Automated Unit Test Generation for Evolving Software

Sina Shamshiri
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello, Sheffield, UK, S1 4DP
sina.shamshiri@sheffield.ac.uk

ABSTRACT

As developers make changes to software programs, they want to ensure that the originally intended functionality of the software has not been affected. As a result, developers write tests and execute them after making changes. However, high quality tests are needed that can reveal unintended bugs, and not all developers have access to such tests. Moreover, since tests are written without the knowledge of future changes, sometimes new tests are needed to exercise such changes. While this problem has been well studied in the literature, the current approaches for automatically generating such tests either only attempt to reach the change and do not aim to propagate the infected state to the output, or may suffer from scalability issues, especially when a large sequence of calls is required for propagation. We propose a search-based approach that aims to automatically generate tests which can reveal functionality changes, given two versions of a program (e.g., pre-change and post-change). Developers can then use these tests to identify unintended functionality changes (i.e., bugs). Initial evaluation results show that our approach can be effective on detecting such changes, but there remain challenges in scaling up test generation and making the tests useful to developers, both of which we aim to overcome.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

Keywords

Automated Unit Test Generation, Genetic Algorithms, Search-Based Testing, Regression Testing

1. INTRODUCTION

Developers evolve software programs by introducing many changes throughout the life-cycle of the software. These changes often range from small refactorings to the addition of large new features. However, some of these changes may affect the originally intended functionality of the software, by introducing unintended bugs – also known as *regression faults*. To avoid regressions in the functionality, engineers write tests as they develop the software, and after making changes developers execute these tests to increase their confidence that the intended functionality of the software is intact. This practice is also referred to as *regression testing* and is commonly used in the industry.

While regression testing can help with early detection of regression faults, developers face several challenges when applying the technique. As the number of tests grows, execution of all tests after every single change can become expensive and impractical. This problem has been well studied in the literature [18] and many techniques such as test selection, prioritization and minimization have been proposed.

The challenges however are not limited to the growing cost of regression testing. Even if all tests are executed, three main problems remain: 1) an existing set of tests is required, 2) the tests are often written without foreseeing future changes, and 3) the effectiveness of the tests in finding regression faults depends on the quality of the written tests. According to the PIE model [15], to reveal a fault, a test has to first execute the fault, infect the state and finally propagate it to the output. While several techniques exist for augmenting existing test suites (e.g., [10, 17]) and generating regression tests (e.g., [2,9,13,14]), the techniques mainly focus on reaching the fault, yet the number of paths to propagate the infected state to the output can explode, which may impose a limit on the scalability of the approach [3].

To address the previous shortcomings, we propose a technique for generating a regression test suite (i.e. a set of unit-tests which contain a sequence of calls executing the class under test) without depending on existing tests. Our approach takes two versions of a class under test, and uses a search-based algorithm [8] with the objective of reaching and propagating the changes between the two versions of the program. We have implemented our approach named EvoSuiteℝ on top of the EvoSuite [5] test generation tool, and our early evaluation of the technique [11] showed encouraging results on examples with propagation issues (i.e. where covering the change alone does not propagate the changed state to the output). Further attempts to evaluate the effectiveness of our approach on detecting real regression faults revealed several challenges. As a part the remaining course of this research we aim to solve these challenges, in addition to evaluating our approach against the state-of-the-art.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803196

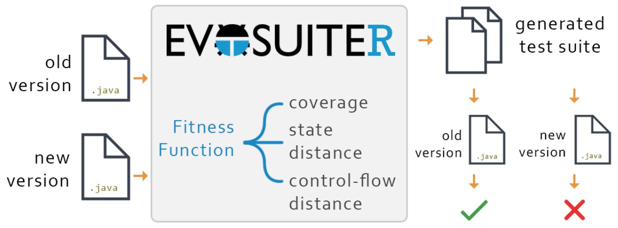

Figure 1: Overview of the approach. Given two versions of a Java class, EvoSuiteR aims to generate a test suite which passes on one version of the class and fails on another one.

