SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2007; **00**:1–7 (DOI: 10.1002/000)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/000

DiffGen: Automated Unit-Test Generation for Regression Testing

Kunal Taneja[†] and Tao Xie[‡]

Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206

KEY WORDS: Software Testing; Regression Testing; Test Generation, Software Maintenance

SUMMARY

Software systems continue to evolve throughout their lifetime. Maintenance of such evolving systems (that include regression testing) is one of the most expensive activities in software development. We present an approach called DiffGen for automated unit-test generation and checking for regression testing of Java programs. Given two versions of a Java class, our approach instruments the code by adding new branches such that if these branches can be covered by a test generation tool, behavioral differences between the two classes are exposed. DiffGen then uses a coverage-based test generation tool to generate test inputs for covering the added branches to expose behavioral differences. We have evaluated DiffGen on finding behavioral differences between 8 data-structure subjects and their versions synthesized through mutation testing and 13 larger subjects and their versions from the Subject Infrastructure Repository. Experimental results show that our approach can effectively expose many behavioral differences that cannot be exposed by state-of-the-art techniques.

1. Introduction

Software systems continue to evolve throughout their lifetime. Maintaining such evolving systems is one of the most expensive activities in the course of software development. Maintaining these evolving systems involves making different kinds of changes to them. While making changes to software systems, developers need to make sure that the changes being made are intended and do not introduce any unwanted side effect. Regression testing of software systems gives developers such confidence. Regression testing is the activity of re-testing a modified version of a software system. In particular,

[†]E-mail: ktaneja@ncsu.edu
[‡]E-mail: xie@csc.ncsu.edu
Contract/grant sponsor: NSF ; contract/grant number: CCF-0725190

regression testing aims to find behavioral differences between the original and the modified version of a software system. A behavioral difference between two versions of a program can be reflected by the difference between the observable outputs produced by the execution of the same test (referred to as a difference-exposing test) on the two versions. The behavioral differences are exposed by tests that pass for one version but fail for the other version. Developers can then inspect these failing tests to decide whether the failing tests indicate the behavioral differences that the developers intend or these failing tests indicate regression faults. If these failing tests indicate intended behavior, developers can modify the failing tests to make them pass and if the failing tests indicate regression faults, the developers can fix the fault(s) in the program that made these tests fail.

Developers need to have regression tests that can expose behavioral differences between the original version and the modified version of a software system. In some situations (such as in legacy systems), when developers do not have an existing test suite, it is tedious for developers to write regression tests from the scratch. Even if the developers have an existing test suite, the test suite may not be sufficient in regression testing. In particular, the tests in the existing test suite may not be able to detect a fault that the developers introduced while modifying the system because the tests may not exercise the situations in which the two versions may behave differently. Therefore, to detect unintended changes, the developers (or testers) often need to augment the existing test suite with new tests.

Developers can generate new tests using some automated test generation tool. Most such test generation tools do not insert test oracles into the generated test suites. In such situations, developers can detect only behavioral differences in which an exception is thrown in one of the original or modified versions. Indeed, there are test generation tools like `JUnit Factory` [20] that insert test oracles (in the form of assertions) in the generated test suite. However, generating tests even with high structural coverage on both versions of a software system is not sufficient to expose all the behavioral differences between these versions.

Recently, Evans and Savoia [12] addressed the problem of detecting behavioral differences by generating tests (using `JUnit Factory` [20]) that achieve high structural coverage separately for an old and new versions of the software system under test. They detected behavioral differences by cross-running the test suites on the two versions of the program under test. High structural coverage of the two program versions might ensure that the modified part of the program is executed. However, only the execution of the modified part is not sufficient to expose behavioral differences as shown by Voas [37]. The modified program has to be executed under specific circumstances to expose behavioral differences in program states, and these differences need to be propagated to the point where they can be observed. Hence, even if we have a test suite that has full structural coverage of the two program versions under test, we cannot guarantee that the execution of the tests in the test suite can expose the behavioral differences between the two versions.

In this paper, we propose an approach called `DiffGen`[*] that takes two versions of a Java class, and then generates regression tests that check if the observable behaviors of the two classes differ. Each test that fails on execution exposes a modified behavior. To ensure that the modified part of the program is executed, and to increase chances of detecting behavioral differences, we instrument the given program versions by adding new branches in the source code such that the coverage of these branches ensures

[*]An earlier versions of this work are described in a conference short paper presented at ASE 2008 [36] and a workshop paper presented at AST 2007 [42].

that the behavioral differences are exposed. Therefore, if a test generation tool is able to generate tests to cover these branches, behavioral differences can be exposed.

This paper makes the following main contributions:

Approach. We propose an approach for generating regression tests that help in detecting behavioral differences between two versions of a given Java class by checking observable outputs and receiver object states.

Evaluation. We evaluate our approach on detecting behavioral differences between 8 data-structure subjects (taken from a variety of sources) and their versions synthesized through mutation testing and 13 larger subjects and their versions from the Subject Infrastructure Repository [10]. The experimental results show that our approach can effectively expose behavioral differences that cannot be detected by previous state-of-the-art techniques [12] based on achieving high structural coverage on either version separately.

2. Example

We next illustrate our approach for detecting behavioral differences between two versions of a Java class with the aid of an example. `DiffGen` takes as input two versions of a Java class and generates a suite of regression tests. The behavioral differences between the two classes are exposed by executing the generated test suite. Tests that fail are the ones that represent inputs for which the behaviors of the two versions have changed, while the passing tests represent the ones for which the behavior remains the same.

Consider the example of a Binary Search Tree shown in Figure 1. Figure 1 shows an old version of the class `BST`, while in a new version of the class `BST`, the statement at Line 14 (underlined in Figure 1) has been modified to the one shown in the comment at Line 14. In the old version, a new node is inserted to the left subtree, if the key of the root node is *less than* that of the newly inserted node. In the new version, the new node is inserted to the left subtree, if the key of the root node is *less than or equal to* that of the newly inserted node. Even if we generate tests (for both versions of `BST` classes) that cover all the branches in both classes, we may not be able to detect this behavioral change. Suppose that initially the tree contained only one node with Key 5 as shown in Figure 2. Later nodes with Keys 3, 6, 2, and 7 are inserted. In particular, the insertion of a node with Key 3 covers the `if` branch at Lines 22-25. The insertion of a node with Key 6 subsequently covers the `if` branch at Lines 15-18. The insertion of a node with Key 2 further covers the `else` branch at Lines 26-27, and finally the insertion of a node with Key 7 covers the `else` branch at Lines 19-20.

Therefore, the insertion of these four nodes in the specified order covers all the branches in the `insert` method in both versions of `BST`, and also results in the same tree for the two versions as shown in Figure 2. However, the behavior differs only if we insert a node with a key equal to the key of an existing node in the tree. A structural-coverage-based test generation tool might not generate such a test because the tool primarily targets at covering various branches and not specifically exposing behavioral differences. For example, the insertion of nodes with Keys 3 and 5 in the two versions of the `BST` class, respectively, results in a different tree as shown in Figure 3.

To detect behavioral differences, `DiffGen` first conducts a fast syntactic check to detect all the methods that have been changed over the two versions syntactically. `DiffGen` detects that the `insert` method has been changed syntactically in the new version. `DiffGen` then synthesizes a test driver

```
1    class BSTOld implements set{
2       Node node;
3       int size;
4       static class Node{
5          MyInput value;
6          Node left;
7          Node right;
8       }
9       public BSTOld() {.....}
10
11      public boolean insert(MyInput m){
12         Node t = root;
13         while(true){
14            if(t.compareTo(m.key)>0)//if(t.compareTo(m.key)>=0
15               if(t.right == null){
16                  t.right = new Node(m);
17                  break;
18               }
19               else
20                  t = t.right;
21            else
22               if(t.left == null){
23                  t.left = new Node(m);
24                  break;
25               }
26               else
27                  t = t.left;
28         .....
29         }
30      }
31      public void remove(MyInput m){.....}
32      public void contains(MyInput m){.....}

34      .....
35   }
```

Figure 1. The BSTOld class as in an old version. In a new version, Line 14 is changed to the one shown in the comment.

