OLS-Repair : One-Line Synthesis based on Unit tests 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 programmes 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 fait leurs preuves. Elles permettent effectivement de réparer des bugs, mais leurs comportements dépendent du code fournis avec le programme défaillant. Elles appliquent des transformations sur le code jusqu'à faire passer les tests. Cependant l'utilisation de codes contenants des bugs est-il une bonne pratique pour réparer automatiquement un programme ? Une question peut alors se poser : Est-il possible de réparer des programmes tout en étant indépendant des bugs présents ?

L'objectif est de montrer qu'un tel comportement est possible. La solution proposée ici est appelée 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 réalisée sur un sous-ensemble de IntroclassJava, un dataset contenant des bugs réels. Afin des respecter le scope définie en 1.3, des transformations synthaxiques 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 importantes. Premièrement les variables accessibles lors d'un comportement exécuté par l'application : les entrées. Ensuite la valeur attendue 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 boîte noire. À la manière de Nopol[3], il de suffit trouver une relation entre les entrées et les sorties à l'aide de solver SMT. Cette relation représentera le comportement nominal de l'application et remplacera la boîte noire. Les tests sont ensuite utilisés comme spécifications exécutables pour valider la solution synthétisée.

1.2 Overview

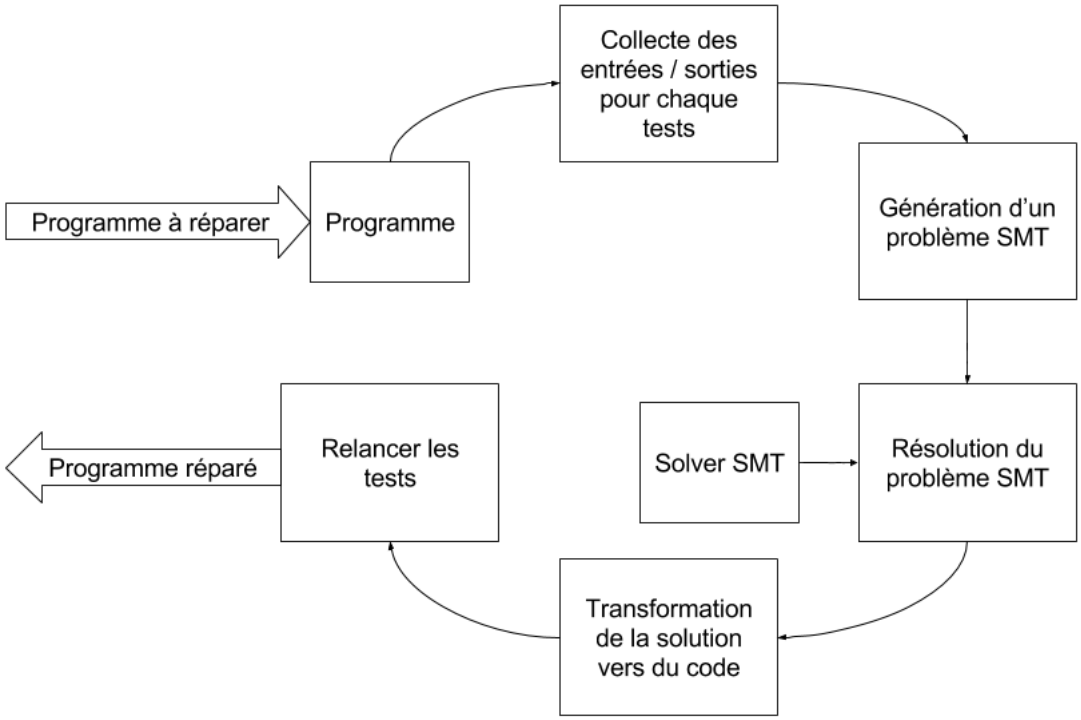

Figure 1.1: Workflow de l'exécution du programme

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 sortie pour 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 définis dans Nopol[3]. Ces algorithmes ajoutent 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 *décidable*, alors de nouveaux opérateurs sont ajoutés 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'itérations effectuées avec le solver SMT

Lorsque le problème est dit *décidable*, une ligne de code est synthétisée en fonction de la solution proposée 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] `http://junit.org/`

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.