2. BACKGROUND

Although the topic of regression testing is well studied in the literature, the majority of the studies focus on reusing existing tests. However, since tests are written prior to the changes, even with an existing test suite they can be inadequate for exercising changed behavior. As a result, researchers have looked at augmenting existing test suites with new tests that can exercise these changes (e.g., [1,17]). Xu et al. [16] also identified that the manner in which existing tests are used for test suite augmentation can affect the outcome of the approach. While these techniques were found to be effective, an existing test suite is still required, and the quality of the provided test suite has an impact on the outcome and performance of the techniques.

Less attention however has been given to automatically generating regression tests. Several techniques such as eXpress [14] use dynamic symbolic execution (DSE) to generate regression tests. Over the past years researchers have made many advancements towards scalability of approaches using DSE to avoid the path explosion problem. However, considering the PIE model, even if the reachability aspect is solved, state infection and propagation remain as separate challenges, since propagation alone can lead to another path explosion problem. A different approach by Boheme et al. [2] looks at generating regression tests using input values that are symbolically partitioned such that they evaluate into the same differential behaviour.

Work has also taken place on using random test generation tools for detecting regression faults. Jin et al. [6] in their approach named BERT, first use a test generation tool such as Randoop to generate a set of tests on the pre-change version of the software, and then execute the generated tests on the post-change version, and observe the behavioral differences. Another approach named DiffGen by Taneja and Xie [13] generates regression tests using instrumented test drivers. By comparing the source code, DiffGen takes methods that are semantically different across two versions of a program, and synthesizes test drivers that compare the execution result of methods. To test their synthesized methods, they use an automated test generation tool to generate tests for their test drivers. Nevertheless, two main challenges remain: 1) neither of the techniques focus on state infection and propagation, 2) no large empirical evaluation of the techniques exists to enable a more in-depth research in this area.

3. APPROACH

To aid the developer with early detection of regression faults before releasing the software, our goal is creating a solution that can take two versions of a program (e.g., before and after a change), and generate regression tests with embedded assertions that fail on one version and pass on the other one. Since the tests reveal the detected changes in the functionality, developers can then assess whether or not the changes were intentional.

3.1 A Search-Based Approach

To achieve this, we investigated the use of a search-based approach for automatically generating regression tests. Specifically, we focused on using a Genetic Algorithm (GA) to search for solutions that can satisfy our objective. A GA is an evolutionary algorithm which over time evolves a population of individual solutions towards an objective, using a fitness function which enables us to optimize for multiple goals simultaneously. In our case, each individual solution (a.k.a. chromosome) is a test suite (set of test cases), and the fitness function calculates how far an individual is from the ultimate solution. Individuals in the population are initially generated randomly, and then evolve over time by the two functions of mutation (i.e., test suite is modified by adding/removing test cases or the statements within the test cases are mutated) and crossover (i.e., two individuals are recombined to form two new offspring).

Since the fitness function drives the direction of the search, we considered three main measurements: 1) *coverage*: the level of goal-coverage achieved by each test suite on both versions of the code, where a *goal* is defined as covering all branches in the class under test in addition to all methods that do not contain any branches, 2) *state distance*: after executing each test suite, how different is the state of all objects in the test suite across the two versions, and 3) *control-flow distance*: for each branch of the software, how far are the two versions from diverging. An overview of our approach is shown in Figure 1.

To implement our approach, we extended EvoSuite and added functionality to support execution and state-capturing of two versions of the same class. To capture and compare the state of the program, we used Java reflection to extract the public and private state of the objects in the test suite, and then compared the values based on their numerical object distance [4]. Finally, if the same test inputs exercises the change and different output values are observed, we add assertions based on the output observed from the original (i.e. pre-change) version. Since in practice we can find methods that produce different output values after each execution (e.g., a random number generator), a threat exists to our approach that such false positives may be detected. To avoid this and to lower the chance of false positive solutions, we re-execute each test case at least 3 times, and the test is only kept if at each execution, it passes on the original version and fails on the changed version.