for a test generation tool such as jCUTE [33] and JUnit Factory [20]. In this section, we give an example of a synthesized driver for jCUTE [33]. The synthesized driver for the BST example is shown in Figure 4. jCUTE is a symbolic-execution [5, 15, 21] based test generation tool, which tries to generate tests to explore feasible paths in the program under test. In the test driver shown in Figure 4, we first invoke the constructors of both BSTOld and BST (the new version of BST) to create objects bstOld and bstNew, respectively. Next we invoke different methods for both the bstOld and bstNew objects inside the switch statement at Line 6 in Figure 4. Note that cute.Cute.input.Integer() at Line 6 generates various integer values trying to explore all feasible paths in the driver and eventually generates tests that contain all combinations of method sequences (whose lengths are up to five,

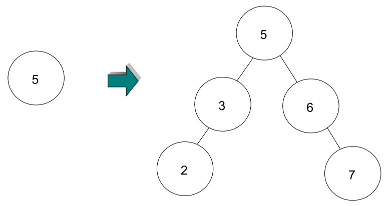

Figure 2. The BST object states before and after nodes with Keys 3, 6, 2, and 7 are inserted, respectively.

reflected by the `for` loop in Line 5). Note that inside all the case statements, we invoke the same method for both the `bstOld` and `bstNew` objects with the same method argument. For example, we invoke the `insert` method for both `bstOld` and `bstNew` inside the `case:0` branch at Line 7 of Figure 4. These symbolic method arguments help in the exploration of all feasible paths within the methods of the class under test. If a method has a non-primitive argument, we make a deep copy of the argument and invoke the methods in the two versions with different copies of the argument. We generate method arguments using `jCUTE`. `jCUTE` generates the arguments in such a way that these arguments help to cover as many branches as possible in the program under test.

After building the object state with various method call sequences, we invoke the methods that were modified between the two versions. As shown at Lines 24 and 25 of Figure 4, we invoke the `insert` method for both versions of the program under test. We then compare the return values obtained by invoking the insert method on the two objects `bstNew` and `bstOld`, as shown in Line 26. We finally compare the object states of the resulting `BSTNew` and `BSTOld` objects. As shown in Lines 28 and 30, we compare the object fields `size` and `Node`. Note that if `BSTOld` and `BSTNew` have no behavioral differences, the obtained return values, and the object states at this point would be the same, and if they are different, we have detected a behavioral difference. Therefore, if `jCUTE` is able to cover one of the branches at Lines 27, 29, and 31, we have detected a behavioral difference. In the instrumentation process, we also change the modifiers of all the fields transitively reachable by objects of the classes under test to `public`, so that we are able to compare these fields directly to detect behavioral differences. In the test generation phase, `DiffGen` uses `jCUTE` to generate a test suite for the driver (in Figure 4) synthesized in the previous phase. In the test suite generated by `jCUTE`, the failing tests cover at least one of the branches at Lines 27, 29, and 31 and hence expose the behavioral differences between the classes under test.

The generated test suite cannot be executed on the original source code, as it requires the fields transitively reachable by objects of the class under test to be public. Therefore, we annotate the original class with JML [23], a Java Modeling Language, in such a way that the generated tests can be executed on the original versions of the class. JML allows developers to specify preconditions and postconditions for each method inside a Java class. These preconditions and postconditions are inserted as comments inside the source code. These annotated Java classes, when compiled using a JML Compiler, result in Java bytecode with runtime checking code.

Given the class under test (`BST`) (the new version) and its old version class (`BSTOld`), `DiffGen` automatically synthesizes for `BSTOld` a wrapper class (`WrBST`) shown in Figure 5. This wrapper class

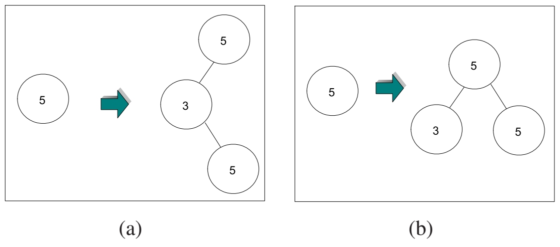

Figure 3. The `BST` object states before and after nodes with Keys 3 and 5 are inserted, respectively for **(a)** the old version of class `BST` and **(b)** the new version of class `BST`.

inherits `BSTOld`, and declares a set of wrapper methods for the public methods (called *reference methods*) declared in `BSTOld`. Each wrapper method (1) invokes the wrapped reference method in `BSTOld` with the same arguments being passed to the corresponding method under test in `BST`, (2) compares the receiver-object state (of the `WrBST`/`BSTOld` class) after the reference-method execution with the object state (of the `BST` class) after the method-under-test execution, and (3) compares the return values of the reference method and method under test if the methods have non-void returns.

`DiffGen` then automatically annotates `BST` with synthesized JML [23] preconditions and postconditions, as well as one extra `WrBST`-type field `_oldThis` and one extra method `_createShadowReferenceObj`. Figure 6 shows the annotated class. The `_createShadowReferenceObj` method constructs an object of the wrapper class `WrBST` and assigns it to `_oldThis`. `DiffGen` annotates the constructor with a JML precondition (marked with "`@`**`requires`**", as shown in Figure 6) that simply invokes `_createShadowReferenceObj` to create a reference-class object for comparisons during later method executions. `DiffGen` also annotates each public method with a JML postcondition (prefixed with "`@`**`ensures`**", as shown in Figure 6). The postcondition of a method `f` invokes on `_oldThis` the corresponding wrapper method with the following arguments: first, the current receiver object after the method execution (if `f` is not static), then, `f`'s arguments (if any, taken in the pre-state, denoted with "\**`old`**" in JML), and finally `f`'s return (if any, denoted with "\**`result`**" in JML). Note that the JML compiler (jmlc) translates the given annotations into Java byte code. For postconditions that involve "\**`old`**`(m)`", jmlc instruments extra byte code at the beginning of the method body to cache the value of the argument `m`, because `m` may be modified at the end of the method execution. The wrapper method requires the preceding arguments to accomplish its three tasks, described earlier. The synthesized wrapper class for the reference class and the synthesized JML annotations for the class under test form the core mechanism for coordinating the execution and result checking of corresponding methods from two versions of the same class.

To make the driver in Figure 4 compatible with the original source code we modify the driver. In particular, we modify the driver shown in Figure 4 by deleting the statements in which we compare the fields of `BST` and `BSTOld` directly (Lines 26-31 in the driver). We also delete the lines related to the method invocations on the `bstOld` object (i.e., Lines 3, 9, 13, 17, and 24). Figure 7 shows an example trace of a test generated for the driver in Figure 4. We provide the trace instead of an actual test to make

```
class BSTTestDriver implements set{
  public static void driver(){
    BSTOld bstOld = new BSTOld();
    BST bstNew = new BST();
    for(int i=0; i< 5; i++)
      switch(cute.Cute.input.Integer()){
        case 0:
        MyInput key = cute.cute.Input.Object(MyInput);
        bstOld.insert(key);
        bstNew.insert(key);
        case 1:
        key = cute.cute.Input.Object(MyInput);
        bstOld.remove(key);
        bstNew.remove(key);
        case 2:
        key = cute.cute.Input.Object(MyInput);
        bstOld.contains(key);
        bstNew.contains(key);
        .....
        default: break;
      }
    }
    MyInput key = cute.cute.Input.Object(MyInput);
    boolean b1 = bstOld.insert(key);
    boolean b2 = bstNew.insert(key);
    if(b1 != b2)
      Assert(false);
    if(bstOld.size != bstNew.size)
      Assert(false);
    if(bstOld.node.equals((bstNew.node))
//Checks if its a deep copy
      Assert(false);
  }
}
```

Figure 4. Test driver synthesized by `DiffGen` for the two versions of the `BST` class.

it more readable. As can be seen in Line 2 of Figure 7, a new object `bstNew` of the `BST` class is created. Lines 3 to 5 involve the invocation of a sequence of methods on `bstNew`. In this test case, the sequence length is three. In particular, an `insert(0)` is invoked, followed by a `remove(0)`, and finally an `insert(1)` is invoked. Line 6 shows the invocation of the textually different method `insert` with argument 1 on `bstNew`. `DiffGen` enables simultaneous execution of both the given versions of BST for the given test case. On executing the test case using DiffGen, a JML exception is thrown because the execution of Line 6 of Figure 7 results in different receiver object states for the objects of the given two versions of BST.

```
public class WrBST extends BSTOld {
  public WrBST() { super(); }

  public boolean equals(Object t) {
     if (!(t instanceof BST))
       return false;
     BST b = (BST)t;
     if (size != b.size) return false;
     if (!root.equals(b.root)) return false;
     return true;
  }

   public boolean insert(BST c, MyInput m, boolean r) {
     return (r == this.insert(m)) && this.equals(c);
   }

   public boolean remove(BST c, MyInput m) {
     this.remove(m);
     return this.equals(c);
   }

   public boolean contains(BST c, MyInput m, boolean r) {
     return (r == this.contains(m)) && this.equals(c);
   }
 }
```

Figure 5. Synthesized wrapper class for `BSTOld`

.

