Identifying Test-Suite-Overfitted Patches through Test Case Generation

Qi Xin, Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI, USA
{qx5,spr}@cs.brown.edu

ABSTRACT

A typical automatic program repair technique that uses a test suite as the correctness criterion can produce a patched program that is test-suite-overfitted, or overfitting, which passes the test suite but does not actually repair the bug. In this paper, we propose DiffTGen which identifies a patched program to be overfitting by first generating new test inputs that uncover semantic differences between the original faulty program and the patched program, then testing the patched program based on the semantic differences, and finally generating test cases. Such a test case could be added to the original test suite to make it stronger and could prevent the repair technique from generating a similar overfitting patch again. We evaluated DiffTGen on 89 patches generated by four automatic repair techniques for Java with 79 of them being likely to be overfitting and incorrect. DiffTGen identifies in total 39 (49.4%) overfitting patches and yields the corresponding test cases. With the fixed version of a faulty program being the oracle, the average running time is about 7 minutes. We further show that an automatic repair technique, if configured with DiffTGen, could avoid yielding overfitting patches and potentially produce correct ones.

Keywords

patch overfitting; test case generation; automatic program repair

1. INTRODUCTION

Given a faulty program and a fault-exposing test suite, an automatic program repair technique aims to produce a correct, patched program that passes the test suite. Being automatic, such a technique could potentially save people significant time and effort. Over the past decade, a variety of automatic repair techniques [5, 9, 14–16, 21–23, 37–39] have been developed. Current repair techniques, however, are still far from maturity: they often yield an overfitting, patched program which passes the test suite but does not actually repair the bug. Studies [30, 33] have shown that early repair techniques suffer from severe overfitting problems. According to [30], the majority of patches generated by GenProg [5], AE [37] and RSRepair [29] are incorrect. More recent techniques look at many other methods (e.g., using human-written patches [10], repair templates and condition synthesis [15], bug-fixing instances [14, 16] and forbidden modifications [34]) for repair. However, their repair performances are still relatively poor. Within a 12-hour time limit, the state-of-the-art repair techniques SPR [15] and Prophet [16] generated plausible patches that pass the test suite for less than 60% bugs in a dataset containing 69 bugs, with more than 60% of the plausible patches (the first found ones) being incorrect.

The low quality of a test suite is a critical reason why an overfitting patch might be generated. Unlike a formal specification, the specification encoded in a test suite is typically weak and incomplete. For example, the fault-exposing test case in the test suite associated with the bug *Math_85* from the Defects4J dataset [8] simply checks whether a method returns a result without any exception thrown, but does not check the correctness of the result. A patch generated by jGenProg (the Java version of GenProg [5]) simply removes the erroneous statement that triggers the exception without actually repairing it. The patch avoids the unexpected exception but deletes the expected functionality of the original program and thus introduces new bugs. It is not surprising that a test suite sometimes contains such weak test cases since the test suite is designed for humans but not for machines, and a human seldomly makes an unreasonable patch by deleting the desirable functionality of a program. However, such a weak test suite harms the performance of an automatic repair technique, e.g., jGenProg. When a patched program that passes the test suite is generated, jGenProg would simply accept it as there is no extra knowledge other than the given test suite to validate its correctness.

In this paper, we propose DiffTGen, a patch testing technique to be used in the context of automatic program repair. DiffTGen identifies overfitting patches generated by an automatic repair technique through test case generation. Based on the syntactic differences between a faulty program and a patched program, DiffTGen employs an external test generator to generate test methods (test inputs) that could exercise at least one of the syntactic differences upon execution. To actually find any semantic difference, DiffTGen instruments the two programs, runs the programs against the generated test method, and compares the running outputs. If the outputs are different, DiffTGen reports the difference to the oracle for correctness judging. If the output of the patched program is incorrect, we know the patch is overfitting. If a correct output could be provided by the oracle, DiffTGen would produce an overfitting-indicative test case by augmenting the test method with assertion statements. (Note that it is not interesting when the running outputs are identical, since they are not related to any changes the patch makes.)

DiffTGen can be combined with an automatic repair technique to enhance its performance. After a patch is generated by the repair technique, DiffTGen may produce a test case showing the patch

DOI: 10.1145/1235

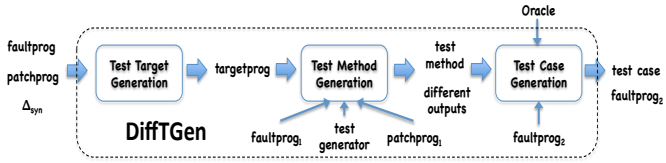

Figure 1: The Overview of DiffTGen. $faultprog$: the faulty program; $patchprog$: the patched program; Δ_{syn}: the syntactic differences between $faultprog$ and $patchprog$; $targetprog$: the test target program; $faultprog_1$, $patchprog_1$: the output-instrumented versions of $faultprog$ and $patchprog$; $faultprog_2$: the test-case-instrumented version of $faultprog$.

```
1  public static boolean toBoolean(String str) {
2   if (str=="true") return true;
3   if (str==null) return false;
4   switch (str.length()) {
5     case 2: { ... }
6     case 3: {
7      char ch = str.charAt(0);
8      if (ch=='y')
9        return
10       (str.charAt(1)=='e'||str.charAt(1)=='E')
11       && (str.charAt(2)=='s'||str.charAt(2)=='S');
12
13     if (ch=='Y')//Changed to "if (str!=null)" (Overfitting Patch)
14       return
15       (str.charAt(1)=='E'||str.charAt(1)=='e')
16       && (str.charAt(2)=='S'||str.charAt(2)=='s');
17    //Inserted "return false;" (Correct Patch)
18    }
19    case 4: {
20      char ch = str.charAt(0);
21      if (ch=='t') {
22        return
23        (str.charAt(1)=='r'||str.charAt(1)=='R')
24        && (str.charAt(2)=='u'||str.charAt(2)=='U')
25        && (str.charAt(3)=='e'||str.charAt(3)=='E');
26      }
27      if (ch=='T') { ... }
28    }
29  }
30  return false;
31 }
```

Figure 2: The Lang_51 Bug & an Overfitting Patch

is overfitting. Such a test case could be added to the original test suite to make the test suite stronger. Using the augmented test suite, the repair technique avoids yielding a category of patches that have similar overfitting properties, and could potentially produce a correct patch (See Section 4.2).

The main contributions we make in this paper are as follows:

- We built a patch testing tool DiffTGen which could identify an overfitting patch generated by an automatic repair technique through test case generation. The tool is currently available at github.com/qixin5/DiffTGen.
- We empirically evaluated DiffTGen on a set of 89 patches generated by four automatic repair techniques for Java: jGenProg [20], jKali [20], Nopol [39], and HDRepair [14] with 79 patches being likely to be overfitting and incorrect. DiffTGen identified 39 (49.4%) patches to be overfitting with the corresponding test cases generated. With a bug-fixed program as the oracle, the average time is only about 7 minutes.
- We empirically evaluated the effectiveness of DiffTGen in the context of automatic repair program. Our results show that an automatic repair technique, if configured with DiffTGen, could avoid generating overfitting patches and generate correct patches eventually.

2. OVERVIEW

In this section, we go over how DiffTGen works with an example. DiffTGen accepts as input a faulty program $faultprog$, a patched program $patchprog$, a set of syntactic differences Δ_{syn} between the two programs, and an oracle. A syntactic difference $\delta_{syn} \in \Delta_{syn}$ is a tuple <$faultstmt$, $patchstmt$> where $faultstmt$ and $patchstmt$ are the respective statements in $faultprog$ and $patchprog$ that are related to the change. Note that a δ_{syn} could have a *null* value for either $faultstmt$ or $patchstmt$ (but not both) to represent an insertion or a deletion. If neither $faultstmt$ nor $patchstmt$ is *null*, δ_{syn} is a replacement. In the context of automatic program repair, a repair technique often produces a patch report containing what changes it has made, and Δ_{syn} could be obtained by a simple report analysis. As output, DiffTGen either produces a test case showing $patchprog$ is overfitting or produces nothing if no such test cases can be found. (For a generated test case, DiffTGen also produces a test-case-instrumented version of $faultprog$. For testing, one needs to run this version against the test case. In the instrumented version, the original semantics of $faultprog$ is preserved, see Section 3.4.2.1.) Intuitively, an overfitting, patched program passes the original test suite but does not actually repair the bug. (A formal definition of an overfitting patch can be found at Section 3.1.) DiffTGen goes through three stages to produce a test case: *Test Target Generation*, *Test Method Generation* and *Test Case Generation*. In the first stage, DiffTGen produces a target program $targetprog$ on which a test generator works to generate test inputs. In the second stage, DiffTGen employs a test generator to actually generate test methods (as test inputs) that uncover semantic differences between $faultprog$ and $patchprog$. In the third stage, DiffTGen produces test cases, if any, showing $patchprog$ is overfitting based on the semantic differences. Figure 1 shows an overview of DiffTGen.

