OLS-Repair : One-Line Synthesis based on unit test and applied to Automatic Software Repair

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 appelé OLS-Repair, elle 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].

L'évaluation a été réalisé sur un sous-ensemble de introclassJava. Afin des respecter le scope définie en 1.3, des transformations synthaxique ont été effectuées. Cette évaluation met l'accent sur l'importance du nombre de tests ainsi que les constantes définies dans OLS-Repair.

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

Travail technique

1.1 Principe

Le principe 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. De ce fait, OLS-Repair ne dépend pas du type de bug car il considère la méthode défaillante comme une boite noir. À la manière de Nopol[3], il suffit trouver une relation entre les entrées et la sortie à l'aide de solver SMT. Cette relation représentera le comportement nominal de l'application et remplacera la boite noir. Les tests sont ensuite utilisé comme spécifications exécutables pour valider la solution synthétisée.

1.2 Overview

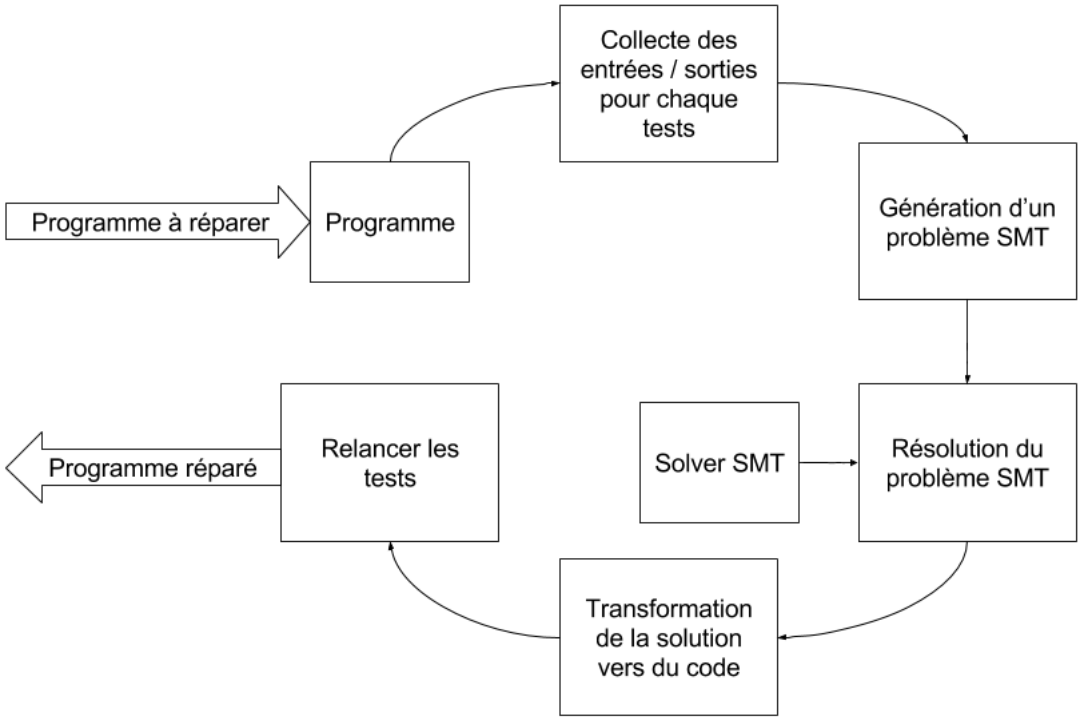

Figure 1.1: Workflow de l'exécution du programme

1.2.1 Collecte

La phase de collecte de données a pour objectif d'obtenir les entrées et la sortie pour chaque tests. Ces entrées / sorties sont vues comment des spécification, elles sont définies de la manière suivante : $S_i(I, O)$, où I représente l'ensemble des entrées pour le test i, et O la valeur attendu par le test i.

1.2.2 Génération

La phase de génération transforme l'ensemble des spécifications collectées en 1.2.1 en un problème SMT en utilisant les mêmes algorithmes que ceux définie dans Nopol[3]. Ces algorithmes ajoute également des constantes à I, valeurs souvent utilisées dans des expressions mathématiques, tel que -1, 0 et 1.

1.2.3 Résolution et synthèse

La phase de résolution est déléguée à un solver SMT. Ce dernier est appelé de manière itérative. Tous d'abord sur un problème avec un nombre d'opérateurs restreints. Ensuite, si le problème n'est pas décidables, alors de nouveaux opérateurs sont ajouté pour augmenter l'espace de recherche. Ces opérateurs sont définis dans la table 1.1

Itération	Opérateurs encodés
1	Aucun opérateur
2	== != < <=
3	! | | &&
4	+ -
5	?
6	*

Table 1.1: Opérateurs encodés en fonction du nombre d'itération effectuées avec le solver SMT

Lorsque le problème est dit *décidable*, une ligne de code est synthétisée par rapport à la solution proposé par le solver. Cette ligne remplacera ensuite le corps de la méthode contenant le bug.

1.2.4 Validation de la synthèse

Les spécifications mentionnées en 1.2.1 permettent de valider ou d'invalider la ligne synthétisée. Une fois insérée dans le code source, le code est compilé et les tests sont exécutés pour valider la synthèse.

1.3 Scope

1.3.1 Programmes

OLS-Repair s'applique uniquement sur des programmes Java. Ces derniers doivent contenir des tests écrits à l'aide du framework JUnit[1].

1.3.2 Collecte

La collecte des spécifications expliquées en 1.2.1 est statiques. OLS-Repair collecte uniquement les valeurs des paramètres de méthodes utilisées dans une assertion JUnit.