3. Approach

`DiffGen` takes as input two given versions of a Java class, and generates regression tests, which on execution expose behavioral differences between the two versions. Our `DiffGen` approach works in two major phases:

- **Test Generation.** In the *Test Generation* phase, we generate regression tests for the two given versions of a Java Class.
- **Test Execution.** In the *Test Execution* phase, we execute the regression tests that were generated in the *Test Generation* phase to expose behavioral differences.

We next describe the two phases in detail.

3.1. The Test Generation Phase

Figure 8 shows the various components involved in our Test Generation Phase. Given two versions of a class, first the *Change Detector* component detects all the syntactically different methods in the two versions of the class under test. The *Instrumenter* component then instruments the source code

```
class BST implements Set {
   ...
   transient WrBST _oldThis;
   boolean _createShadowReferenceObj() {
      _oldThis = new WrBST();
      return true;
   }

   /*@normal_behavior
     @requires _createShadowReferenceObj();
   @*/
   public BST() { ... }

   /*@normal_behavior
     @ensures _oldThis.insert(this, \old(m));
   @*/
   public boolean insert(MyInput m) { ... }

   /*@normal_behavior
     @ensures _oldThis.remove(this, \old(m));
   @*/
   public void remove(MyInput m) { ... }

   /*@normal_behavior
     @ensures _oldThis.contains(this, \old(m), \result);
   @*/
   public boolean contains(MyInput m) { ... }
}
```

Figure 6. `BST` annotated with synthesized JML preconditions and postconditions

.

```
            public boolean testBST(){
                BST bstNew = new BST();
                 bstNew.insert(0);
                 bstNew.remove(0);
                 bstNew.insert(1);
                 bstNew.insert(1);
             }
```

Figure 7. Trace of a test case generated for the test driver in Figure 4.

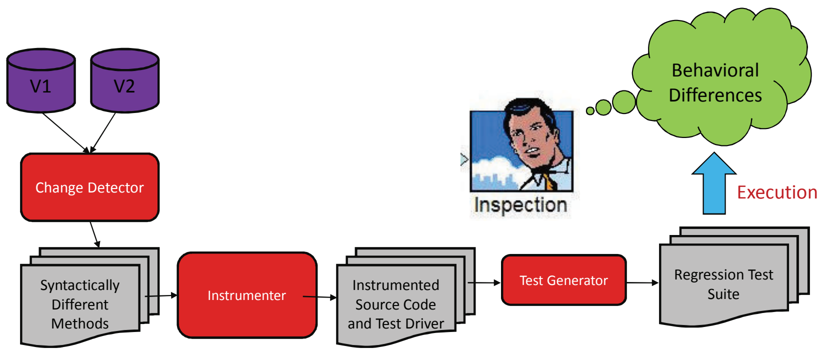

Figure 8. Overview of the *Test Generation* phase.

and generates a test driver. The *Test Generator* component then generates a regression test suite. The generated regression test suite is executed (in the *Test Execution* phase) to detect behavioral differences.

3.1.1. Change Detector

For all the corresponding method pairs in the two versions of the class under test, the *Change Detector* checks for textual similarity between the two versions. The *Change Detector* selects the corresponding method pairs by matching their names and signatures. If the two methods in a method pair are textually the same, they cannot be semantically different and thus are considered to have the same behaviors. The *Change Detector* filters out all the pairs with textually the same methods, and selects all the methods that are different textually. For the BST class (Figure 1), The *Change Detector* detects that the method `insert` has been modified.

3.1.2. Instrumenter

The *Instrumenter* component instruments the source code of the two versions of the class under test, and synthesizes a test driver for the *Test Generator* component to generate tests. The *Instrumenter* component first changes the modifier of all the fields transitively reachable from objects of the two given classes to *public*. This mechanism enables us to compare the object states after a sequence of method invocations on the objects of the two versions of the class under test directly by comparing the object fields. We synthesize two different types of drivers for two test generation tools (`jCUTE` and `Junit Factory`) to generate tests. The first kind of driver that we synthesize for `jCUTE` can in general be used for other symbolic-execution based tools, while the second kind of driver that we synthesize for `JUnit Factory` can be used for other test generation tools. We next describe the two kinds of drivers in detail.
`jCUTE` Test Driver. The `jCUTE` test driver synthesized by the *Instrumenter* component for the `BST`

example (Figure 1) is shown in Figure 4. In the driver, we first make new objects of both versions of the class under test by invoking one of the constructors. If a constructor requires one or more primitive arguments, we first generate symbolic primitive argument values by using `jCUTE` and pass the same argument values to invocations of both constructors. If the constructors require a non-primitive argument, we generate a symbolic argument object by using `jCUTE`, which generates a symbolic object state for the non-primitive argument. Once we have the objects created for the new and old versions of the class under test, we make a sequence of method invocations on the objects. For each method invocation, we pass the same arguments. If a method requires a non-primitive argument, we generate it using `jCUTE`, make a deep copy of the generated argument, and pass a different copy to invoke the methods in the two versions. If the non-primitive argument type contains static variables, we make a deep copy of the static variable as well, and replace the values of these static copies after the execution of the first method. We set the maximum length of the sequence of method invocations to be the number of methods in the original version of the class under test. If a change is found only in the constructor of the class under test, we do not invoke any sequence of method invocations.

Once we have made a sequence of method invocations, we invoke the methods that were found to be textually different by the *Change Detector* component. If any of these methods has a non-void return type, we compare the return values of the method. If the return values are different, we know that there is a behavioral difference between the two versions of the class under test. Therefore, we add a branch in the driver containing `assert(false)`(as shown in Line 27 of Figure 4), which is executed if the return values of these methods are different. If the only textually different method is the constructor, we do not invoke it again. Finally, we check the equivalence of resulting states of the two objects. If the same series of method invocation results in different receiver object states, we know that the behaviors of two methods are different. If the resulting receiver object states are the same, it means that the behaviors of the two versions are the same for this given input. For checking the equivalence of the resulting states, we compare the fields inside the class under test. We add branches in the driver containing `assert(false)`(as shown in Lines 29 and 31 of Figure 4), which are executed if any pair of the fields is not equivalent in the two versions. The execution of any of the branches at Lines 27, 29, and 31 of Figure 4 ensures that the behaviors of the two class versions are different. Therefore, the instrumentation converts the problem of finding behavioral differences between two versions of a class into a branch coverage problem. If we are able to generate tests that cover any of the branches at Lines 27, 29, and 31, we are able to find a behavioral difference between the two versions.

`JUnit Factory` Test Driver. The `JUnit Factory` test driver synthesized by the *Instrumenter* component for the `BST` example (Figure 1) is shown in Figure 9. The driver class contains one method for every changed method inside the class under test. These methods are a kind of parameterized unit tests [32]. Each method has an object of the original version of class under test as an argument. The rest of the arguments of the method are the same as the arguments of the changed method in the class under test. For example, in Figure 9, the method `compareInsert` compares the behaviors of the two versions of the method `insert`. An argument of `compareInsert` is an object (`bstOld`) of `BSTOld`. Inside the body of `compareInsert`, we make a new object of `BST` (Line 3) and deep copy the field of `bstOld` to the fields of `bstNew` (Line 4). We then invoke the changed method from the objects of the two versions of class under test (Lines 5 and 6). We next compare the return values and the resulting object states of `bstNew` and `bstOld` with new branches (Lines 8, 10 and 12) .

```
public class BSTJUFDriver{
   public void compareInsert(BSTOld oldBST,
        MyInput input){
      BST bstNew = new BST();
      bstNew = copyObject(oldBST);
      boolean b1 = bstOld.insert(input);
      boolean b2 = bstNew.insert(input);
      if(b1 != b2)
         Assert(false);
      if(bstOld.size != bstNew.size)
          Assert(false);
       if(!bstOld.root.equals(bstNew.root))
          Assert(false);
    }
 }
```

Figure 9. Test Driver synthesized for JUnit Factory.