We use the example shown in Figure 2 to explain the three stages. The faulty program (in Java) is a real bug (*Lang_51*) in the Defects4J bug dataset [8]. The functionality of the program is to convert a string into a boolean value. The fault-exposing test case from the test suite associated with the bug invokes the method `toBoolean` with the string "tru" as the value for `str`. Upon execution with `str="tru"`, the method `toBoolean` is expected to return *false* as the output. However, without the correct return statement inserted at Line 17, the branch of `case 4` is executed where an *IndexOutOfBoundsException* is thrown at Line 25. A patched program that modifies the if-condition at Line 13 (from `ch=='Y'` to `str!=null`) is generated by an automatic repair technique NoPol [39] in the repair experiments conducted by Martinez et al. [18]. The patched program now works fine for the input "tru" (it returns *false* after executing the return statement starting at Line 14) and passes the original test suite, but it does not actually repair the bug. For this example, DiffTGen generates a new test case with the input string "@es" which exposes a new failure: the expected output is *false* but the patched program returns *true*. With a fixed version of the program being the oracle, it only took DiffTGen about 3.8 minutes to generate the test case.

2.1 Test Target Generation

In the first stage, DiffTGen generates a program which we call the test target program, or $targetprog$, based on $faultprog$, $patchprog$, and the syntactic differences Δ_{syn} between them. $targetprog$ is the actual program on which a test generator later works to generate test inputs. It is an extended version of $patchprog$ with dummy statements inserted as the coverage goals. A test input that is generated by a test generator with at least one of the coverage goals satisfied can lead to a differential execution between $faultprog$ and $patchprog$. Such an input is likely to uncover a semantic difference δ_{sem} between the two programs and further expose an overfitting behavior of $patchprog$.

To obtain $patchprog$, for each $\delta_{syn} \in \Delta_{syn}$, DiffTGen inserts a dummy statement in $patchprog$. For simple cases, where a patching modification does not involve changing an if-condition, DiffTGen simply inserts a dummy statement before the modified state-

```
1 @Test public void test078() throws Throwable  {
2   boolean boolean0 = BooleanUtils.toBoolean("@es");
3 }
```

Figure 3: A Test Method Generated by EvoSuite

```
1 public static boolean toBoolean(String str) {
2   Object o_7au3e = null;
3   String c_7au3e =
4   "org.apache.commons.lang.BooleanUtils";
5   String msig_7au3e =
6   "toBoolean(String)" + eid_toBoolean_String_7au3e;
7   try {
8     o_7au3e = toBoolean_7au3e(str);
9     FieldPrinter.print(o_7au3e, eid_toBoolean_String_7au3e,
10                       c_7au3e, msig_7au3e, 0, 5);
11  } catch (Throwable t7au3e) {
12    FieldPrinter.print(t7au3e, eid_toBoolean_String_7au3e,
13                       c_7au3e, msig_7au3e, 0, 5);
14    throw t7au3e;
15  } finally {
16    eid_toBoolean_String_7au3e++;
17  }
18  return (boolean) o_7au3e;
19 }
```

Figure 4: The Output-Instrumented Version of $faultprog$.

ment (for insertion or replacement) or in place of the removed statement (for deletion). For more complicated cases, where a patching modification is related to an if-statement and effectively modifies an if-condition (which is a common situation [19, 26]), DiffTGen produces a synthesized if-statement containing a dummy statement and inserts it in $patchprog$. The advantage of such a synthesized if-statement is as follows: a test input that covers the dummy statement (the coverage goal) would expose different branch-taking behaviors related to a modified if-statement between $faultprog$ and $patchprog$. Such a test input is thus likely to uncover a δ_{sem} between the two programs.

For the example in Figure 2, DiffTGen creates a $targetprog$ by inserting a newly synthesized if-statement before Line 13. The synthesized if-statement is as shown below.

```
if (((ch=='Y')&&!(str!=null))||(!(ch=='Y')&&(str!=null))) {
int delta_syn_3nz5e_0 = -1; } //A dummy statement
```

In the context of the program, the if-condition is equivalent to *if (ch!='Y')*. In this example, for any input *str* that covers the dummy statement in $targetprog$, it would lead to a differential execution between $faultprog$ and $patchprog$: the input would not exercise the return statement in $faultprog$ (starting at Line 14) but would exercise the one in $patchprog$.

2.2 Test Method Generation

In the second stage, DiffTGen employs an external test generator (we use EvoSuite [4]) to generate test methods (test inputs) for $targetprog$ that can cover at least one of the dummy statements upon execution. (Note that a test method contains no assertion statements, but there is at least one assertion statement in a test case.) For our example, one of the generated test methods is shown in Figure 3.

A generated test method can exercise a δ_{syn} between $faultprog$ and $patchprog$ upon execution, but may or may not be able to uncover a δ_{sem}. To tell whether a test method can uncover a δ_{sem}, DiffTGen creates instrumented versions of $faultprog$ and $patchprog$ (called the *output-instrumented* versions), runs them against the test method to obtain running outputs, and compares the outputs. In an output-instrumented version of a program (either $faultprog$ or $patchprog$), DiffTGen creates statements that print as outputs values that can be affected by δ_{syn}.

For our example, DiffTGen creates an output-instrumented version of `toBoolean` shown in Figure 4[1] (which can be used for either $faultprog$ or $patchprog$). Essentially, the code calls the original version of the method at Line 8 (the one shown in Figure 2, now renamed as `toBoolean_7au3e`) and prints the returned value `o_7au3e` at Lines 9-10. Along with the return value, the code also prints other values (e.g., one of them is the full class name of the method `c_7au3e`) which DiffTGen later uses to retrieve the output value for producing an assertion statement for a test case. If any exceptions are thrown, DiffTGen would also print the exceptional information (Lines 12-13). More details can be found in Section 3.3.1.

[1]The code needs a JDK version higher than 1.5 to compile.

```
1 public static boolean toBoolean(String str) {
2   Object o_7au3e = null;
3   String c_7au3e =
4   "org.apache.commons.lang.BooleanUtils";
5   String msig_7au3e =
6   "toBoolean(String)" + eid_toBoolean_String_7au3e;
7   try {
8     o_7au3e = toBoolean_7au3e(str);
9     addToORefMap(msig_7au3e, o_7au3e);
10    addToORefMap(msig_7au3e, null);
11    addToORefMap(msig_7au3e, null);
12  } catch (Throwable t7au3e) {
13    addToORefMap(msig_7au3e, t7au3e);
14    throw t7au3e;
15  } finally {
16    eid_toBoolean_String_7au3e++;
17  }
18  return (boolean) o_7au3e;
19 }
```

Figure 5: The Test-Case-Instrumented Version of $faultprog$.

```
1 //The output of the faulty program (instrumented)
2 Test Method: test078
3 PRIM_LOC:(E)0,(C)org.apache.commons.lang.BooleanUtils,(MSIG)
      ↪ toBoolean(String)0,(I)0
4 TYPE:Boolean
5 VALUE:false
6
7 //The output of the patched program (instrumented)
8 Test Method: test078
9 PRIM_LOC:(E)0,(C)org.apache.commons.lang.BooleanUtils,(MSIG)
      ↪ toBoolean(String)0,(I)0
10 TYPE:Boolean
11 VALUE:true
```

Figure 6: The outputs of running the faulty program and the patched program (both instrumented) against the test method in Figure 3 (the input).

