DiffGen: Automated Regression Unit-Test Generation

Kunal Taneja
Department of Computer Science
North Carolina State University
Email:ktaneja@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Email:xie@csc.ncsu.edu

***Abstract*— Software programs continue to evolve throughout their lifetime. Maintenance of such evolving programs, including regression testing, is one of the most expensive activities in software development. We present an approach and its implementation called DiffGen for automated regression unit-test generation and checking for 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 class versions 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 21 classes and their versions. Experimental results show that our approach can effectively expose many behavioral differences that cannot be exposed by state-of-the-art techniques.**

I. Introduction

Software programs continue to evolve throughout their lifetime. Maintaining such evolving programs is one of the most expensive activities in the process of software development. Maintaining these evolving programs involves making different kinds of changes to them. While making changes to a software program, developers need to make sure that the changes being made are intended and do not introduce any unwanted side effect. Regression testing of software gives developers such confidence. A major part of software maintenance cost is in fact spent on regression testing.

Developers need to have regression tests that can expose behavioral differences between the original version and the modified version of a software program. In some situations (such as in legacy systems), when developers do not have an existing test suite, it is expensive 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 a developer introduced while modifying the program 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.

Recently, Evans and Savoia [5] addressed the problem of detecting behavioral differences by generating tests (using `JUnit Factory` [8]) that achieve high structural coverage separately for an old and new versions of the program 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 [13]. 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 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 21 classes (taken from a variety of sources) and their versions. The experimental results show that our approach can effectively expose behavioral differences that cannot be detected by previous state-of-the-art techniques [5] based on achieving structural coverage on either version separately.

II. Related Work

Our previous Orstra approach [14] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given

```
1    class BSTOld implements set{
2       Node node;
3       int size;
4       public BSTOld()  {.....}
5       public boolean insert(MyInput m){.....}
6       public void remove(MyInput m){.....}
7       public void contains(MyInput m){.....}
8       .....
9    }
```

Fig. 1. The BSTOld class as in an old version.

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 [5] 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 [5] and showed the effectiveness of our approach.

III. 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. Figure 1 shows an old version of the class BST, while in a new version the method `insert` is modified and the other methods remain unmodified. We use this example to explain our approach in detail. We next describe the components of our approach.

A. 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), `Change Detector` selects the method `insert`, which has been modified.

B. 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

```
1   public class BSTJUFDriver{
2      public void compareInsert(BSTOld oldBST,
           MyInput input){
3         BST bstNew = new BST();
4         bstNew = copyObject(oldBST);
5         boolean b1 = bstOld.insert(input);
6         boolean b2 = bstNew.insert(input);
7         if(b1 != b2)
8            Assert(false);
9         if(bstOld.size != bstNew.size)
10           Assert(false);
11        if(!bstOld.root.equals(bstNew.root))
12           Assert(false);
13     }
14  }
```

Fig. 2. Test Driver synthesized for JUnit Factory

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 driver for `JUnit Factory` [8] to generate tests. In general, the test driver synthesized for `JUnit Factory` can be used for other test generation tools. The `JUnit Factory` test driver synthesized by the *Instrumenter* component for the `BST` example (Figure 1) is shown in Figure 2. The driver class contains one method for every changed method of the class under test. These methods are a kind of parameterized unit tests [12]. 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 2, 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 fields of `bstOld` to the fields of `bstNew` (Line 4). With the same argument (the other argument of `compareInsert`), we then invoke the changed method on the objects of the two versions of the 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) .

C. Test Generator

The *Test Generator* component uses `JUnit Factory` [8] for test generation. `JUnit Factory` is a commercial automated characterization test generator based on Agitar [1].

The input to the *Test Generator* component is the instrumented code and the test driver generated by the *Instrumenter* component. `JUnit Factory` is used to generate tests for the test driver. `JUnit Factory` tries to achieve maximum structural coverage and thus tries to generate inputs so that branches at Lines 8 , 10, and 12 of Figure 2 can be covered. If such inputs are generated, behavioral differences between the two classes are exposed.

D. Test Execution

Making public the fields of 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 execution of the generated regression tests requires the fields

of the class under test to be public, and hence the tests cannot be executed on the original source code. To execute the regression tests on the original source code, we use our previously developed Diffut framework [15] to execute the generated test suite on the original code of the class versions under test.