3.1.3. Test Generator

The Test Generator component uses either of `jCUTE` [33] or `JUnit Factory` for test generation. `jCUTE` is a concolic-execution-based test generator tool. Concolic testing uses symbolic execution to generate inputs that guide the execution of a program to explore alternate paths. In particular, symbolic execution collects the constraints from all the branching points involved in a path, changes these constraints to new ones that will help explore a different path, and solves the new constraints to generate an input for the different path. Whenever a part of the constraints is too complex to be handled by a constraint solver, concrete values are used to replace the symbolic values in the part of the constraints. `JUnit Factory` is a commercial automated characterization test generator based on Agitar [3].

The input to the *Test Generator* component is the instrumented code and the test driver generated by the Instrumenter component. `jCUTE` (or `JUnit Factory`) uses the test driver to generate regression tests. `jCUTE` tries to generate test inputs to cover all the branches in the test driver and the two versions of the class under test. In particular, it solves path constraints to explore feasible paths in the classes under test to generate concrete test inputs. Hence it tries to solve path constraints to cover the branches shown in Lines 27, 29, and 31 of Figure 4. All the tests that cover one of the branches fail because of the invocation of `assert(false)`. Hence, all the generated tests that fail on execution expose behavioral differences. Similarly, `JUnit Factory` also tries to achieve maximum structural coverage and thus tries to generate inputs so that branches at Lines 8 and 10 of Figure 6 can be covered.

3.2. Test Execution Phase

Making public the fields inside both versions of the class under test is helpful for comparing object states directly, and hence it is helpful in regression test generation. However, the generated regression tests require the fields in the class under test to be public, and hence the tests cannot be executed on the original source code. To be able to execute the regression tests on the original source code, we developed a framework that annotates the source code with JML [23] annotations such that we would be able to execute the generated tests on the original source code (i.e., not requiring the fields inside the class under test to be public). In particular, whenever a method in the new version of the given class is invoked from the generated test suite, our framework enables us to invoke a corresponding method in the old version of the class under test. After the two methods are invoked, `DiffGen` checks the return values of the two invoked methods along with the two resulting receiver object states to find behavioral differences.

We first describe how we represent states of non-primitive-type objects. Based on the state representation, we define the input and output of a method execution. We then present techniques for providing the same inputs to pairs of corresponding methods from the two class versions and for comparing the outputs of the pairs of methods. Finally, we describe how we execute the test suite generated in the Test Generation phase to detect behavioral differences between two versions of the class under test.

3.2.1. State Representation

When a variable (such as the return or receiver of a method invocation) is a non-primitive-type object, we use concrete-state representation from our previous work [41] to represent the variable's value or state. A program executes on the program state that includes a program heap. The concrete-state representation of an object considers only parts of the heap that are reachable from the object. We also call each part a *heap* and view it as a graph: nodes represent objects, and edges represent fields. Let P be the set consisting of all primitive values, including `null`, integers, etc. Let O be a set of objects whose fields form a set F. (Each object has a field that represents its class; array elements are considered index-labeled fields of the array objects.)

Definition 1. *A* heap *is an edge-labelled graph* $\langle O, E\rangle$*, where* $E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}$.

Heap isomorphism is defined as graph isomorphism based on node bijection [4].

Definition 2. *Two heaps* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there is a bijection* $\rho : O_1 \rightarrow O_2$ *such that:*

$$\begin{aligned} E_2 \quad = \quad & \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \\ & \{\langle \rho(o), f, o'\rangle | \langle o, f, o'\rangle \in E_1, o' \in P\}. \end{aligned}$$

The definition allows only object identities to vary: two isomorphic heaps have the same fields for all objects and the same values for all primitive fields.

The state of an object is represented with a *rooted* heap:

Definition 3. *A* rooted heap *is a pair* $\langle r, h\rangle$ *of a root object* r *and a heap* h *whose all nodes are reachable from* r.

Another way of representing an object state is to use the method-sequence-representation technique [17, 41]. The technique uses sequences of method invocations that produce the object. The state representation uses symbolic expressions with the grammar shown below:

$$\begin{aligned} \text{exp} &::= \text{prim} \mid \text{invoc } \text{“}\texttt{.state}\text{”} \mid \text{invoc } \text{“}\texttt{.retval}\text{”} \\ \text{args} &::= \epsilon \mid \text{exp} \mid \text{args } \text{“}\texttt{,}\text{” exp} \\ \text{invoc} &::= \text{method “}\texttt{(}\text{” args “}\texttt{)}\text{”} \\ \text{prim} &::= \text{“}\texttt{null}\text{”} \mid \text{“}\texttt{true}\text{”} \mid \text{“}\texttt{false}\text{”} \mid \text{“}\texttt{0}\text{”} \mid \text{“}\texttt{1}\text{”} \mid \text{“}\texttt{-1}\text{”} \mid \ldots \end{aligned}$$

Each object or value is represented with an expression. Arguments for a method invocation are represented as sequences of zero or more expressions (separated by commas); the receiver of a non-static, non-constructor method invocation is treated as the first method argument. A static method invocation or constructor invocation does not have a receiver. The `.state` and `.retval` expressions denote the state of the receiver after the invocation and the return of the invocation, respectively.

3.2.2. *Method Execution*

The execution of an object-oriented program produces a sequence of method executions.

Definition 4. *A* method execution *is a six-tuple $e = (m, S_{args}, S_{entry}, S_{exit}, S_{args'}, r)$, where m is the method name (including the signature), S_{args} are the argument-object states at the method entry, S_{entry} is the receiver-object state at the method entry, S_{exit} is the receiver-object state at the method exit, $S_{args'}$ are the argument-object states at the method exit, and r is the method return value.*

Definition 5. *The* input of a method execution *$(m, S_{args}, S_{entry}, S_{exit}, S_{args'}, r)$ is a pair (S_{args}, S_{entry}) of the argument-object states and the receiver-object state at the method entry.*

Definition 6. *The* output of a method execution *$(m, S_{args}, S_{entry}, S_{exit}, S_{args'}, r)$ is a triple $(S_{exit}, S_{args'}, r)$ of the receiver-object state at the method exit, the argument-object states at the method exit, and the method return value. (Our framework and its implementation consider only the receiver-object state at the method exit and method return value; argument-object states at the method exit are rarely updated and thus often ignored for checking the output.)*

Note that when m's return is $void$, r is $void$; when m is a static method, S_{entry} and S_{exit} are empty; when m is a constructor method, S_{entry} is empty.

3.2.3. *Method-Execution Comparison*

`DiffGen` treats one of the two class versions (the new version after modification) as the class under test and the other version, called a *reference class* (the original version), as the class to be checked against during program execution. `DiffGen` provides two key steps in supporting simultaneous execution and checking of the two class versions: pre-method-execution setup and post-method-execution checking.

3.2.4. *Pre-Method-Execution Setup*

The step of pre-method-execution setup occurs before the execution of a method in the class under test (called the *method under test*). In this step, `DiffGen` collects and prepares the same input (defined in

Section 3.2.2) to be executed on the corresponding method in the reference class (called the *reference method*). Note that the execution of the reference method can occur in this step (before the execution of the method under test), concurrently with the execution of the method under test, or after the execution of the method under test (in the step of post-method-execution checking).

Using the method-sequence representation (presented in Section 3.2.1), `DiffGen` does not need to reconstruct a receiver-object state before every execution of a reference method; instead, it can reuse the receiver-object state constructed after the execution of the previous reference method. In such an implementation, a shadow object of the reference class may be needed for each object of the class under test, and the framework needs to ensure that the same method sequences are invoked on these two objects during their life time. The example in Section 2 shows one framework implementation based on the method-sequence representation.

The behaviors of the method under test and the reference method are sometimes intentionally made different for a subdomain of inputs. For example, the reference method that can handle only positive integers is extended to handle additionally negative integers. When inputs fall into the new subdomain (e.g., negative integers in the preceding example), the differences in the outputs of the two method executions should be ignored. If developers would like to exclude the warnings of these differences automatically, they need to construct a predicate method for determining whether an input falls into this subdomain.

3.2.5. Post-Method-Execution Checking

The step of post-method-execution checking occurs after the executions of a method in the class under test and its corresponding reference method. As discussed earlier in the pre-method-execution setup, these two method executions can be concurrent or sequential (in any order). This step compares the outputs (defined in Section 3.2.2) of the two executions.