3.2 Results Achieved So Far

3.2.1 Proof of Concept

To evaluate our technique, we first used several small-sized non-trivial examples (e.g., CreditCard [11], BankAccount [6]) where simply reaching the change and infecting the state would not result the internal different states to propagate to the output. We found our approach to be more effective than the state of the art test generation tools (i.e. BERT using Randoop and EvoSuite) in detecting the changes, and moreover, our fitness function was more successful compared to using coverage alone [11]. As a result, we investigated the effectiveness our approach on two large sets of artificial (mutants) and real bugs.

For the set of mutants, we wanted to understand whether our technique can detect more mutants than the state of the art test generation tools. As a result, we used over 500 mutants of two open-source projects[1] that could not be detected by tests generated by the tools EvoSuite and Randoop. Additionally, for the set of real bugs, we used the Defects4j repository [7] which contains 357 bugs across 5 different open source projects. Finally, to better understand the effectiveness of each metric in our fitness function (e.g., coverage, state-distance and control-flow distance), we compared the effectiveness of all combinations of the metrics, in addition to each of them individually. Overall, the combination of all three metrics was the most effective.

However, after comparing our GA with random search, we found random search to be quite capable in identifying the changes. Although several cases were only found by the GA, on average random search was as successful as GA, and even more successful on few other cases. This finding presented us with the question of why an unguided search technique can be equally or more successful.

3.2.2 Limitations of search-based approaches

After further investigating the surprisingly successful outcome of random search, we proceeded to study whether in-practice this outcome is limited to regression test generation, or whether it may generalize to test generation. The result of the study showed no significant difference between GA and random search for over 78% of subjects [12]. We found the two underlying reasons for this observation: 1) A large number of object-oriented classes were simple enough for both techniques to cover, and 2) The type of conditional branches common in object-oriented programs, are conditions that cannot be used currently by the GA for guidance(e.g., a condition on the return value of a method).

In addition to the search algorithm, we found our implementation to have a higher overhead compared to random search – evaluating up to 3x fewer tests during the search. This enables random search to explore more solutions, whereas the GA spends a considerable portion of the budget on capturing and comparing the state. Furthermore, this raises a new question of whether a metric for increasing the diversity needs to be introduced.

3.2.3 Evaluating real regression faults

Another challenge we were presented with was evaluating our technique against the state of the art. As a necessary step towards enabling the evaluation of regression test generation tools we conducted a large empirical study into the effectiveness of the state-of-the-art test generation tools on detecting real faults [2]. In the study, we generated tests using 3 tools – EvoSuite, Agitar (commercial tool), and Randoop – on the post-fix version of 357 bugs from the Defects4J repository, and then executed the tests on the prefix version to see whether the regression in the functionality can be detected. Our results showed that although the tools overall managed to find over 55% of faults, no tool managed to individually find more than 40.6% of the faults. This is while the tools managed to reach high levels in coverage, even up to an overall average of 86.7% by one of the tools.

We also identified several key weaknesses/strengths of the tools. For instance, some of the bugs were only found by one of the tools which accessed/validated the private state of the class under test. Moreover, we found that for test suites that were unable to detect the fault, over 50% of them did fully cover the changed code. While this may indicate a lack of adequate oracles (i.e. assertions), it also shows that either the fault infected the state of the program and yet did not propagate to the output, or a specific dataflow was required to infect the state, which we aim to investigate further.

[1]Numerics4j and Apache Commons IO

[2]To appear in ASE2015.

4. REMAINING CHALLENGES

As mentioned in Section 3, after evaluating our approach and the state-of-the-art test generation tools, we identified several key challenges. In this section, we discuss these challenges and how we aim to solve and evaluate them in the remaining course of the study.