DiffTGen runs the output-instrumented versions of $faultprog$ and $patchprog$ against the test method shown in Figure 3 to obtain two outputs in Figure 6. (To do so, DiffTGen first removes the test method's annotation *@Test* and runs a class containing a main method where the test method is called.) The outputs basically show that for the first execution (indicated by *(E)0* at Lines 3 & 9) of the `toBoolean` method in the `BooleanUtils` class, the return values (indicated by *(I)0* at Lines 3 & 9) are different: one being `false` and the other being `true`. In the next stage, DiffTGen produces a test case based on the two different outputs.

2.3 Test Case Generation

In the third stage, DiffTGen compares the two outputs generated in the previous stage to identify specific values that are different, and then asks the oracle to tell which is correct. If the value generated by $patchprog$ is incorrect, DiffTGen determines $patchprog$ to be overfitting with a test case generated.

Given the generated output strings shown in Figure 6, DiffTGen found that output values (at Lines 5&11) are different and are comparable since their location properties (the *PRIM_LOC* values at Lines 3&9) are the same. DiffTGen then asks an oracle to determine which output value is correct. For this example, we used the fixed version of the faulty program (the manually fixed version available in the Defects4J dataset) and found that the output value of $faultprog$ (which is *false*) is correct but the output value of $patchprog$ (which is *true*) is incorrect. (To do so, we created an output-instrumented version for the fixed version and ran it against the test method to obtain the expected output. In general, a human oracle would be needed and DiffTGen needs to be amenable to a human. We leave the research of involving a human oracle for test

```
1 @Test public void test078() throws Throwable  {
2   BooleanUtils.clearORefMap();
3   boolean boolean0 = BooleanUtils.toBoolean("@es");
4   List obj_list_7au3e = (List) BooleanUtils.oref_map
5   .get("toBoolean(String)0");
6   Object target_obj_7au3e = obj_list_7au3e.get(0);
7   assertFalse(
8   "(E)0,(C)org.apache.commons.lang.BooleanUtils," +
9   "(MSIG)toBoolean(String)0,(I)0",
10  ((Boolean) target_obj_7au3e).booleanValue());
11 }
```

Figure 7: Test Case Generated by DiffTGen

case generation as our future work.)

With the expected output provided by an oracle, DiffTGen creates a test-case-instrumented version for $faultprog$ (Figure 5) and produces a test case (Figure 7) by augmenting the test method with an assertion statement and other statements for creating the assertion. In the test-case-intrumented version of $faultprog$, DiffTGen saves the reference to the object `o_7au3e`, the target object whose value to be asserted, in a static map field *oref_map* in the class of `toBoolean`. In the test case (Figure 7), DiffTGen creates two statements (Lines 4-6) obtaining the target object and one assertion statement (Lines 7-10) asserting the value to be *false* as expected. More details can be found in Section 3.4. DiffTGen finally reports the patch to be overfitting with the generated test case as an evidence.

3. METHODOLOGY

In this section, we first give the definition of an overfitting patch, and then elaborate on the three stages that DiffTGen takes to identify an overfitting patch with a test case generated.

3.1 The Definition of an Overfitting Patch

Let $faultprog$ be the faulty program and I be the input domain of $faultprog$. I can be divided into two sub-domains I_0 and I_1 on which $faultprog$ has the *correct* and *incorrect* behaviors respectively. Let $fixprog$ be a correct version of $faultprog$ that only repairs the bug and does not contain any new features. Assuming both programs are deterministic, then we have $\forall i_0 \in I_0.faultprog(i_0) = fixprog(i_0) \wedge \forall i_1 \in I_1.faultprog(i_1) \neq fixprog(i_1)$ where we use $p(i)$ to denote the program behavior of p on a specific input i. Let $patchprog$ be a patched program that was generated by a repair technique and can pass a test suite that $faultprog$ failed. Assuming $patchprog$ is also deterministic, then we have $\exists i_1 \in I_1.patchprog(i_1) = fixprog(i_1)$. A repair technique can produce an overfitting patch which does not actually repair the bug. An overfitting patch (or a patched program) can be categorized into two types:

- **Overfitting-1**: The patch repairs *some (or even all)* of the incorrect behaviors of the original program but *breaks* some of its correct behaviors.
- **Overfitting-2**: The patch repairs *some (but not all)* of the incorrect behaviors of the original program and *does not break* any of its correct behaviors.

For a $patchprog$ that is overfitting-1, we have

$$\exists i_0 \in I_0.\exists i_1 \in I_1.patchprog(i_0) \neq fixprog(i_0) \wedge patchprog(i_1) = fixprog(i_1)$$

For a $patchprog$ that is overfitting-2, we have

$$\forall i_0 \in I_0.patchprog(i_0) = fixprog(i_0) \wedge \exists i_{10} \in I_1.\exists i_{11} \in I_1.i_{10} \neq i_{11} \wedge patchprog(i_{10}) \neq fixprog(i_{10}) \wedge patchprog(i_{11}) = fixprog(i_{11})$$

```
1 int faultprog(int x) {
2   if (x < 999) { x++; } //the faulty statement
3   return x; }
4 int patchprog(int x) {
5   if (x < 1000) { x++; } //the patched statement
6   return x; }
7 int targetprog(int x) {
8   if ((!(x<999)&&(x<1000))||((x<999)&&!(x<1000))){
9     int delta_syn_3nz5e_0 = -1; } //a dummy statement
10  if (x < 1000) { x++; }
11  return x; }
```

Figure 8: A Test Target Example

Our definition is consistent with the definition of a bad fix given by Gu et al. [6]: a bad fix either introduces disruptions (regressions) or does not cover all the bug-triggering inputs or both[2]. A patched program that is overfitting-1 introduces regressions and is not acceptable, but a patched program that is overfitting-2 does not introduce regressions (though it only makes a partial repair) and may thus be considered as still valid. DiffTGen can identify a patched program[3] to be overfitting-1 by finding an input that exposes a semantic difference between $faultprog$ and $patchprog$ and further showing the semantics of $patchprog$ is incorrect while the semantics of $faultprog$ is correct with the assistance of an oracle. However, it cannot directly identify a patched program to be overfitting-2. Identifying such a patched program involves two steps: (1) showing $\exists i_1 \in I_1.patchprog(i_1) \neq fixprog(i_1)$ (**Overfitting-2a**) and (2) showing $\forall i_0 \in I_0.patchprog(i_0) = fixprog(i_0)$ (**Overfitting-2b**). DiffTGen can achieve (1) by finding a test input exposing a semantic difference between $faultprog$ and $patchprog$ and further showing the semantics are both incorrect[4]. However, it cannot achieve (2) by proving the patched program contains no regressions.

3.2 Test Target Generation

In the first stage, DiffTGen creates a test target program, or $targetprog$, based on the syntactic differences Δ_{syn} between $faultprog$ and $patchprog$. $targetprog$ is the program on which a test generator later works to generate test inputs that uncover semantic differences between $faultprog$ and $patchprog$.

DiffTGen creates $targetprog$ by extending $patchprog$ with dummy statements inserted (one for each δ_{syn}). The inserted dummy statements do nothing but can be detected by a test generator as the coverage goals. DiffTGen inserts dummy statements into $targetprog$ in such a way that at least a dummy statement would be executed if and only if the execution of $faultprog$ and $patchprog$ would differ.

For simple cases, where a patching modification δ_{syn} does not involve modifying an if-condition (e.g., it modifies an assignment), DiffTGen simply creates a dummy statement and inserts it in front of the modified statement (for insertion or replacement), or in place of the deleted statement (for deletion) in $patchprog$. If a generated test input can cover the dummy statement upon execution, the input would cover the modified statement in $patchprog$ but the unmodified statement in $faultprog$, and would thus lead to a differential execution between $faultprog$ and $patchprog$.

The more complicated cases arise when δ_{syn} is related to an if-statement and effectively modifies an if-condition. (This is a common situation [19, 26]. In fact, there exist repair techniques that only look at condition-related bugs [38, 39].) In such cases, it might be ineffective just to insert a dummy statement in front of an

[2]In our definition, we consider such a patch to be overfitting-1.

[3]A patched program is known to have something repaired, since it passed the test suite that the original program failed.