The formats of the outputs (especially the receiver-object state) can be different for the method under test and the reference method. As discussed earlier, developers would need to provide conversion methods that convert the output of the method under test to the format of the reference method. For checking output only (e.g., when the method-sequence representation is used to represent the receiver-object states in the inputs of method executions), developers may define an abstraction function [24, 41] such as an `equals` method for mapping the receiver-object states that are deemed to be equivalent into the same abstract values.

3.2.6. Test Execution

Figure 10 shows the overview of our `Test Execution` phase. The `Test Execution` phase takes as inputs the class under test, `C`, and its reference class, `R` (an old version of `C`). The `Test Execution` phase also takes as input the regression test suite generated for the two class versions `C` and `R`. `DiffGen` generates a wrapper class that extends `R`. Our current implementation of `DiffGen` uses the annotations of Java Modeling Language (JML) [23] to implement the steps of pre-method-execution setup and post-method-execution checking. `DiffGen` annotates each constructor or public method of `C` with JML preconditions and postconditions. The preconditions for constructors create shadow reference objects for later method execution and behavior comparison, while postconditions for public methods invoke the corresponding methods in the wrapper class to compare the behaviors of the corresponding

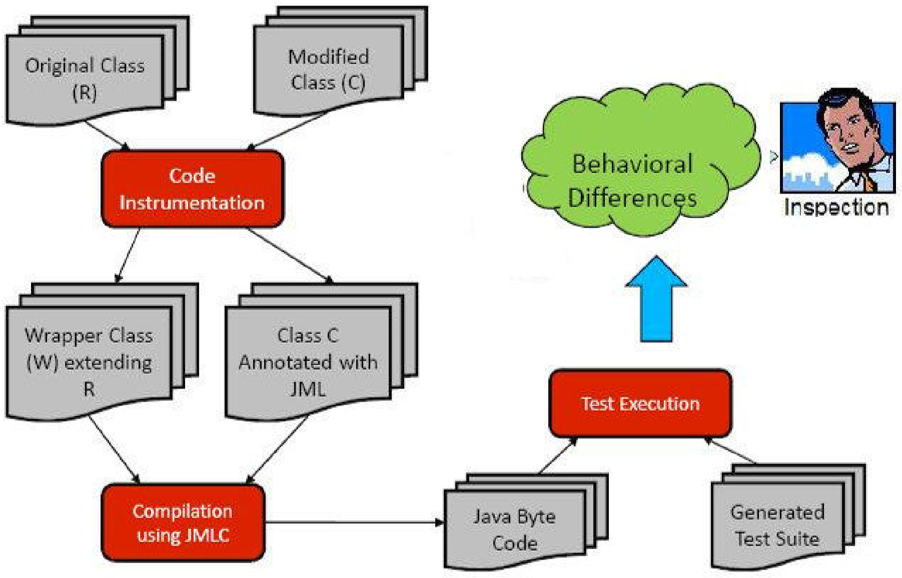

Figure 10. Overview of the Test Execution phase.

methods in `C` and `R`. Developers can customize the wrapper class or the JML annotations to adjust for the intended changes of inputs and outputs between `C` and `R`.

`DiffGen` compiles the wrapper class and class `C` (annotated with JML annotations) using the JML compiler that generates Java bytecode with runtime checking code. When there is any behavioral difference between classes `C` and `R`, the runtime checking code throws an exception signifying JML postcondition violations.The generated bytecode is executed with the regression test suite generated in the *Test Generation* phase.

The wrapper class (`W`) inherits from the reference class (`R`) and declares one constructor for each constructor in the Class `R`. Each constructor takes the same arguments as the corresponding constructor in the class `R`. Each constructor invokes the corresponding parent constructor (`super`). The wrapper class also defines an `equals` method for comparing the object states of the class under test and the reference class after the method execution. If the class under test has defined an `equals` method, the `equals` method defined in the wrapper class is simply copied from the one defined in the class under test. Developers can also customize the content within this `equals` method for constructing appropriate mappings between the object fields of the class under test and the reference class. The wrapper class for the BST class in Figure 1 (reference class) is shown in Figure 5.

The wrapper class (`W`) includes a wrapper method for each public method defined in the reference class (`R`) as well as the class under test (`C`). The wrapper method's signature includes extra parameters beyond the ones defined in the original method in the reference class. First, if the original method is not a static method, a parameter with the type of the class under test is inserted as the first parameter in the wrapper method's signature. During program execution, this first method argument is the receiver object of the class under test after the execution of the corresponding method under test (details are described in Section 3.4). All the three public methods of `ReferenceBST` are not static; therefore, an extra parameter of the `BST` type is inserted to each of these methods as the first parameter. Second, if the return of a public method of the reference class is not `void`, an extra parameter of the return type is appended as the last parameter of the wrapper method. The `contains` method is such an example in Figure 5.

3.3. Precondition Synthesis

Before we present the details of synthesized preconditions for the method under test (in the class under test), we describe one extra helper field that is inserted to the class under test. Figure 6 shows the class under test (`BST`) for the BST example in Figure 1 annotated with the synthesized JML preconditions and postconditions, as well as one extra `WrBST`-type field `_oldThis` and one extra method `_createShadowReferenceObj`. Figure 6 shows the annotated class. The `_createShadowReferenceObj` method constructs an object of the wrapper class `WrBST` and assigns it to `_oldThis`. `DiffGen` annotates the constructor with a JML precondition (marked with "`@`**`requires`**", as shown in Figure 6) that simply invokes `_createShadowReferenceObj` to create a reference-class object for comparisons during later method executions. The synthesized preconditions implement the step of the pre-method-execution setup (Section 3.2.4). Note that in preconditions we have not performed caching or collection of the method argument `m`, because this operation has been accomplished with the use of `\old(m)` in the synthesized postconditions (Section 3.4).

3.4. Postcondition Synthesis

In the synthesized postconditions for the method under test `f` (prefixed with "`@`**`ensures`**", as shown in Figure 6), the corresponding wrapper method is invoked with the first argument being the current receiver object (if `f` is not a static method) followed by `f`'s arguments (if any, being enclosed with "`\old()`"), and the last argument being the `f`'s return (if any, denoted with "`\result`" in JML). Note that the JML compiler (jmlc) translates the given postconditions, such as "`\old(m)`", by instrumenting the extra code at the beginning of the method body for caching the values of the argument `m`, because `m` may be already modified at the end of the method execution.

Inside the method body of each invoked wrapper method, the original method in the reference class is invoked on `this` with the method arguments that are used to invoke the corresponding method under test in the class under test. If the original method's return is not `void`, the return value of the original method is compared with the last argument of the wrapper method, being the return value of the method under test. In summary, the synthesized postconditions implement the step of the post-method-execution checking (Section 3.2.5).

4. Experiments

This section presents our experiments conducted to address the following research question:

- Can the regression test suite generated by `DiffGen` effectively detect regression faults that cannot be detected by previous state-of-the-art techniques [12]?

If the answer is yes, then DiffGen can be used to complement state-of-the-art techniques to improve regression-fault detection capability.

4.1. Experimental Subjects

Table I lists eight Java classes that we use in first set of experiments. `UBStack` is the illustrative example taken from the experimental subjects used by Stotts et al. [34]. `IntStack` was used by Henkel and Diwan [17] in illustrating their approach of discovering algebraic specifications. `ShoppingCart` is an example for JUnit [6]. `BankAccount` is an example distributed with Jtest [27]. The remaining four classes are data structures previously used to evaluate Korat [4]. The first four columns show the class name, the number of methods, the number of public methods, and the number of non-comment, non-blank lines of code for each subject, respectively. The last column shows the branch coverage achieved by tests generated by `JUnit Factory` for the original version.

4.2. Experimental Setup

Although our ultimate research question is to investigate if `DiffGen` can detect regression faults not detected by previous state-of-the-art techniques, our subject classes were not equipped with such faults; therefore, we used MuJava [25], a Java mutation testing tool, to seed faults in these classes. MuJava modifies a single line of code in an original version in order to produce a faulty version. For each mutant and the original class version, we generate tests using `JUnit Factory` [20].

To evaluate the fault-detection capability of our `DiffGen` approach, we compare the the fault-detection capability of the test suite generated by `DiffGen` with the fault-detection capability of test suites generated by `JUnit Factory`, separately for two versions of the class under test.

