Automated Transplantation and Differential Testing for Clones

Tianyi Zhang, Miryung Kim

ICSE '17 Proceedings of the
39th International Conference on Software Engineering
Pages 665-676

発表者:野口耕二朗

目次

- 背景, 目的
- GRAFTERの構築
- 評価実験
- まとめ, 所感

背景, 目的

背景

- クローンペアの一方はテストされるが, もう一方はテストされないことがよくある
- もう一方のテストがされないことでそのクローンにバグが混入しても見つけることが出来ない

目的

- クローンの実行時の振る舞いの差異を考慮しつつ，クローンに対するテストの再利用を可能にするツールGRAFTERを構築する

- GRAFTERで生成されたもう一方のクローンのテストがバグを検出するかどうかを確かめる

GRAFTERの構築

事例

```
1  public class Copy extends Task{
2   private IncludePatternSet includes;
3
4   public void setIncludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7  -    StringTokenizer tok=new StringTokenizer(patterns,",");
8  -    while(tok.hasMoreTokens()){
9  -      includes.addPattern(tok.next());
10 -    }
11 +    String[] tokens = StringUtils.split(patterns, ",");
12 +    for(String tok : tokens){
13 +      includes.addPattern(tok);
14 +    }
15    }
16  }
17   ...
18 }
19
20 public class IncludePatternSet{
21  public Set<String> set;
22  public void addPattern(String s) { set.add(s); }
23   ...
24 }
```

(a) Correctly edited clone in the Copy class

```
1  public class Delete extends Task{
2   private ExcludePatternSet excludes;
3
4   public void setExcludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7  -    StringTokenizer tok=new StringTokenizer(patterns,",");
8  -    while(tok.hasMoreTokens()){
9  -      excludes.addPattern(tok.next());
10 -    }
11 +    String[] tokens = StringUtils.split(patterns, ".");
12 +    for(String tok : tokens){
13 +      excludes.addPattern(tok);
14 +    }
15    }
16  }
17   ...
18 }
19
20 public class ExcludePatternSet{
21  public Set<String> set;
22  public void addPattern(String s) { set.add(s); }
23   ...
24 }
```

(b) Inconsistenly edited clone in the Delete class

Fig. 1: Similar edits to update the use of `StringTokenizer` API to `StringUtils.split` in `Copy` and `Delete`.

```
1 @Test
2 public void testCopy(){
3   Task copyTask = FileUtils.createTask(FileUtils.COPY);
4   ...
5   copyTask.setIncludes("src/*.java, test/*.java");
6   JobHandler.fireEvent(copyTask);
7   assertTrue(checkFileCopied());
8 }
```

Fig. 2: A test case for the `Copy` class.

事例

- クラスDeleteのファイルセパレータ部分でバグがあるがテスト忘れで見つけることができていない
- 右図のようにクラスCopyのコードを書き換えることでCopyのテストを再利用することが出来る

```
1  public class Copy extends Task{
2    private IncludePatternSet includes;
3  +  private ExcludePatternSet excludes;
4
5    public void setIncludes(String patterns){
6      ...
7      /* this is stub code inserted for data transfer*/
8  +    ExcludePatternSet excludes_save = excludes;
9  +    excludes = new ExcludePatternSet();
10 +    excludes.set = includes.set;
11
12     /* the original code is replaced with the grafted code
           from setExcludes*/
13 -    if(patterns != null && patterns.length() > 0){
14 -      String[] tokens = StringUtils.split(patterns, ",");
15 -      for(String tok : tokens){
16 -        includes.addPattern(tok);
17 -      }
18 -    }                                   Original Clone (Deleted)
19 +    if(patterns != null && patterns.length() > 0){
20 +      String[] tokens = StringUtils.split(patterns, ".");
21 +      for(String tok : tokens){
22 +        excludes.addPattern(tok);
23 +      }
24 +    }                                   Grafted Clone (Inserted)
25
26     /* this is stub code inserted for data transfer*/
27 +    includes.set = excludes.set;
28 +    excludes = excludes_save;
29   }
30 }
```

Fig. 3: GRAFTER grafts the clone in `Delete` (lines 19-24) in place of the original clone in `Copy` (lines 13-18) for test reuse. GRAFTER inserts stub code (highlighted in yellow).

大まかな流れ

1.クローンの変数などの特定

2.コードの移植や未定義の変数などの挿入

3.入力や中間のデータを移植する

4.テストの結果と中間の状態を出力

クローンの変数などの特定

- 以降の手順に必要になる2つの変数を定義
  - consumed variable
    - テストがない方のクローンで使われており, テストがある方のクローンで定義する必要がある変数
  - affected variable
    - 値が移される側で更新される可能性がある変数

- consumed variableはクローン内部のデータの流れに関わってくる

- affected variableはクローンの外部で更新されるデータの流れに関わってくる

クローンの変数などの特定

- Def(F)：コード片Fで定義された変数
- Use(F)：コード片Fで使用された変数
- In−Scope(P)：プログラムの位置Pで更新可能な変数
- consumed(F,M,C)=(Def(C)∪Def(M)\Def(f))∩Use(F)
- Affected(F,P)=Use(F)∩In-Scope(P)

コードの移植や未定義の変数などの挿入

- 型の安全性を保証するための5つのルールを適用
  - 移植する変数名はテストがない方のものを使う。かぶっている場合は先頭に_をつける
  - 定義されていないメソッドはテストがない方から持ってくる。同じメソッド名のメソッドがある場合は変数名の場合と同じ方法をとる
  - クローンの間で型が異なる場合，型を合わせるためのスタブを生成する
  - return文が返す値が方に合わない可能性があるので，返す値をreturnする前に合うようにする
  - 再帰を使っている場合は返り値をテストが無い方のものに合わせるようにする