[4]Note that DiffTGen does not find an input showing the semantics of the two programs are identical but incorrect. Such an input is not directly related to what changes a patch makes.

if-statement whose condition is modified. Figure 8 is an example where the faulty program `faultprog` and the patched program `patchprog` are shown at the top and in the middle. The faulty if-condition `x<999` at Line 2 was changed to `x<1000` at Line 5. If DiffTGen simply creates a dummy statement and inserts it before the if-statement at Line 5 as the coverage goal, then a test generator could quite possibly end up with finding an input `x` taking a random value, say 33, to make the dummy statement covered. However, such an input can expose no semantic difference between the two programs.

To address the problem, DiffTGen creates a synthesized if-statement and inserts it before the modified statement or at the modification place in $targetprog$. The new if-statement contains a new if-condition. It also contains a dummy statement as its then-branch. The advantage of such a synthesized if-statement is as follows: a generated test input that can cover the dummy statement would expose different branch-taking behaviors between the unmodified statement in $faultprog$ and the modified statement in $patchprog$. For example, in Figure 8, DiffTGen creates a synthesized if-statement starting at Line 8. For the dummy statement at Line 9 to be covered, a test input x has to satisfy the condition at Line 8 which is essentially `x==999`. Such an input can expose different branch-taking behaviors between $faultprog$ and $patchprog$: Given `x==999`, $faultprog$ does not execute its then-branch `x++`, but $patchprog$ does. This input further exposes a semantic difference between the two programs, the return value of $faultprog$ is 999, but the return value of $patchprog$ is 1000.

DiffTGen considers in total 10 different types of modifications to produce dummy statements to be inserted in $targetprog$. Table 1 shows the 10 cases with code examples. The three cases *Non-partial-if Insertion*, *Non-partial-if Deletion*, and *Other Change* cover the simple cases where DiffTGen simply inserts dummy statements into $patchprog$ to produce $targetprog$. For each of the other cases where the modification effectively changes an if-condition, DiffTGen creates a synthesized if-statement to be inserted in $patchprog$. (Note that some of the cases can be considered as changing if-conditions. For example, inserting a partial if-statement `if(c)s` can be considered as changing the condition of an if-statement `if(false)s` from `false` to `c`.) To create a target program, for each $\delta_{syn} \in \Delta_{syn}$, DiffTGen looks at the 10 change cases in the same ordering as listed in Table 1 (from top to bottom), finds the first change case that is matched, produces the new statement, and inserts it in $targetprog$.

3.3 Test Method Generation

In this stage, DiffTGen employs a test generator EvoSuite to generate test methods (test inputs) for $targetprog$ with at least one of the coverage goals satisfied (i.e., with at least one of the dummy statements covered). Such a test method can exercise at least a δ_{syn} and can cause the executions of $faultprog$ and $pathprog$ to differ. However, the test method may not be able to expose any semantic difference δ_{sem} between the two programs. To determine whether a test method can expose a δ_{sem}, DiffTGen creates instrumented versions of $faultprog$ and $patchprog$, runs the two instrumeted versions against the test method to obtain running outputs, and compares the outputs. We call such an instrumented program on which DiffTGen executes to obtain outputs an *output-instrumented* program. For the rest of the section, we focus on explaining how to create an output-instrumented version of a program.

3.3.1 Creating an Output-Instrumented Version

DiffTGen needs to be able to detect whether a given test run exposes a semantic change between $faultprog$ and $patchprog$. In the simplest case, a test method (as a test driver) runs a patched method directly and any difference is seen in the return value of the method. However, real-world patches are seldom that simple: a test method might call other methods which in turn call the patched method; the difference between two executions might not be reflected in the return value, but might be reflected in a changed field accessible from an argument passed to the method.

To accommodate these various possibilities, DiffTGen creates an output-instrumented version of a program by augmenting the program with printing statements. We assume a patching modification is made within a method and a semantic change can propagate to the "input" and "output" elements of the method. We define the input elements of a method to be the arguments (including the *this* argument) that are passed to the method on entry, and we define the output elements to be the return value and any exceptions thrown on exit[5]. For each $\delta_{syn} \in \Delta_{syn}$, DiffTGen looks at the input and output elements of the method that δ_{syn} is involved (also called the *delta-related* method), and prints the values of the elements and the types. (Note that DiffTGen does not print any input argument that is of a primitive type, a *String* type, or is passed as a *final* type, since a change cannot propagate to such an argument after the method execution.)

DiffTGen actually calls a printer (*FieldPrinter*) that we created to print values and types. For an element that is of a primitive type or a *String* type, the printer simply prints its value and type; For an element that is an array, a list, a set or a map, the printer creates a list for the element, and prints the list elements in turn; For an element that is of a Java *Throwable* type, the printer calls the *toString* method and prints the returned string as the value, and it prints the keyword "Throwable" as the type; For an element that is of other types, the printer uses Java reflection[6] to explore the structure of the element[7] (as an object) and prints the fields in a depth-first approach for which we use 5 as the maximum depth for exploration.

At the implementation level, for each delta-related method m_δ, DiffTGen creates a stub method m'_δ whose method signature, method name, and parameter names are equal to those of m_δ. DiffTGen then renames m_δ. In m'_δ, it creates a statement calling the renamed m_δ in a try statement. After calling m_δ, DiffTGen creates statements calling *FieldPrinter.print* to print the input and output elements of m_δ. In the catch clause, it creates a statement printing the thrown exception. The printer accepts six arguments. The first argument is the element to be printed. The printer either simply prints the value of the element and its type or explores the element's internal structure to print a sequence of values and the corresponding types. For each value, the printer also prints the retrieval information showing how the value can be retrieved from an execution (e.g., indicating the printed value is the return value of the method in its first execution). For printing the retrieval information, the printer also accepts as arguments the call count (which is associated with m'_δ), the class name, the extended method signature (which is a string consisting of the method signature of m_δ and the call count), and the property of the element to be printed (indicating, e.g., it is a return value). The final argument the printer accepts is the maximum printing depth (we use 5). In the finally clause, DiffTGen creates a statement increasing the call count. In m'_δ, DiffTGen also creates other statements that define variables

[5]Note that a change can also propagate to a static class field which currently we do not handle. We consider handling this type of changes as part of our future work.

[6]We use FieldUtils from the apache package commons-lang3-3.5.

[7]DiffTGen ignores an element that is declared to be a *final* or a *static* type which usually does not contain a semantic change.

Table 1: Test Target Generation of 10 Change Cases

change case	faultstmt	patchstmt	targetprog
Partial-if Insertion	null	if(c){s}	patchprog with if(c){dummystmt} inserted before patchstmt
Non-parial-if Insertion	null	s	patchprog with dummystmt inserted before patchstmt
Partial-if Deletion	if(c){s}	null	patchprog with if(c){dummystmt} inserted where faultstmt is deleted
Non-partial-if Deletion	s	null	patchprog with dummystmt inserted where faultstmt is deleted
If-Guard Insertion	s	if(c){s}	patchprog with if(!c){dummystmt} inserted before patchstmt
If-Guard Deletion	if(c){s}	s	patchprog with if(!c){dummystmt} inserted before patchstmt
If-Cond Change	if(c1){s}	if(c2){s}	patchprog with if(!c1&&c2||c1&&!c2){dummystmt} inserted before patchstmt
If-Cond-Else Change	if(c1){s}	if(c2){s}else{e2}	patchprog with if(!(c1&&c2)){dummystmt} inserted before patchstmt
If-Cond-Then Change	if(c1){s1}else{e}	if(c2){s2}else{e}	patchprog with if(c1||c2){dummystmt} inserted before patchstmt
Other Change	s1	s2	patchprog with dummystmt inserted before patchstmt

A partial-if-statement does not have an else branch.

and return the final result (if needed).

To obtain outputs, DiffTGen creates a test class, copies each test method (with the annotation *@Test* removed) to the class, creates a main method in the class, and calls the main method to run each test method over the output-instrumented versions of $faultprog$ and $patchprog$. An output is printed in a stylized form so that the corresponding lines can be easily compared.