Experiments using `jCUTE`. Figure 11 shows the test driver that we use for generating tests separately for the original class and the mutant version for the `BST` example in Section 2. As shown in the figure, we make an object of only one version of `BST`, we later invoke the sequence of methods on this object in a similar way as in the driver synthesized by `DiffGen` (shown in Figure 4). The test driver for other subjects also has the same structure as the driver for `BST`. We use the same driver to generate tests for both the original and mutant version separately. Once the tests are generated, we use our *Test Execution* phase (described in Section 3.2) to execute the generated tests for each pair of the original and mutant versions of the class. This phase enables the simultaneous execution of both classes for each test in the generated test suite, and compares their observable outputs and resulting receiver object state to expose behavioral differences. This approach for finding behavioral differences is similar to the one proposed by Evans and Savoia [12] (discussed in Sections 1 and 7). For simplicity, we refer to the preceding approach as `SeparateGen` in the rest of this paper.

The second to the last column of Table I shows the branch coverage achieved by tests generated by `jCUTE` on average for the original as well as a mutant version of the Java class under test, while the last column shows the branch coverage achieved by tests generated by `JUnit Factory`. To generate tests for the original and a mutant versions used in `SeparateGen`, we set the upper limit on the number of paths to be explored by `jCUTE` to 2000.

```
class BSTTestDriver implements set{
   public static void driver(){
      BST bst = new BST();
      for(int i=0; i< 5; i++)
         switch(cute.Cute.input.Integer()){
            case 0:
         MyInput key = cute.cute.Input.Object(MyInput);
         bst.insert(key);
     case 1:
         key = cute.cute.Input.Object(MyInput);
         bst.remove(key);
      case 2:
         key = cute.cute.Input.Object(MyInput);
         bst.contains(key);
      .....
      default: break;
         }
      }
   }
}
```

Figure 11. Test driver used to generate tests for the original and a mutated version of the BST class, separately, using the SeparateGen approach.

We next measure the number of killed and unkilled mutants by SeparateGen. For all the mutants that were not killed by SeparateGen, we use DiffGen to generate regression tests, and record the number of killed mutants among the unkilled mutants by SeparateGen. We consider a mutant killed if a JML-postcondition-violating exception is thrown while executing these tests. Note that there are some mutants whose behaviors are the same as the behaviors of the original version. On manual inspection, we found out that in many mutants, the value of a local parameter is increased (or decreased) using a postfix increment (or decrement) operator, and the variable is never used after the mutated statement. We also count such mutants among the mutants not killed by DiffGen.

Our DiffGen approach is more expensive than the SeparateGen approach for achieving a similar coverage of classes under test. To be fair about the time given to both approaches to find behavioral differences, we set the maximum number of paths to be explored by jCUTE (while generating regression tests using DiffGen) to be 200 (which is much less as compared to the number of paths explored by jCUTE while generating tests using SeparateGen). We also observed that the time taken to generate tests using our DiffGen approach (time for exploring 200 paths) was substantially less than the time taken to generate tests using SeparateGen (time for exploring 2000 paths). This observation shows that our experimental setup at least does not favor our DiffGen approach in terms of resource allocation.

Experiments using JUnit Factory. We generate tests separately for the two versions of class under test with JUnit Factory. To generate the tests using JUnit Factory, we do not need a test driver. We directly feed the two versions separately to JUnit Factory and generate tests. We run the generated tests for the original version on a mutant version, and run the generated tests for the mutant version on the original version of the class under test to expose behavioral differences. This approach is the one used by Evans and Savoia [12]. We then use `DiffGen` to find behavioral differences between the original version of class under test and the mutants that were left unkilled by `SeparateGen`, in a similar way as what we do with `jCUTE` in our experimental setup.

4.3. Metrics

We measure the total number of mutants generated for each subject, the number of mutants not killed by `SeparateGen` (denoted by `u`), the number of mutants killed by `DiffGen` among the ones not killed by `SeparateGen` (denoted by `k`). We also measure the number of mutants that had the same behavior as the original version (denoted by `s`) among the mutants that were not killed by `DiffGen` or `SeparateGen`. We next measure Improvement Factor (`IF1`) of `DiffGen` over `SeparateGen`. `IF1` = $\frac{k}{u}$. We measure another improvement factor (`IF2`) of `DiffGen` over `SeparateGen` by excluding the mutants with the same behavior as the original version of class (denoted by s). `IF2` = $\frac{k}{u-s}$. The values of IF1 and IF2 indicate the extra fault-detection capability of `DiffGen` over `SeparateGen`.

4.4. Results

This section summarizes the results that we obtained for the experiments we conducted.
Results for experiments using `jCUTE`. Table II shows the results from the experiment that we conducted. Column 1 shows the name of the subject. Column 2 shows the total number of mutants. Columns 3 shows the number of mutants left unkilled by `SeparateGen` using `jCUTE` for test generation. Columns 4 shows the number of mutants killed by `DiffGen` using `jCUTE` for test generation, among the mutants that were not killed by `SeparateGen` (using `jCUTE` for test generation). Column 5 shows the number of mutants with the same behavior as the original class version. Columns 6 and 7 show the Improvement Factors `IF1` and `IF2`, respectively of `DiffGen` (using `jCUTE` for test generation). From Table II, we observe that `DiffGen` (using `jCUTE` for test generation) has an Improvement Factor `IF2` varying from 40% to 83% for all the subjects with the exception of `IntSet` that has an improvement factor of 21.7%.

On manual inspection of the unkilled mutants generated for `IntStack`, we found out that most of the mutants not killed by `DiffGen` had faults in branches that were not covered by the generated tests. In particular, these branches were the ones for handling the behavior of `IntStack` when the stack size exceeded the `Initial Capacity` of `IntStack`. `jCUTE` could not generate tests that inserted as many elements in the stack such that the stack size became equal to the `Initial Capacity` of `IntStack`. Therefore, these branches could not be covered either by the tests generated for `SeparateGen` or by the tests generated by `DiffGen`. Hence these mutants were not killed by either approach.
Results for experiments using `JUnit Factory`. Table II shows the results obtained from our experiment using `JUnit Factory`. Columns 8 shows the number of mutants left unkilled by `SeparateGen` using `JUnit Factory` for test generation. Columns 9 shows the number of mutants killed by `DiffGen` using `JUnit Factory` for test generation, among the mutants that were not killed by `SeparateGen` (using `JUnit Factory` for test generation). Columns 10 and 11 show the Improvement Factors `IF1` and `IF2`, respectively of `DiffGen` (using `JUnit Factory` for test generation). From Table II, we observe that the `SeparateGen` approach kills more mutants using

```
1   public boolean allDifferent(){
2      int n = size - 1;
3         for (int i = 0; i < n; i++){
4            for (int j = i + 1; j < n; j++) { //for (int j = i + 1; j < n--; j++){
5               if (elements[i] == elements[j])
6                  return false;
7            }
8         }
9      return true;
10  }
```

Figure 12. A method from `DisjSet` class, where Line 4 was mutated to the one shown in the comment.

`JUnit Factory` for test generation than the `SeparateGen` approach using `JUnit Factory` for test generation. This result is expected because of the high coverage achieved by the test suites generated by `JUnit Factory`. In total, `DiffGen` (using `JUnit Factory` for test generation) has a high Improvement Factor `IF2` of above 33%.

On manual inspection of unkilled mutants for `DisjSet` (which has a relatively low Improvement Factor `IF2` of 23%) , we found that many of the unkilled mutants contained a fault injected in a loop condition. Consider a representative of such types of mutants in Figure 12. Line 4 of the method `allDifferent` (in class `DisjSet`) was mutated to the one shown in the comment at the same line. Expression `n` was mutated to `n--` in the inner loop.

In summary, the evaluation answered the question that we mentioned in the beginning of Section 4. `DiffGen` can detect a substantial percentage of faults that were not detected by `SeparateGen`, which is the representative of previous state-of-the-art techniques in test generation. In particular, `DiffGen`, using `jCUTE` for test generation, is able to detect 51.31% of faults that `SeparateGen` could not detect and `DiffGen`, using `JUnit Factory` for test generation, is able to detect 33.06% of faults that `SeparateGen` could not detect.