1.3.3 Forme des tests

Afin de pouvoir collecter de manière statique les paramètres de méthodes, les tests doivent être écrits de la manière présentée dans la figure 1.2.

```
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: Types supportés par OLS-Repair

1.4 Implementation

OLS-Repair est implementé en Java. Cette approche utilise Spoon[2], une librairie d'analyses et de transformations de codes sources pour instrumenter les tests, collecter les entrées / sorties et insérer un patch.

[2] `http://spoon.gforge.inria.fr/`

La transformation des spécifications en problème SMT utilise les algorithmes déjà définis dans Nopol[3]. Ces algorithmes utilisent 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 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]
```

[3] https://github.com/smtlib/jSMTLIB
[4] http://smtlib.cs.uiowa.edu/
[5] https://z3.codeplex.com/

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 au moins un test qui ne passe pas. L'évaluation de OLS-Repair a été effectuée uniquement sur les projets *median* et *smallest*.

2.1.1 Transformation

Dans le but de respecter le scope défini en 1.3, des transformations synthaxiques ont été réalisées 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 **??** et **??** montrent la transformation apportée. Les projets *median* et *smallest* transformés sont disponibles sur le 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 test 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 du test dit *Whitebox* ont été prises 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: Impacts des constantes sur la pertinence de la solution

Peu importe les constantes, les spécifications de *Whitebox* sont toujours respectées car elles sont définies dans le problème SMT. Par contre, les spécifications de *Blackbox* ne sont pas toujours respectées cat la ligne synthétisée n'est pas assez générique. La présence certaines constantes tel que 0 peut mener le solver vers une solution triviale, comme pour *Smallest* où toutes les sorties O de *Whitebox* sont égales à 0. La figure 2.3 représente une ligne synthétisée par OLS-Repair.

```
public int median(int[] param) {
    return (param[2] < param[0] + param[1])?(param[2]):(param[1]);
}
```

Figure 2.3: Ligne synthétisée par rapport aux tests *Whitebox* de *median* avec comme constantes [-1 ; 0 ; 1]

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 certaines spécifications *Blackbox*, cependant elles sont ne sont jamais respectées dans l'intégralité. Toutefois, si l'on regarde uniquement les spécifications prisent en compte, OLS-Repair répare 100% des projets considérés.

Limitations

Le nombre de tests est determinant car il représente une spécification pour le programme à réparer. Si ils sont peu nombreux, le patch peut être trivial et s'éloignera du comportement nominal. À l'inverse, si ils sont trop nombreux, il est envisageable que le solver n'arrive pas a résoudre toutes les contraintes.

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

Conclusion

Grâce à son utilisation de problèmes SMT, des tests comme spécifications et sa vision de la méthode défaillante comme une boîte noire, OLS-Repair permet de répondre brillamment à la question initiale : Est-il possible de réparer des programmes tout en étant indépendant des bugs présents ? Cette approche pourrait également être utilisée lors d'un développement en TDD[1] dans le but d'écrire uniquement les tests et de synthétiser le comportement voulu plutôt que de l'écrire.

OLS-Repair est encore jeune et il existe de nombreux axes d'améliorations : la collecte des données pourrait être dynamique afin de récuperer plus d'informations comme par exemple, les valeurs des attributs de classes lors de l'exécution d'un test. La détection de la méthode défaillante pourrait se baser sur les stacktraces des différents tests. Il est aussi possible d'augmenter le nombre de types de données supportés dans les spécifications. Une idée originale serait également d'étendre le principe des méthodes comme boîte noire à des fragments de codes. Pour finir, on pourrait imaginer l'intégration de cette approche dans des outils tels que Nopol[3] puisque le comportement est sensiblement identique.

[1] https://fr.wikipedia.org/wiki/Test_driven_development

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.