3.4 Test Case Generation

In the previous stage, DiffTGen runs the output-instrumented versions of $faultprog$ and $patchprog$ against a test method to obtain running outputs. In this stage, DiffTGen compares the outputs to identify specific values that are different, and then asks the oracle to tell which is correct. When the value generated by $patchprog$ is incorrect, DiffTGen determines $patchprog$ to be overfitting. If a correct value could be provided by the oracle, DiffTGen performs two steps to produce a test case: (1) creating a test-case-instrumented version (for the original $faultprog$ for which a test case is created) and (2) augmenting the test method. DiffTGen uses the two steps mainly to create an assertion in the test case that asserts the value (that was checked and compared) to be equal to the expected one provided by the oracle.

3.4.1 Comparing the Running Outputs

DiffTGen compares the running outputs of $faultprog$ and $patchprog$ to identify *comparable* values that are different[8]. Two values are comparable if the two pieces of retrieval information associated with the values (indicating how the values can be generated) are identical. More specifically, DiffTGen goes through the two outputs (as two strings) line by line in parallel. When the two lines examined both start with *VALUE* (e.g., Lines 5&11 in Figure 6), DiffTGen obtains the corresponding value items which we call the *check value*s. DiffTGen also obtains the retrieval information by looking at two lines before the current lines that start with *PRIM_LOC*. We call the corresponding value items the *loc value*s. When the two loc values are identical but the two check values are different, DiffTGen successfully identifies comparable values that are different, and it provides to the oracle (1) the test method, (2) the loc value, (3) the two check values, and (4) the types of the check values (obtained from one line after the check value lines that start with *TYPE*). DiffTGen asks the oracle to determine which value is correct (and if the value types are different, what is the correct type). If neither is correct, DiffTGen further asks the oracle to provide a correct value (possibly with a value type). An oracle may not provide a correct value or a type (correctness judging between two values might not be easy for a human oracle). In that case, DiffTGen discards the current check values and keeps looking for other check values in the outputs. (For our experiments in Section 4, DiffTGen uses a fixed version of $faultprog$ as the oracle.)

3.4.2 Generating a Test Case

Given an expected value (possibly with an expected type) and a loc value used to generate the value to be asserted, DiffTGen produces a test case mainly by augmenting the test method with an assertion statement. To create the assertion statement, DiffTGen needs to do three things: (1) obtain the input/output element to be asserted; (2) obtain the value to be asserted from the input/output element; (3) produce an assertion statement asserting the value to be equal to the expected value.

(2) and (3) are easy to do. Once an input/output element is available, DiffTGen parses the loc value to obtain the access path which it needs to follow to obtain the value to be asserted (or the *target* value). With the access path being ready, DiffTGen uses Java reflection to explore field structure of the element, creates statements that follow the path to obtain the target value syntactically, and inserts the statements in the test method. Then DiffTGen simply creates an assertion statement asserting the target value to equal to the expected value.

The difficulty lies in (1): how to obtain the input/output element to be asserted (or the *target* element). For the simple test method as shown in Figure 3, the target element (i.e., the return value `boolean0`) is syntactically available. In general, however, the target element might not be syntactically available in the test method: consider the case where the delta-related method (where a patching modification is made) is a private method called by a public method called in the test method. To still be able to syntactically obtain the target element in the test method, DiffTGen creates an instrumented version of $faultprog$, which we call the test-case-instrumented version, that keeps track of the input/output elements of a delta-related method by storing the elements in a map (as a static field of the method's located class). Later, to syntactically obtain an input/output element in the test method, DiffTGen simply creates a statement that refers to the field map to get the element.

For the rest of the section, we first explain how to create a test-case-instrumented version of a program, then explain how to augment a test method to produce a test case.

3.4.2.1 Creating a Test-Case-Instrumented Version.

In a test-case-instrumented version, the parent class of each delta-related method contains a static field map named *oref_map* that stores the input and output elements of the method. The key of the map is a string consisting of the signature of the delta-related method and a call count associated with the method (i.e., the *extended method signature*). The value of the map is a list of the input/output elements.

Creating a test-case-instrumented version is similar to creating an output-instrumented version: DiffTGen looks at each delta-related method m_δ (where a patching modification is made), creates a stub method m'_δ, renames m_δ, and creates a try-statement within m'_δ where m_δ is called. Here, after this method call, instead of creating statements printing the input/output elements, DiffTGen creates statements calling a static method *addToORefMap* it creates to store the elements in the map *oref_map*. *addToORefMap* accepts two arguments: (1) the extended method signature of m_δ (before it

[8]DiffTGen currently does not produce any test case based on output values that are not comparable.

```
1 Object target_obj_7au3e = null;
2 boolean not_thrown = false;
3 try {
4   <CLASS NAME>.clearORefMap();
5   <TESTING CODE GENERATED BY EVOSUITE>
6   not_thrown = true;
7   fail();
8 } catch (Throwable t) {
9   if (not_thrown) { fail("Throwable␣Expected!"); }
10  else {
11    target_obj_7au3e=...;//get the input/output element
12    assertEquals(<MESSAGE>, <EXPECTED VALUE>,
13      ((Throwable) target_obj_7au3e).toString()); }}}
```

Figure 9: Augmenting a Test Method with an Expected Throwable

is renamed, as a key stored in *oref_map*) and (2) the input/output element (stored in a list as the value of the key). DiffTGen calls *addToORefMap* to store in a list the return value, the *this* argument, and the method arguments in turn. (If an element is not available, it stores *null*.) Similarly, in the catch clause, DiffTGen calls *addToORefMap* to store the thrown exception.

3.4.2.2 Augmenting the Test Method.

Given the expected value provided by the oracle, the loc value obtained from the running output, and a test-case-instrumented version created, DiffTGen finally produces a test case by augmenting the test method. In the test case, DiffTGen mainly creates statements that (1) syntactically obtain the target element (i.e., the input/output element to be asserted) by referring to the static field map (*oref_map*) created in the test-case-instrumented version, (2) syntactically obtain the target value (i.e., the value to be asserted) by following the access path contained in the loc value to explore the target element, and (3) assert the target value to be equal to the expected value.

More specifically, DiffTGen creates a test case whose method signature is identical to that of the test method and contains an extra label *@Test*. In the test case, DiffTGen first creates a statement clearing the map *oref_map* contained in the test-case-instrumented version. Next it copies all the statements from the test method. Next it creates statements to syntactically obtain the target element by referring to *oref_map* using the extended method signature and the property value it obtained from the loc value (the property value is actually the index of the target element in the element list stored in *oref_map*). Next it creates statements to syntactically obtain the target value from the target element. Again, when the target element is not of a primitive, a *String*, or a *Throwable* type, DiffTGen uses Java reflection to explore the structure of the target element and follows the access path (contained in the loc value) to get the target value. Finally, DiffTGen creates a *JUnit* statement asserting the target value to be equal to the expected value.

Note that when the element to be asserted is an exception, DiffTGen uses the template shown in Figure 9 to produce a test case. Essentially, DiffTGen creates a try statement and copies the testing statements from the test method to the try-body. DiffTGen creates the augmented statements in the catch clause.

4. EMPIRICAL EVALUATION

To empirically evaluate the effectiveness of DiffTGen, we ask two questions:

- **RQ1**: Could DiffTGen identify overfitting patches generated by automatic repair tools? What is its performance?
- **RQ2**: Could DiffTGen enhance the reliability of an automatic repair technique and guide the technique to produce correct patches?

We conducted two experiments to answer the two questions. We next show each experiment in turn.

4.1 RQ1

To evaluate the performance of DiffTGen in identifying overfitting patches, we created a patch dataset containing 89 patches (the patched programs) generated by four automatic repair techniques for Java: jGenProg [20], jKali [20], NoPol [39] and HDRepair [1] with 10 of the patches being correct (see Section 4.1.1). We ran DiffTGen on each patched program and its original faulty program. Our results show that DiffTGen found 39 out of the 79 (89-10) patches (49.4%) to be overfitting with the corresponding test cases generated.

