. La figure 1.1 représente le schéma d'exécution de la solution proposé.

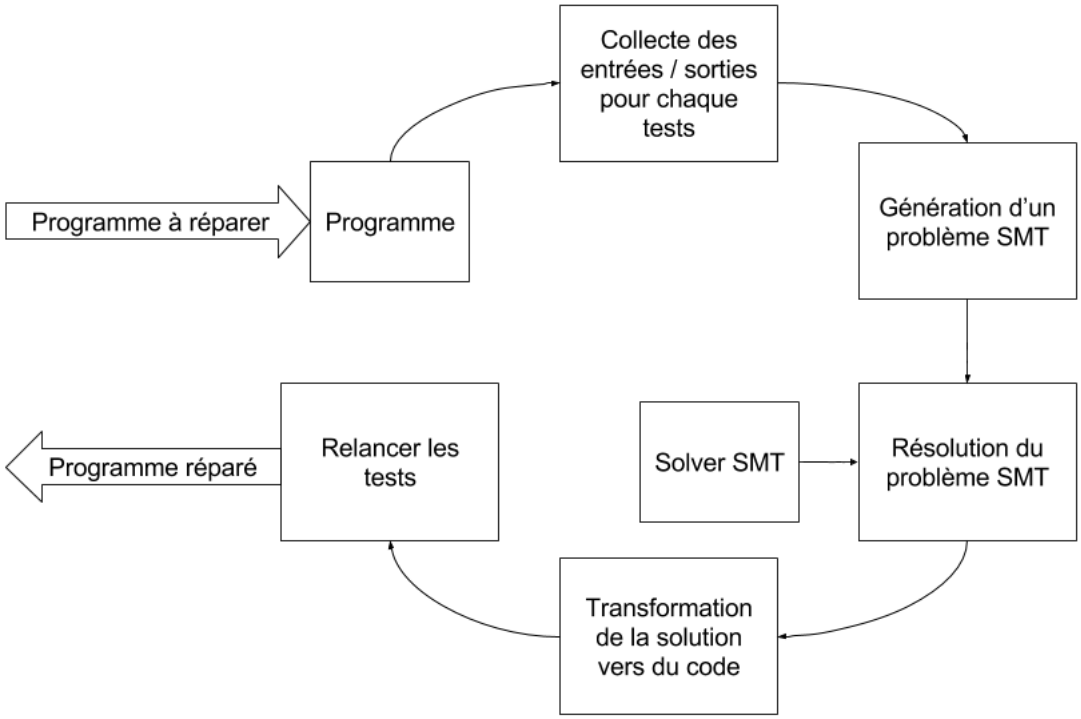

Figure 1.1: Workflow de l'exécution du programme

1.4 Scope

1.4.1 Programmes

Programme Java, contenant des tests JUnit défaillant.

1.4.2 Forme des tests

Appel direct de methode dans l'assert

1.4.3 Collecte

Paramètres des méthodes

1.4.4 Type des entrées / sorties

Si tous est entiers/réelles ou boolean, ca marche facilement sinon des problèmes interviennent

1.5 Implementation

L'implémentation est réalisé en Java. Pour analyser les tests et collecter les entrées / sorties, cette approche utilise Spoon[1], une librairie d'analyse et de transformation de code source.

La transformation des spécifications en problème SMT utilise les algorithmes déjà défini dans Nopol[3]. Ces algorithmes utilise JSMTLIB[2], une librairie permettant la génération de script SMT-LIB[3], un langage compréhensible par des solvers SMT.

Le solver utilisé est Z3[4], ce solver embarque les théories non-linéaires sur les réels, ce qui le démarque des autres.

[1] `http://spoon.gforge.inria.fr/`
[2] `https://github.com/smtlib/jSMTLIB`
[3] `http://smtlib.cs.uiowa.edu/`
[4] `https://z3.codeplex.com/`

Évaluation

??????????????????

Limitations

Le nombre de test est determinant car il représente la spécification du programme. Si pas assez, le patch peut être trivial et s'éloignera de la solution originale. Si trop de test, il est possible que le solver n'arrive pas a résoudre toutes ces contraintes.

L'identification de la méthode à réparer peut être difficile, si cette dernières n'est pas directement appelé dans l'assertion. De plus, le bug peut potentiellement provenir de plusieurs bugs dans différentes méthodes du codes.

Conclusion

CONCLUSION

OUVERTURE

Synthétiser des fragments de code, et non pas de méthodes entières.

L'intégration dans des outils tels de Nopol car le comportement est sensiblement identiques.

Améliorer la collecte des données pour récupérer plus que uniquement les paramètres de méthodes.

Améliorer la détection du code défaillant en recoupant les stacktraces des differents tests

Bibliography

[1] Forrest S Weimer W Le Goues C, ThanhVu Nguyen. Genprog: A generic method for automatic software repair. 2011.

[2] Abhik Roychoudhury Satish Chandra Hoang Duong Thien Nguyen, Dawei Qi. Semfix: Program repair via semantic analysis. 2013.

[3] Daniel Le Berre Favio DeMarco Martin Monperrus, Jifeng Xuan. Automatic repair of buggy if conditions and missing preconditions with smt. 2014.