4.5. Experiments on Larger Subject Programs

We conducted additional experiments on larger subject programs to validate that our approach is useful for these subjects. These subjects and their faults are taken from the Subject Infrastructure Repository (SIR) [10]. We conducted experiments on three available versions of the JTopas [19] subject from SIR. Among the three versions, we chose the classes that had faults available at SIR. There were 13 such classes and 38 faults in total available for these classes. We tested on versions of these classes prepared by seeding all the available faults in the SIR repository for these classes one by one. These 13 classes were the same subjects used by Evans and Savoia [12].

We did not use `jCUTE` in our experiments for these subjects because `jCUTE` cannot deal with strings, and many of the methods in these classes had strings as arguments. In addition, many of the fields in these classes were of interface types; hence, `jCUTE` was unable to instantiate objects for these classes. We used `JUnit Factory` to generate tests for the original version and each faulty version of the

Table I. Experimental subjects

Class	#Methods	#Public	#LOC(ncnb)	`jCUTE` Branch Cov	`JUF` Branch Cov
IntStack	5	5	44	78%	100%
UBStack	11	11	106	79%	100%
ShoppingCart	9	8	70	100%	100%
BankAccount	7	7	34	100%	100%
BinSearchTree	13	8	246	84%	100%
BinomialHeap	22	17	535	85%	87%
DisjSet	10	7	166	100%	100%
FibonacciHeap	24	14	468	89%	98%

Table II. Experimental Results

Class	#M	#JC Unk Mutants	DG Killed	#Same Behavior	`IF1` (%)	`IF2` (%)	#JUF Unk	DG Killed	`IF1` (%)	`IF2` (%)
IntStack	85	44	5	21	11.4	21.7	21	0	0	-
UBStack	187	18	6	7	30	54.5	15	6	40	75
ShoppingCart	18	10	5	4	50	83.3	7	3	42.85	100%
BankAccount	35	17	8	6	47	72.7	6	0	0	-
BinSearchTree	125	18	11	4	61.1	78.6	13	4	30.77	44.44
BinomialHeap	281	63	25	19	39.7	56.8	39	8	20.51	40
DisjSet	385	87	22	33	25.28	40.7	97	15	15.46	23.44
FibonacciHeap	339	137	45	43	32.8	47.87	53	5	9.43	50
Total	1455	394	127	137	32.23	51.31	251	41	16.33	33.06

subjects. We used the `SeparateGen` approach used by Evans and Savoia [12] to check whether the faulty versions were detected using the approach. We used our approach to detect the seeded faults that were not detected by the `SeparateGen` approach. Our subject classes are shown in Table III. Column 1 of the table shows the version of JTopas. Column 2 shows the name of the class. Column 3 shows the lines of code in the class. Column 4 shows the number of faults available in the repository for that class. Column 5 shows the faulty versions that were not detected by the approach `SeparateGen`. Finally the last column shows the number of faulty versions detected by our `DiffGen` approach among the ones not detected by the `SeparateGen` approach.

The results show that our `DiffGen` approach detects 5 of the 7 faults that were not detected by the `SeparateGen` approach.

5. Discussion

In this section, we discuss some of the limitations of the current implementation of our approach and how they can be addressed.

Changes on methods or signatures. The current implementation of `DiffGen` cannot deal with refactorings or other maintenance activities that change the name or signature of a method (such as Rename-Method and Changed-Method-Signature Refactorings [13]). The *Change Detector* component in the *Test Generation* phase detects corresponding methods in the two versions by

Table III. Experimental Results

Version	Class	LOC	#Faults	#Undetected	#Detected
v1	ExtIOException	78	3	0	-
v1	AbstractTokenizer	1672	3	1	1
v1	Token	159	1	0	-
v1	Tokenizer	287	1	0	-
v1	ExtIndexOutOfBoundsException	67	2	0	-
v2	ExtIOException	89	2	0	-
v2	ThrowableMessageFormatter	137	2	0	-
v2	AbstractTokenizer	2966	4	2	2
v2	Token	447	4	0	-
v3	EnvironmentProvider	240	3	1	0
v3	PluginTokenizer	407	1	0	-
v3	StandardTokenizer	1992	8	2	2
v3	StandardTokenizerProperties	2736	4	1	0
Total	13 classes		38	7	5

comparing the method names and signatures. In particular, `DiffGen` considers two methods to be corresponding if their names and signatures match. A refactoring detection tool [9] can be used to find corresponding methods that were refactored in the new version of the given class. `DiffGen` can then detect the behavioral differences for the methods whose names were changed. However, for methods or constructors whose signatures were changed (methods with a modified, added, or deleted parameter), developers would need to write a conversion method to convert the input format of the methods or constructors in the original version to the inputs required by the new version.

Changes on fields. The current implementation of `DiffGen` compares objects by directly comparing the fields in the objects of the class under test. However, if a field is deleted, added, or modified in the new version of the class under test, `DiffGen` cannot correctly compare the receiver object states. This situation can be addressed by invoking various observer methods on objects under comparison and comparing the return values of these observer methods.

Multiple changes. Although in our experiments we detect behavioral changes between an original and a mutant version of a class with a single fault injected. Our current implementation can deal with multiple changes in the same method, since we work at the granularity of a method. To detect multiple changes spread in multiple methods, the synthesized test driver can be modified to invoke all the textually different methods in a switch statement (instead of invoking just one method) after the invocation of a sequence of methods . The return values of the invoked methods and the resulting receiver object states can then be compared to detect behavioral differences. Figure 13 shows the test driver synthesized for the BST class when multiple changes are made to methods `insert` and `delete`. In the test driver, we first invoke the constructors of both `BSTOld` and `BST` (the new version of `BST`) to create objects `bstOld` and `bstNew`, respectively. Between Lines 4 and 5 we invoke a sequence of methods on the objects `bstOld` and `bstNew` as in the driver shown in Figure 4. We then invoke the changed methods in the switch statement at Line 6. The return values of the invoked methods and the resulting receiver object states can then be compared to detect behavioral differences (Lines 16-21).

Guided exploration. The runtime performance of the current implementation of `DiffGen` can be improved by conducting a guided exploration of paths. Currently, `DiffGen` uses `jCUTE` [33] to explore

```
class BSTTestDriver implements set{
   public static void driver(){
      BSTOld bstOld = new BSTOld();
      BST bstNew = new BST();
      /*Sequence of Method Invocations*/
      ....
      ....
      boolean b1, b2;
      switch(cute.Cute.input.Integer()){
         case 0:
            MyInput key = cute.cute.Input.Object(MyInput);
            boolean b1 = bstOld.insert(key);
             boolean b2 = bstNew.insert(key);
         case 1:
            MyInput key = cute.cute.Input.Object(MyInput);
            boolean b1 = bstOld.delete(key);
            boolean b2 = bstNew.delete(key);
      }
      if(b1 != b2)
         Assert(false);
      if(bstOld.size != bstNew.size)
         Assert(false);
      if(bstOld.node.equals((bstNew.node))//Checks if its a deep copy
         Assert(false);
}
```

Figure 13. Test driver synthesized by `DiffGen` for the two versions of the `BST` class. The methods `insert` and `delete` are changed between the old and the new versions of the `BST` class.

various paths in order to eventually cover a path leading to behavioral differences between the two given versions. However, many paths need not be explored, if these paths do not help in exposing behavioral differences between the two given versions (e.g., these paths do not involve any changed code location). The runtime performance of `DiffGen` can be improved by guiding the path exploration to avoid these paths. A static analysis can be done to find those branches or paths that do not lead to the changed locations. We can then prevent `jCUTE` from exploring these branches or paths by making `jCUTE` backtrack for these branches or paths.

Testing multiple classes. The current implementation of `DiffGen` tests two versions of one class at a time. There can be some behavioral differences missed by our approach. Consider that a method in some other class is invoked from the original and modified version of the class under test. There is a change in the invoked method, but there is no change in the two versions of the class under test. Our change detector would find no textual difference between the two versions of the class under test. Hence, no behavioral differences are found. Such situations can be addressed by either testing all the classes or extending the change detector to find textual differences between the transitive closure of all the methods invoked from the class under test.

Other limitations. Since we use `jCUTE` [33] or `JUnit Factory` for test generation, the limitations in `jCUTE` or `JUnit Factory` are inherited by `DiffGen`. For example, `jCUTE` does not provide effective support for generating method sequences that produce desirable receiver-object states or non-primitive-argument states. Currently, in our test driver, we generate method sequences of lengths up to the number of public methods inside the class under test. However, many faults may be detected by sequences of only a larger length. Due to this limitation in `jCUTE`, we could not detect many behavioral differences in mutants of `IntStack`. To deal with this situation, we can use testing tools such as Evacon [18] to address this limitation.