4.1.1 Experimental Setup

Patch Dataset. The current implementation of DiffTGen is in Java. To evaluate its performance, we collected all patches generated by four automatic repair techniques: jGenProg, jKali, NoPol and HDRepair for bugs in the Defects4J dataset [8] which is commonly used for evaluating an automatic repair technique for Java. Martinez et al. did an experiment [18] running three repair tools: jGenProg, jKali and NoPol on the Defects4J bugs and generated in total 84 patches. We included all these patches in our dataset. For patches generated by HDRepair, we contacted Le et al. (the authors of [14]) and obtained a set of 14 patches (for each of the 14 repaired bugs, we used the first found patch reported by HDRepair). We also included these patches in our dataset. Among the 84+14=98 patches, we found 9 patches are syntactically repetitive. We removed them and obtained a final dataset containing 89 individual patches[9]. It turns out each patch makes only a small change on only one statement.

Among the 89 patches, we determined 10 patches to be correct by syntactically comparing each of the 89 patches against the correct human patch (the fixed version) associated with the bug in the dataset (the syntactic comparisons are easy and the correctness of these 10 patches are obvious, see our provided links below for what they are). For the remaining 79 patches, we consider them as possibly incorrect[10].

DiffTGen. To test if a patch is overfitting or not, we ran DiffTGen with the faulty program $faultprog$, the patched program $patchprog$, and the syntactic changes between the two as input. For a syntactic change, we manually identified the two change-related statements from $faultprog$ and $patchprog$ respectively. As the oracle, we used the human-patched program (the fixed version) in the Defects4J bug dataset associated with the bug[11]. For correctness judging, DiffTGen created an output-instrumented version for a bug's fixed version and ran it against any test method generated by EvoSuite twice to mark any printed fields whose values are inconsistent during the two executions. DiffTGen considers the marked fields as non-deterministic and does not use them for test case generation. By using a fixed version of the bug as the oracle, DiffTGen runs automatically to produce a test case.

DiffTGen employs *EvoSuite-1.0.2* to generate test methods. (We did not use EvoSuite's functionality to generate assertions in a test method because we found the generated assertions often do not ex-

[9]See github.com/qixin5/DiffTGen/tree/master/expt0/dataset for all the 89 patches we used (including the ones we identified to be correct) and the patches we removed.

[10]It is not easy to determine the correctness of the 79 patches by hand since they are not syntactically identical to the corresponding human patches (this is a reason why a tool like DiffTGen is needed). The rate of overfitting patches identified by DiffTGen (49.4%) is actually a lower bound.

[11]Note that the human patches only make changes about bug repairs and do not add any new features for the original bugs. This makes sure a test case generated by DiffTGen specifies the correct behavior of a bug but not any new features expected.

Table 2: The Running Result of DiffTGen (#Bugs: 89, #Bugs that are likely to be incorrect: 79)

Running Setup	Time	#SynDiff	#SemDiff	#Overfitting	#Regression (Overfitting-1)	#Defective (Overfitting-2a)
trial30_time60	6.9m	72	61	39	34	18
trial10_time180	8.0m	73	59	36	32	13
trial3_time600	11.4m	73	56	32	28	12
trial1_time1800	30.6m	69	48	27	23	8

For a running setup, DiffTGen ran the trials in parallel.
Note that DiffTGen cannot identify an Overfitting-2 patch, but can identify a patch to have an Overfitting-2a behavior.

pose any semantic differences between $faultprog$ and $patchprog$.) EvoSuite uses an evolutionary search algorithm and allows the user to specify a searching timeout. For our experiments, as the default setup, DiffTGen generates test methods by calling EvoSuite in 30 trials with the searching timeout being 60s for each trial (or the setup *trial30_time60*). We implemented DiffTGen to run the trials in parallel. In Section 4.1.2, we compared the performances of DiffTGen running in different setups. We ran all the experiments on a machine with 8 AMD Opteron 6282 SE processors and 8G memory.

4.1.2 Results

The Performance of DiffTGen. DiffTGen's running result can be found in Table 2[12] (the first row corresponds to the default running setup). From left to right, the table shows the running setup (*Running Setup*); the average running time in minutes (*Time*); the numbers of bugs for which the syntactic difference between the two programs (the patch and the bug) has been exercised (*#SynDiff*); a semantic difference between the two programs has been found (*#SemDiff*); overfitting-indicative test cases have been generated (*#Overfitting*); regression-indicative test cases have been generated (*#Regression*); and defective-indicative test cases (the semantics of the two programs are both incorrect) have been generated (*#Defective*). Note that we consider the time duration of a run to be from the start of the run to the time when an overfitting-indicative test case is generated or when DiffTGen terminates with no such test case is generated (but we did not actually stop running DiffTGen until it terminated).

As shown, DiffTGen identified 39 patches to be overfitting (see Table 3 for what they are) with the corresponding test cases generated. For 34 patches, DiffTGen generated test cases showing they contain regressions (i.e., showing the semantics of the patch is incorrect but that of the bug is correct). For 18 patches, DiffTGen generated test cases showing they are defective (i.e., the semantics of $faultprog$ and $patchprog$ are both incorrect). Note that DiffTGen could generate two different test cases for a patch showing it not only contains regressions but also is defective. This explains why the sum of the last two columns in Table 2 can be greater than the fifth column. Our results show that DiffTGen is efficient: it takes about 7 minutes on average to test a patch (with or without test cases generated).

For 72 (80.9%) patches, DiffTGen found at least a test method for each patch that exercises the syntactic change between the patched and the original programs. For the other 17 patches, we found EvoSuite generated no test method at all for 4 of the patches. This could happen either because EvoSuite failed to generate anything within the time limit or because an error occurred during its run. For the other 13 of the 17 patches, although EvoSuite generated test methods, they do not exercise the syntactic changes and would not be useful to reveal any semantic differences. We think the reason could be that the overall goal of EvoSuite is to generate test methods to achieve a high coverage of the class under test, and it is not designed to generate test methods to cover a certain statement in particular.

A test method that exercises the syntactic change may or may not reveal a semantic difference. Using the underlying search algorithm of EvoSuite plus the synthesized if-statements created in the test target, for 84.7% (61/72) of the patches, DiffTGen obtained test methods that uncover some semantic differences. For 11 of the 72 patches, however, the test methods do not reveal any semantic difference. In general, finding a test that uncovers a semantic difference between two programs is undecidable: there could be a large number of paths exercising a syntactic change but only a small fraction of them may reveal a real semantic difference. Below is an example. For the bug *Chart_1*, jGenProg creates a patch by deleting the first if-statement.

```
1  if (dataset != null) { return result; } //Patch by deletion.
2  ... //The code here may change the value of "result".
3  //But no change would be made if "dataset" is empty & non-null.
4  return result;
```

To test the patch, DiffTGen creates a test target program by inserting a newly synthesized if-statement (*if(dataset!=null){dummystmt}*) at Line 1 (i.e., at the place where the original if-statement is deleted). Using EvoSuite, DiffTGen found a test method which initializes `dataset` to be a new empty object (non-null), and the dummy statement is exercised. However, since the object is empty, no changes are made to `result`, and no semantic difference is actually made (the code for this is not shown). The problem here is that the search algorithm of EvoSuite tends to build "simple" objects to satisfy its coverage goals. A simple object here (the empty `dataset`) would not reveal any semantic difference although the syntactic difference is exercised.

When a semantic difference is found, DiffTGen asks the oracle for semantic checking. DiffTGen found 34 patches that contain regressions with the corresponding test cases generated. For 18 patches, DiffTGen generated test cases showing they are defective (the outputs of $faultprog$ and $patchprog$ are both incorrect), though they may or may not contain regressions. We use a simple example shown below to explain this.

```
1 int foo(int x) {
2   x = x + 1; //Bug. Should be x = x * 2;
3   //x = x + 2; (Patch)
4   return x; }
```

A patch changes the buggy statement at Line 2 to a new statement at Line 3. The patched program works fine for an input $x = 2$. We know it contains regressions because the program fails for an input $x = 1$ for which the original program works fine. But we also know the patched program is generally defective, since for many other inputs (e.g., when $x = 3$), both the original and the patched programs fail.