入力や中間のデータを伝達する

- クローンの開始地点で, テストが無い方の変数の値をテストがある方の変数に伝達するためにスタブを生成する

- クローンの終了地点で, テストが無い方の変数をテストがある方の変数の値に更新する

テストの結果比較と中間状態の比較

- テストレベルの比較
  - クローンペアに対して同じテストを実行してそれぞれの結果を比較
- 状態レベルの比較
  - クローンの終了地点のaffected variableをそれぞれ比較

評価実験

データセットとRQ

- データセット

Subject	LOC	Test#	Branch	Stmt	Pair#
ant-1.9.6	267,048	1,864	45%	50%	18
Java-APNS-1.0.0	8,362	103	59%	67%	7
xmlsec-2.0.5	121,594	396	59%	65%	27

TABLE I: Subject Programs

- リサーチクエスチョン
  - RQ1:GRAFTERはコードの移植にどれだけ成功するか
  - RQ2: クローンの振る舞いの違いを見つける能力についてGRAFTERはクローンバグを静的解析するツールにどれだけ匹敵するか
  - RQ3:プログラムの間違いによる予期せぬ振る舞いの違いを見つける能力にどれだけ長けているか

RQ1の実験結果

- 52組のクローンペアの内49組のクローンペアでコードを移植することに成功した（94%）

- 3組のクローンが失敗した原因はクローンの変数の型を合わせることが出きなかったため

RQ2の実験結果

- クローンの振る舞いの違いを検出できる能力をGRAFTERとクローンのバグを静的解析するツール[2]と性能を比較

- GRAFTERはテストレベルで20組, 状態レベルで41組検出

- 静的解析ツールで16組検出

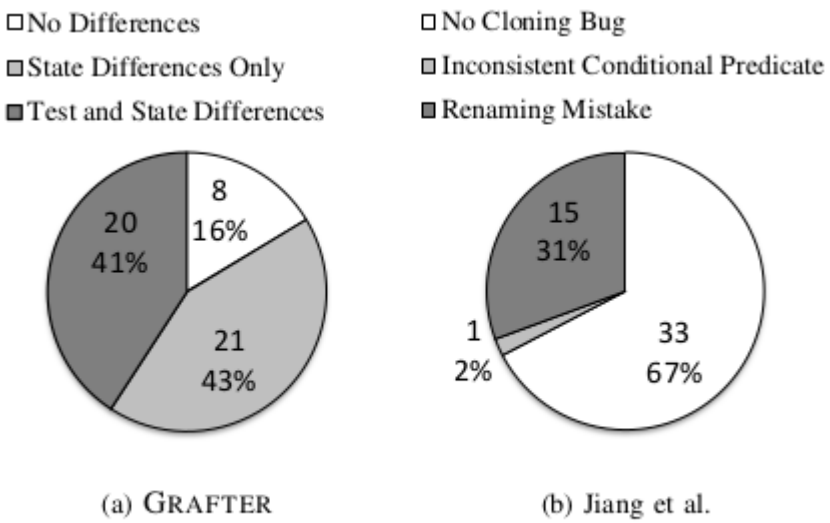

Fig. 6: Comparison between GRAFTER and Jiang et al.

RQ3の実験結果

- ミューテーション解析を用いる. 30のクローンペアに361ミュータントを注入
  - ミュータント解析;テスト対象のコードにわざとバグを入れ, それをテストがどれだけ検出できるかを測る手法
- GRAFTERはテストレベルで46%, 状態レベルで64%のミュータントを検出

RQ3の実験結果

- 静的解析ツールで35%検出

- 検出できるミュータントの種類でもGRAFTERが優位

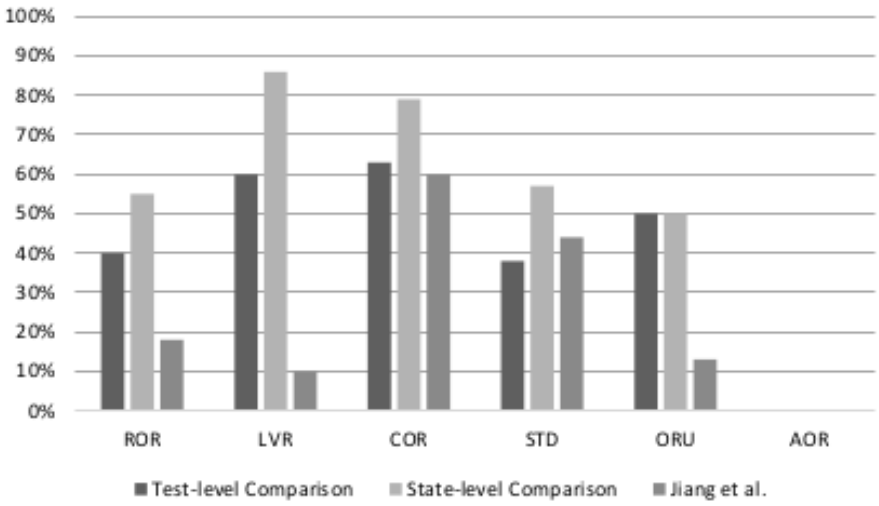

Fig. 9: Mutant killing ratio for different mutant kinds

まとめ, 所感

まとめ

- GRAFTERによってクローンのテストを高確率で再利用することが出来た

- GRAFTERによってテストを再利用することにより，テストされていない方のクローンでバグを検出することを静的解析ツールより出来た

所感

- 実行時のクローンの振る舞いの違いが実際の開発ではかなり厄介であることは勉強になった
- 大規模開発でこのツールを使いこなすことができればかなり負担を減らせると思った