[1] `http://junit.org/`

1.3.3 Forme des tests

Afin de pouvoir collecter de manière statique les paramètres de méthodes, les tests doivent être écrit de la manières suivantes :

```
public void test() throws Exception {
    Calculatrice calc = new Calculatrice();
    int expected = 6;
    int a = 2;
    int b = 4;
    Assert.assertEquals(expected, calc.add(a, b));
}
```

Figure 1.2: Exemple de code permettant à OLS-Repair la collecte des spécifications

1.3.4 Type des entrées / sorties

OLS-Repair se limite à différents types pour I et O. La table 1.2 représente les types supportés. Afin de prendre en compte des tableaux, tous ces derniers doivent être de la même taille.

Type	I	O
int	✓	✓
int[]	✓	✘

Table 1.2: Opérateurs encodés en fonction du nombre d'itération effectuées avec le solver SMT

1.4 Implementation

OLS-Repair est implementé en Java. Pour analyser les tests et collecter les entrées / sorties, cette approche utilise Spoon[2], 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[3], une API permettant la génération de script SMT-LIB[4]. Ce langage est un standard compréhensible par différents solvers SMT.

Le solver utilisé est Z3[5], ce solver embarque les théories non-linéaires sur les réels, ce qui le démarque des autres. C'est grâce à Z3 que l'opérateur de multiplication mentionné en 1.2.3

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

peut être utilisé.

1.5 Utilisation

OLS-Repair est paramétrable, il se lance de la manières suivante :

```
Usage: OLS_Repair
       -s, --source-path path_buggy_program
       -j, --junit-path path_junit_jar
       -z, --z3-path path_z3_executable
      [-c, --constant one_constant_to_add]*
      [-o, --override]
      [-u, --use-blackbox]
```

Évaluation

2.1 IntroclassJava

Le dataset IntroclassJava[1] est un ensemble de programmes java buggés d'étudiants de L1 de l'Université de Californie. Ce dataset est une transformation de Introclass[2], codé en C. Chaque programme vient avec un moins un test qui ne passe pas. L'évaluation de OLS-Repair a été effectué uniquement sur les projets *median* et *smallest*.

2.1.1 Transformation

Dans le but de respecter le scope définie en 1.3, des transformations synthaxiques ont été réalisé sur le dataset. Ces transformations ne changent en rien les spécifications du programme, elles permettent cependant à OLS-Repair d'effectuer la phase de collecte correctement et de rendre le code plus réaliste. Les figures 2.1.1 et 2.1.1 montre la transformation apportée. Les projets *median* et *smallest* transformés sont disponible sur GitHub de OLS-Repair[3].

```
public void test() throws Exception {
    median mainClass = new median();
    String expected =
        "Please enter 3 numbers separated by spaces > " +
        "6 is the median";
    mainClass.scanner = new java.util.Scanner("2 6 8");
    mainClass.exec();
    String out = mainClass.output.replace("\n", " ").trim ();
    assertEquals(expected.replace(" ", ""), out.replace(" ", ""));
}
```

Figure 2.1: Exemple de test avant transformation

[1] https://github.com/Spirals-Team/IntroClassJava
[2] http://dijkstra.cs.virginia.edu/genprog/resources/autorepairbenchmarks//IntroClass/
[3] https://github.com/maxcleme/OLS-Repair

```
public void test() throws Exception {
    median mainClass = new median();
    int expected = 6 ;
    int[] input = {2,8,6};
    Assert.assertEquals(expected, mainClass.median(input));
}
```

Figure 2.2: Exemple de tess après transformation

2.1.2 Redondance

Comme expliqué en 1.1, les spécifications sont indépendantes du code présent dans la méthode contenant un bug. Si OLS-Repair peut réparer une version du dataset, alors toutes les autres versions ayant les mêmes spécifications seront réparables également.

2.2 Résultats

Les constantes ajoutées à I lors de la génération du problème SMT ont un rôle important pour le solver. La table 2.1 montre que les lignes synthétisées diffèrent en fonction de ces constantes. Pour réaliser cette évaluation, uniquement les spécifications de la classe dit *Whitebox* ont été prise en compte.

Constante	Median		Smallest	
	Whitebox	Blackbox	Whitebox	Blackbox
∅	6/6	3/7	8/8	3/8
[0]	6/6	3/7	8/8	0/8
[-1 ; 1]	6/6	4/7	8/8	3/8
[-1 ; 0 ; 1]	6/6	4/7	8/8	0/8
[2 ; 4 ; 8 ; 16 ; 32 ; 64]	6/6	3/7	8/8	3/8

Table 2.1: Impactes des constantes sur la pertinence de la solution

Peut importe les constantes, les spécifications de *Whitebox* sont toujours respectées car elles sont définie dans le problème SMT. Par contre, les spécifications de *Blackbox* ne sont pas toujours respectées pour cause que la ligne synthétisé n'est pas assez générique. En effet, la présence de constante tel que 0 peu mener le solver vers une solution triviale, comme pour *Smallest* où toutes les sorties O de *Whitebox* sont égales à 0.

En généralisant sur l'ensemble du dataset IntroclassJava original, OLS-Repair peut réparer les 48 versions défaillantes de *median* ainsi que les 45 versions défaillantes de *smallest* car les tests sont identiques pour chaque versions. Seulement, uniquement les spécifications *Whitebox* sont respectées. Comme expliqué en 2.1, il est possible que la ligne synthétisée respecte certaine spécification *Blackbox*, cependant elles sont ne sont jamais respectées dans l'intégralité. Cependant, si l'on regarde uniquement les spécifications prise en compte, OLS-Repair répare 100% des projets considérés.

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.