6. Threats to Validity

The threats to external validity primarily include the degree to which the subject programs are representative of true practice. Our subject JTopas from the SIR is a medium-scale real world open-source software system. The subject has also been used for experiments by evaluating various approaches [12, 38]. Our data-structure subjects are from various sources and the Korat data structures have nontrivial size for unit testing. We currently use third-party tools `jCUTE` [33], a state-of-the-art test generation tool that is based on concolic execution, and `JUnit Factory` [20], a commercial test generation tool. These threats could be further reduced by experiments on more subjects and third-party tools. The main threats to internal validity include faults in our tool implementation and faults in the third-party tools that we used to generate tests. To reduce these threats, we have manually inspected the execution traces for several program subjects. The main threats to construct validity include the uses of those metrics in our experiments to assess our approach. To assess the effectiveness of our tool, we measure the percentage of faults detected by our tool but not detected by the `SeperateGen` approach. These faults were seeded by a mutation testing tool called `MuJava` [25] to approximate the real regression faults introduced as an effect of changes made in the maintenance process. Although empirical studies showed that faults seeded by mutation testing tools yield trustworthy results [1], these threats can be reduced by conducting more experiments on real regression faults together with fault-free changes or by conducting experiments using more mutation testing tools.

7. Related Work

Our previous Orstra approach [40] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given test suite and collects the return values and receiver-object states after the execution of the methods under test. Based on the collected information, Orstra synthesizes and inserts new assertions in the test suite for asserting against the collected method-return values and receiver object states. However, this approach observes the behavior of the original version to insert assertions in the test suite generated for only the original version. Therefore, the test suite might not include test inputs for which the behavior of a modified version differs from the original version.

Evans and Savoia [12] recently proposed an automated differential testing approach in which they generate test suites for the two given versions of a software system (say *V1* and *V2*) using `JUnit Factory`. Let the generated test suites for the two versions *V1* and *V2* be *T1* and *T2*, respectively. Their approach then ran test suite *T1* on *V2*, and test suite *T2* on *V1*. They found 20-30% more behavioral differences, as compared to the traditional regression testing approach, i.e., executing test suite *T1* on Version *V2*. In our experiments using `JUnit Factory` for test generation, we compared our approach to the one used by Evans and Savoia [12] and showed additional benefits achieved by our approach.

Sometimes the quality of the existing tests might not be sufficient to expose behavioral differences between two program versions by causing their outputs to be different. Then some previous regression test generation approaches generate new tests to expose behavioral differences. DeMillo and Offutt [8] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during mutation testing [7]) in Fortran programs. Winstead and Evan [39] proposed a preliminary approach to generate differential tests for C programs. They use genetic algorithms to find inputs for which the two method versions behave differently. They deal with only primitive argument types. Hence their approach cannot be used for any object-oriented language, since there are more complexities involved. For example, the receiver object state may change after an invocation of a method, but the return values of the two methods under test may be the same. Korel and Al-Yami [22] proposed an approach for differential test generation for Pascal programs. They use a search-based chaining approach to generate inputs for which the two methods under test take a different execution path. They require that there is only a slight change in the methods in the two versions. Although the preceding approaches for procedural programs can be applied on individual methods of the class under test, `DiffGen` is developed to conduct regression unit testing of object-oriented programs.

Apiwattanapong et al. [2, 31] use data and control dependence information along with state information gathered through symbolic execution, and provide guidelines for testers to augment an existing regression test suite. Their approach does not generate any test case, but provides guidelines for testers to augment an existing test suite.

Some existing capture and replay techniques [11, 26, 30] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit against the captured outputs.

Ren et al. [29] developed a change impact analysis tool called Chianti. Chianti uses a test suite to produce an execution trace for two versions of a software system, and then categorizes those changes into different types. Since Chianti uses only the old test suite, it might not exercise behavioral differences between the two versions of the software system under test.

Differential symbolic execution [28] determines behavioral differences between two versions of a method (or a program) by comparing their symbolic summaries [14]. The approach also requires symbolic summaries of the methods invoked transitively from the method under test. Summaries can be computed only for methods amenable to symbolic execution. However, summaries cannot be computed for methods whose behavior is defined in external libraries not amenable to symbolic execution. Our approach still works in practice when these external library methods are present as our approach does not require summaries.

Dig et al. proposed a tool called *RefactoringCrawler* [9] that detects refactorings between two versions of a software system. *RefactoringCrawler* detects refactorings in two phases. It uses syntactic analysis to get a list of entity pairs that are similar textually in the first phase. In the second phase, it uses references among entities between the two versions to refine the results obtained from the first phase. Our previous work [35] extends *RefactoringCrawler* to detect refactorings in software libraries. *RefactoringCrawler* is complementary to `DiffGen` since *RefactoringCrawler* finds refactored entities in two versions of a software system and `DiffGen` finds methods with behavioral differences. The output of both tools helps the developers of API client code to adapt their systems to the evolved APIs.

Harman et al. proposed the idea of testability transformation [16]. Testability transformation aims to transform a program to improve the performance of a test generation technique. The transformation might not preserve the semantics of the program under test. However, the generated tests can be executed on the original version of the program under test. Our technique of transforming the program by making public the fields transitively reachable by objects of the classes under test is an instance of testability transformation. The tests generated by our approach cannot be run on the original class versions; however, our *Test Execution* phase enables the generated tests to be executed on the original source code.

8. Conclusion

Software systems are created during development, but continue to evolve throughout their (often long) lifetime. A considerable percentage of costs of maintaining such programs are due to regression testing, which is the activity of retesting a software program after it is modified. We have developed an approach and its implementation called `DiffGen`. `DiffGen` takes as input two versions of a Java class and generates a regression test suite for the two given versions. The behavioral differences between the two versions are exposed on executing the test suite. Experimental results show that `DiffGen` can effectively detect regression faults that cannot be detected by the state-of-the-art techniques [12].

9. Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443. We thank Shreyas Kale and Darko Marinov for their help in improving this paper.

REFERENCES

1. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411. ACM, 2005.
2. Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. TAICPART*, pages 137–146, 2006.
3. Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ISSTA*, pages 169–180. ACM, 2006.
4. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.
5. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proc. CCS*, pages 322–335, 2006.
6. Mike Clark. Junit primer. Draft manuscript, 2000.
7. Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.
8. Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *TSE*, 17(9):900–910, 1991.
9. Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP*, pages 404–428, 2006.

10. Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
11. Sebastian Elbaum, Hui Nee Chin, Matthew Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proc. FSE*, pages 253–264, 2006.
12. Robert B. Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *Proc. FSE*, pages 549–552. ACM, 2007.
13. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
14. Patrice Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.
15. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *Proc. PLDI*, pages 213–223, 2005.
16. Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *TSE*, 30(1):3–16, 2004.
17. Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP*, pages 431–456, 2003.
18. Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. ASE*, pages 425–428, 2007.
19. Jtopas website, 2006. `http://jtopas.sourceforge.net/jtopas/`.
20. Junit factory website, 2006. `http://www.JunitFactory.com/`.
21. James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
22. Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proc. ISSTA*, pages 143–152, 1998.
23. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
24. Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
25. Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.
26. Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions. In *Proc. WODA*, pages 29–35, 2005.
27. Parasoft Jtest manuals version 4.5. Online manual, 2003. `http://www.parasoft.com/`.
28. Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.
29. Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. OOPSLA*, pages 432–448, 2004.
30. David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, November 2005.
31. Raul Andres Santelices, PavanKumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *Proc. ASE*, pages 218–227, 2008.
32. Wolfram Schulte. Pex–an intelligent assistant for rigorous developer testing. In *Proc. ICECCS*, page 161. IEEE Computer Society, 2007.
33. Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Proc. CAV*, pages 419–423, 2006. (Tool Paper).
34. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, 2002.
35. Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proc. ASE*, pages 377–380, November 2007.
36. Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, 2008.
37. Jeffrey M. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.
38. Tao Wang and Abhik Roychoudhury. Hierarchical dynamic slicing. In *Proc. ISSTA*, pages 228–238, 2007.
39. Joel Winstead and David Evans. Towards differential program analysis. In *Proc. WODA*, 2003.
40. Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.
41. Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.
42. Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proc. AST*, pages 5–11, 2007.