There are 22 cases where the found semantic differences do not reveal any overfitting properties of a patch. For 5 cases, DiffTGen only produced repair-indicative test cases (we found they correspond to the correct patches, for the other 5 correct patches, DiffTGen does not produce any test cases). For 17 cases, the semantic differences are not interesting or cannot be leveraged by DiffTGen for semantic checking. For example, the semantic difference between the faulty program and a patched program generated by jGenProg (for *Chart_7*) is related to a class field named *time* whose type is *long*. Such a field is time-related, and is not reliable for semantic checking. DiffTGen runs the oracle program twice to identify such fields and refuses to use them for semantic checking. For this example, DiffTGen generated no test cases. There are also forms of semantic changes that DiffTGen currently does not support for correctness judging. For example, a list has one more element added in the patched program. Since the values of the new element added has no loc values matched in the faulty program (see

[12]Due to the space limit, we only show a summary of the results in Table 2. The complete result tables can be found at https://github.com/qixin5/DiffTGen/tree/master/expt0/result.

Table 3: 39 Overfitting Patches Identified by DiffTGen

Repair Tool	Bug ID
jGenProg	M2, Ch3∗, M40, Ch5, M80∗, Ch15, M78, T4, M8, M95, M81
jKali	M32, M2, Ch13, Ch26, M40, Ch5, Ch15, T4, M95, M81, M80
NoPol	Ch21, L51, L53, M33†, Ch13, M40, M87, M97, M57, M104, Ch5, M80∗, M105, M81∗
HDRepair	C10†, L6†, M50∗†

Ch: Chart, C: Closure, L: Lang, M: Math, T: Time.
ID with †: The correct patch exists in the tool's search space.
ID with ∗: Only defective-indicative test cases were generated.

Table 4: Repair Experiment 0

ID	Time	#Patch	#SynDiff Patch	#Correct Patch
Math_95_jGenProg	1.8m	10	2∗	0
Chart_15_jKali	28.7m	5	1	0
Chart_26_jKali	81.2m	2	1	0
Chart_13_Nopol	3.3m	10	1	0
Math_50_HDRepair	88m	7	6	4

∗: The two generated patches are invalid since they do not pass the test cases generated by DiffTGen. We believe it is a failure of jGenProg.

Section 3.4.1 for how DiffTGen does correctness judging), DiffTGen would not produce any test case based on the new element which causes the semantics to be different.

Setup Comparison. DiffTGen employs EvoSuite to generate test methods. To do so, EvoSuite uses evolutionary algorithms. To investigate how EvoSuite affects DiffTGen's results, we compared the default setup of DiffTGen *trial30_time60* (i.e., running EvoSuite in 30 trials with the search time of each trial limited to 60s) to three other setups: *trial1_time1800*, *trial3_time600* and *trial10_time180* (we limit EvoSuite's overall search time to be 30 minutes to have these setups created). As the results in Table 2 show, DiffTGen needs to run EvoSuite in more than one trial to obtain better results. Sacrificing the search time (e.g., from 600s to 60s) for more trials (e.g., from 3 to 30) would cause the number of change-exercised test methods to slightly decrease (from 73 to 72) but would enhance the overall testing performance: the running time reduces (by about 40%) and the number of generated overfitting-indicative test cases increases (from 32 to 39).

4.2 RQ2

DiffTGen identified 39 patches to be overfitting with test cases generated. In the context of automatic program repair, we want to know whether DiffTGen could work together with an automatic repair technique to make the repair technique avoid generating overfitting patches and produce correct patches eventually. So in this experiment, we ran the four repair tools (jGenProg, jKali, NoPol and HDRepair) on the 39 bugs for which DiffTGen generated new test cases showing the original patches are overfitting (we augmented the corresponding test suites associated with the bugs with the new test cases). If new patches were generated, we ran DiffTGen again, and if new test cases were generated, we augmented the test suites and ran the repair techniques again, so on and so forth.

Figure 10 is a summary of the results. It shows that the repair techniques with DiffTGen configured avoid yielding any incorrect patches for 36 bugs eventually. For 33 of the 36 bugs, we find that there do not exist correct patches in the repair tools' search spaces. So the best the tools can do is to yield no patches, and DiffTGen makes them achieve that. For 3 of the bugs (*Math_33_Nopol*, *Closure_10_HDRepair* and *Lang_6_HDRepair*), the corresponding repair tools could potentially produce a correct patch, but they did not since their search spaces of patches are too large and the correct patches were not actually found. For *Math_50*, HDRepair eventually produced four correct patches with the assistance of DiffTGen (see Section 4.2.3). For 3 of the 39 bugs (*Math_95_jGenProg*, *Chart_13_Nopol* and *Math_50_HDRepair*), there were incorrect patches generated eventually. jGenProg produced two invalid patches for *Math_95* which did not pass the test cases generated by DiffTGen. DiffTGen failed to generate overfitting-indicative test cases for three patches: *Chart_13_Nopol*, *Math_50_HDRepair_0* and *Math_50_HDRepair_1* which are overfitting and incorrect.

Table 5: DiffTGen Experiment 0

ID	Time	SynDiff	SemDiff	Overfitting	Regression (Overfitting-1)	Defective (Overfitting-2a)
Chart_26_jKali	23.4m	true	true	true∗	true	false
Chart_15_jKali	22.8m	true	true	true∗	true	false
Chart_13_Nopol	11.0m	true	true	false	false	false
Math_50_HDRepair_0	11.2m	true	false	false	false	false
Math_50_HDRepair_1	16.7m	true	true	false	false	false

∗: A following repair experiment shows that jKali failed to produce any patches using the test suite augmented with the newly generated overfitting-indicative test case.

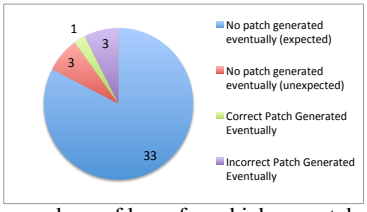

Figure 10: The numbers of bugs for which no patches (expected or unexpected), correct patches and incorrect patches were eventually generated. (For *Math_50_HDRepair*, both correct and incorrect patches were generated.)

4.2.1 Experimental Setup

For each patch in Table 3, DiffTGen generated an overfitting-indicative test case. We added the test case to the test suite associated with the bug and obtained an augmented test suite (if multiple test cases have been generated for a patch, we added the one showing the patch contains regressions). For each bug, we next ran the repair technique (the one produced its initial patch) with the augmented test suite to try to find a new patch. For each of the four repair techniques, we ran it in 10 trials with the time limit being two hours for each trial. The original repair experiments reported in [14] ran HDRepair to repair a bug with a buggy method provided manually. To be consistent, we provided HDRepair with same buggy methods provided in [1] for repairing three of the bugs *Closure_10*, *Lang_6*, and *Math_50*. For any new patches generated, we ran DiffTGen again to generate new test cases. In this experiment, we used the default setup of DiffTGen for test case generation. Currently, we do not have an integrated version of a repair technique and DiffTGen. So each time we ran a repair technique, we manually added the newly generated test case to the test suite, and each time we ran DiffTGen, we manually provided it with the syntactic changes that the patch makes.

4.2.2 The Potential Of Producing a Correct Patch

We analyzed the fixed version (the human patch) for each of the bugs listed in Table 3 and found that for only 4 bugs (marked with †), the corresponding repair techniques could potentially produce correct patches. For the other bugs, the correct patches do not exist in the tools' search spaces. We find that jGenProg, jKali and NoPol have their own limitations. jGenProg often fails to produce a correct patch if the fix statements do not exist in the original faulty program. jKali can only produce patches that remove statements. NoPol can only repair an if-condition-related bug whose fix needs a simple change (on only one condition). Compared to the other techniques, HDRepair could potentially generate correct patches for its three bugs. Its search space is much larger, but it leverages historical repair data to make the search guided.

4.2.3 Results

As Figure 10 shows, there are in total 36 bugs for which no patches were generated by the repair techniques. For 33 of the bugs, the corresponding repair techniques do not have the abilities in producing correct patches, and the fact that no patches were eventually generated is expected. For three of the bugs (*Math_33_Nopol*, *Clo-*

sure_10_HDRepair and *Lang_6_HDRepair*), although the corresponding repair techniques can potentially produce correct patches, they failed to do so since the search spaces are large and the correct patches were not actually found.