IV. 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 [5]?

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

A. Experimental Subjects

Table I lists eight Java classes that we use in the first experiment. UBStack is the illustrative example taken from the experimental subjects used by Stotts et al. [11]. IntStack was used by Henkel and Diwan [6] in illustrating their approach of discovering algebraic specifications. ShoppingCart is an example for JUnit [3]. BankAccount is an example distributed with Jtest [10]. The remaining four classes are data structures previously used to evaluate Korat [2]. 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 coverage achieved by tests generated by JUnit Factory for the original version.

B. Experimental Setup

Although our ultimate research question is to investigate whether 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 [9], 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 [8].

To evaluate the fault-detection capability of our DiffGen approach, we compare the fault-detection capability of the test suite generated by DiffGen with the fault-detection capability of test suites generated by the approach of Evans and Savoia [5]. Their approach generates tests separately for the two versions of class under test with JUnit Factory. Then their approach runs the generated tests for the original version on a mutant version, and runs the generated tests for the mutant version on the original version of the class under test to expose behavioral differences. For simplicity, we refer to their approach as SeparateGen. 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.

C. Measures

We measure the total number of mutants generated for each subject, the number of unkilled mutants 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 same behavior as the original version of class. IF2 = $\frac{k}{u-s}$. The values of IF1 and IF2 indicate the extra fault-detection capability of DiffGen over SeparateGen.

D. Results

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 unkilled mutants by SeparateGen. Columns 4 shows the number of mutants killed by DiffGen, among the mutants that were not killed by SeparateGen. 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 of DiffGen, respectively. From Table II, we observe that DiffGen has an Improvement Factor IF2 varying from 40% to 100% for all the subjects with the exception of DisjSet, which has an improvement factor of 23.4%.

In summary, the evaluation answered the question the we presented in the beginning of Section IV. 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 was able to detect from around 23% to 100% of faults that SeparateGen could not detect.

E. 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) [4]. We conducted experiments on three available versions of the JTopas [7] 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 were available for them. 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 subjects were the same ones used by Evans and Savoia [5].

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 used by Evans and

TABLE I
EXPERIMENTAL SUBJECTS

class	meths	public	ncnb	Cov
IntStack (IS)	5	5	44	100%
UBStack (UBS)	11	11	106	100%
ShoppingCart (SC)	9	8	70	100%
BankAccount (BA)	7	7	34	100%
BinSearchTree (BST)	13	8	246	100%
BinomialHeap (BH)	22	17	535	87%
DisjSet (DS)	10	7	166	100%
FibonacciHeap (FH)	24	14	468	98%

TABLE II
EXPERIMENTAL RESULTS

class	#Mutants	#JUF UnKilled	DG Killed	Same Behavior	IF1 %	IF2 %
IS	85	21	0	21	0	-
UBS	187	15	6	7	40	75
SC	18	7	3	4	42.8	100
BA	35	6	0	6	0	-
BST	125	13	4	4	30.8	44.4
BH	281	39	8	19	20.5	40
DS	385	97	15	33	15.5	23.4
FH	339	53	5	43	9.4	50

Savoia. Finally the last column shows the number of faulty versions detected by our approach among the ones not detected by the approach used by Evans and Savoia.

The results show that our `DiffGen` approach detects 5 of the 7 faults that were not detected by the approach used by Evans and Savoia.

V. DISCUSSION

In this section we discuss some of the limitations of the current implementation of our tool 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). The *Change Detector* component in the *Test Generation* phase detects corresponding methods in the two versions by 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 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 classes 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 [14].

TABLE III
EXPERIMENTAL RESULTS

Ver	class	LOC	F	U	D
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

VI. CONCLUSION

Software programs 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 generated test suite. Experimental results show that `DiffGen` can effectively detect regression faults that cannot be detected by the state-of-the-art techniques.

ACKNOWLEDGMENTS

This work is supported in part by NSF grant CCF-0725190.

REFERENCES

[1] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ISSTA*, pages 169–180, 2006.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

[3] M. Clark. JUnit primer. Draft manuscript, 2000.

[4] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[5] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. FSE*, pages 549–552, 2007.

[6] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP*, pages 431–456, 2003.

[7] JTopas website, 2006. http://jtopas.sourceforge.net/jtopas/.

[8] JUnit Factory website, 2006. http://www.junitfactory.com/.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[10] Parasoft Jtest manuals version 4.5. Online manual, 2003. http://www.parasoft.com/.

[11] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. XP/Agile Universe*, pages 131–143, 2002.

[12] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[13] J. M. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.

[14] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.

[15] T. Xie, K. Taneja, S. Kale, and D. Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proc. AST*, pages 5–11, 2007.