For 5 of the bugs, there were patches generated by the repair techniques. In Table 4, the first column shows the bugs and the repair techniques. The fourth column shows that there were in total 11 different patches generated. Among the 11 patches, we found 4 patches generated by HDRepair for *Math_50* are correct: they essentially remove the faulty statement *x0 = 0.5 * (x0 + x1 - delta)* (see https://github.com/qixin5/DiffTGen/tree/master/expt1 for these 4 patches, all the other generated patches and all the generated test cases). We also found two patches generated by jGenProg for *Math_95* are invalid: they did not pass the test cases previously generated by DiffTGen. We next ran DiffTGen again for the other five (11-4-2) patches and the corresponding bugs. As the result shown in Table 5, DiffTGen identified two overfitting patches, *Chart_26_jKali* and *Chart_15_jKali*, with the corresponding test cases generated. We added each test case to the bug's test suite, and then ran jKali to repair the two bugs again. This time, no patches were generated by jKali. For the other three patches (*Chart_13_Nopol*, *Math_50_ HDRepair_0 & 1*), we believe they are overfitting and incorrect, but DiffTGen did not produce any overfitting-indicative test cases[13].

4.3 Discussion

We conducted two experiments showing the *feasibilities* of (1) using DiffTGen to identify overfitting patches within a short amount of time (a few minutes) and (2) combining DiffTGen with a repair technique to enhance the technique's reliability.

In the experiments, DiffTGen used a bug-fixed version as the oracle. In general, however, we need a human oracle, and DiffTGen should provide testing information that is human-amenable. This is the research we consider to do to make DiffTGen more practical. Debugging techniques involving a human like [11, 32] provide technical support for how this could be done.

DiffTGen employs EvoSuite to generate test methods. There are cases where EvoSuite failed to generate any test methods exercising any changes that the patch makes. We think using more sophisticated techniques (e.g., [28]) may improve this but may also take more time to run and make DiffTGen less scalable. In the future, we would like to implement some of the relevant techniques for Java and see whether they could enhance the overall performance of DiffTGen. Given the fact that the current version of DiffTGen runs fast, we believe it could always be used for a first trial.

5. RELATED WORK

In the context of automatic program repair, an overfitting patch is indeed a bad fix generated by a repair technique. The bad fix problem has been studied by Gu et al. [6]. They define a bad fix as either not handling all the bug-triggering inputs or introducing new bugs or both. Our definition of an overfitting patch is consistent with their definition. In this paper, we focus on identifying a bad fix in the context of automatic program repair. Our technique DiffTGen can generate test cases (not just test inputs) exposing an overfitting behavior of a patch. Our work is related to existing works (e.g., [40]) that study how a human-made change becomes a bad fix. In our studied context, however, a patch is generated by a repair technique, not a human. The study by Yu et al. [41] investigates whether test case generation can help a repair technique produce more non-overfitting patches. Our work is related but focuses on identifying an overfitting patch.

DiffTGen is related to TESTGEN [12], DiffGen [35] and BERT [7, 24] in employing an external test generator for test generation and comparing the program outputs to identify any semantic differences. Compared to DiffTGen, these three techniques are used for identifying regressions. DiffTGen however could identify not only regressions but also a patch's other overfitting behaviors. The three techniques only report to the user any differential behaviors detected. DiffTGen does so but in addition generates actual test cases. The three techniques were tested on modified programs where the modifications were randomly seeded or human-made. DiffTGen was tested in the context of automatic program repair.

DiffTGen is also related to other regression, differential or patch testing techniques that are based on symbolic execution. DiSE [28] combines static program analysis and directed symbolic execution to find inputs exercising a modification. The differential symbolic execution technique [27] uses method summaries to characterize program semantic behaviors. With the support of a theorem prover, it compares two method summaries to identify semantic differences. eXpress [36] combines dynamic symbolic execution (DSE) and path pruning to generate tests revealing program behavioral differences. KATCH [17] starts with an existing test input that has the "best" potential to cover a modification based on a defined reaching distance. Based on this input, KATCH uses either symbolic execution or definition switching to generate new inputs to cover the modified code. The shadow technique [3, 25] uses concolic execution to find test inputs uncovering the semantic differences between two programs. For each if-condition after a change point in the original program, the technique tries to find test inputs to force the original and the patched programs to have different branch-taking behaviors if the concrete executions do not reveal such behaviors. DiffTGen's synthesized if-statement has a similar idea, but is only applied to the changed if-condition, not all the if-conditions affected by a change. Although the shadow technique that leverages symbolic execution can potentially capture more differing, branch-taking behaviors, it seems expensive. According to [25], it may take a few hours to finish. Compared to the above testing techniques, DiffTGen is designed and has been evaluated in the context of automatic program repair. It performs differential testing, but goes one step further in producing test cases. DiffTGen is more lightweight and has been shown to work fast. Again, it can not only identify regressions but a patch's other overfitting behaviors.

DiffTGen is also broadly related to works [2, 13, 31] doing patch verification. Verification generally means more work than testing, and may need some sort of correctness criterion. Compared to such techniques, a testing technique like DiffTGen seems more appropriate to be used in the context of automatic program repair.

6. CONCLUSION

Automatic program repair techniques often produce overfitting patches which do not actually repair the bugs. In this paper, we presented a patch testing technique DiffTGen which could identify overfitting patches through test case generation. We demonstrated through experiments the feasibility of using DiffTGen in the context of automatic program repair: DiffTGen can identify about a half of the overfitting patches with test cases generated in only a few minutes. An automatic repair technique, if configured with DiffTGen, could produce less overfitting patches and more correct patches. Our future work will look at (1) optimizing DiffTGen with more sophisticated test generation techniques and (2) making DiffTGen more practical by using a human oracle.

[13]Two of the three patches (*C13_Nopol* & *M50_HDRepair0*) make changes on statements created for instrumentation. It could be avoided but involves modifying a repair technique. In the future, we want to do so and see how the results would change.

7. REFERENCES

[1] *The code repository of the history driven program repair technique*. https://github.com/xuanbachle/bugfixes.

[2] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *ICSE*, pages 302–311, 2013.

[3] C. Cadar and H. Palikareva. Shadow symbolic execution for better testing of evolving software. In *ICSE*, pages 432–435, 2014.

[4] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*, pages 416–419, 2011.

[5] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13, 2012.

[6] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *ICSE*, pages 55–64, 2010.

[7] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *ICST*, pages 137–146, 2010.

[8] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440, 2014.

[9] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *ASE*, pages 295–306, 2015.

[10] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

[11] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE*, pages 301–310, 2008.

[12] B. Korel and A. M. Al-Yami. Automated regression test generation. *ACM SIGSOFT Software Engineering Notes*, pages 143–152, 1998.

[13] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *ICSE*, pages 1047–1058, 2014.

[14] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *SANER*, pages 213–224, 2016.

[15] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, pages 166–178, 2015.

[16] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.

[17] P. D. Marinescu and C. Cadar. Katch: high-coverage testing of software patches. In *ESEC/FSE*, pages 235–245, 2013.

[18] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. *Springer Empirical Software Engineering*, 2016.

[19] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Springer Empirical Software Engineering*, 20(1):176–205, 2015.

[20] M. Martinez and M. Monperrus. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA Demonstration Track*, pages 441–444, 2016.

[21] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE*, pages 448–458, 2015.

[22] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701, 2016.

[23] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[24] A. Orso and T. Xie. Bert: Behavioral regression testing. In *WODA*, pages 36–42, 2008.

[25] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: testing for divergences between software versions. In *ICSE*, pages 1181–1192, 2016.

[26] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[27] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *ESEC/FSE*, pages 226–237, 2008.

[28] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, pages 504–515, 2011.

[29] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265, 2014.

[30] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pages 24–36, 2015.

[31] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.

[32] E. Y. Shapiro. *Algorithmic program debugging*. MIT press, 1983.

[33] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *ESEC/FSE*, pages 532–543, 2015.

[34] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury. Anti-patterns in search-based program repair. In *ESEC/FSE*, pages 727–738, 2016.

[35] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *ASE*, pages 407–410, 2008.

[36] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. express: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11, 2011.

[37] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: models and first results. In *ASE*, pages 356–366, 2013.

[38] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, pages 416–426, 2017.

[39] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 43:34–55, 2016.

[40] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ESEC/FSE*, pages 26–36, 2011.

[41] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. Technical Report 1703.00198, Arxiv, 2017.