4.1 Isolation of Changes

Our approach currently does not take into account how many times the same change is revealed. As a result, many different test cases can be generated which identify the same change, or one test may reveal the same change more than once. Moreover, reviewing all such tests manually by the developer can be time-consuming and costly. We aim to minimize tests by determining and grouping tests that reveal the same change, such that the developer is presented with a set of unique changes. To evaluate this, we will specifically measure the reduction in the length and number of tests, to see whether any significant reductions can be made.

4.2 Continuous Test Suite Augmentation

Once a test reveals a change which the developer classifies as a regression fault, it is possible that the same fault can happen in the future. We propose a continuous regression test generation technique, where on every run of our approach, we improve the previously generated regression test suite by augmenting it with tests that had previously revealed changes. To evaluate this, we are interested in the rate of growth of the test suite size over time, and whether the tests can be more resilient to regressions in the future, especially when compared to tests generated by coverage-based techniques.

4.3 Addressing State Infection

As mentioned in Section 1, to reveal a fault, a test has to reach the fault, infect the state and then propagate it. While our approach aims to reach/execute changes by maximizing the coverage, and also aims to propagate the infected state to the output using the state-distance/control-flow-distance metrics, the infection of state is left to chance. For instance, in a test that even fully covers the class under test, to infect the state, the order in which the statements are executed is important. Consider the class `Foo` as below:

```
public class Foo{
    private int value = 1;

    public void setValue(int new){
-       value = new;
+       value = 2 * new;
    }

    public int getValue(){
        return value;
    }
}
```

Observe that the `setValue` method has been changed to set the private field `value` to twice the provided input. A simple test can detect the change:

```java
public Test0(){
   Foo foo = new Foo();
   f.setValue(1);
   int value = f.getValue();
   assertEquals(value, 1);
}
```

In the test case above, the assertion fails on the changed version – and thus detects the change, since `value` is now equal to 2. However, if the lines 3 and 4 in the test case above are swapped, the coverage remains at 100%, yet the test is unable to identify any changes. One way to overcome this would be to derive symbolic conditions that can result in state infections. To avoid the issue of path explosion, we can increase of the chance of state infection by increasing the diversity of our solutions.

Currently, none of the measurements used in the fitness function of our GA reward the use of different data flows in the test suites. As a result, for instance, if a branch is already covered with one data-flow, the search can get stuck in a local optima, since any new solution that covers the same branch in a different way is discarded. We propose using a diversity metric that maximizes the use of different data-flows, and as a result can grow the test suite's diversity over time. Since increasing the diversity may also result in spending more time on discovering new solutions, we aim to measure the success rate of the approach to evaluate whether the tradeoff can be beneficial to the outcome of the search.

4.4 Addressing Scalability

One of the shortcomings of our approach is the high overhead of capturing and comparing state differences, which can account to even half of the search budget. Since this process is performed as part of the fitness evaluation of every individual and execution data is captured after each statement execution in the test suite, for instance, an increase in the size and complexity of the class under test can result in an increase in the level of memory consumption. We aim to investigate and evaluate different optimization techniques to lower this overhead, to limit the amount of data that is captured and compared.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented our approach for generating regression tests based on two version of a software, using a search-based technique. Our approach guides the search towards maximizing the state distance across the versions, to increase the chance of propagating the different states to the output. Although the results of our early experiments on small non-trivial examples were promising, on detecting real regressions we found random search to be equally performant for the majority of subjects in our experiment. Specifically, we learned about challenges presented by the search space, where the GA cannot use the majority of conditional branches for guidance, and the necessity of improving the GA to overcome these. As a result, we identified and discussed some of the shortcomings of our approach such as usability of the tests, effectiveness of our generated solutions and the overhead imposed by our technique. One key remaining challenge for us is that while EvoSuiteℝ aims to reach and propagate the state changes, state infection is still by chance. Therefore, we aim to maximize this chance by increasing the diversity in our population, or by deriving symbolic conditions to achieve state infection.

After overcoming the challenges presented in this paper, we plan to conduct a large empirical study to evaluate the effectiveness of our technique compared to the state-of-the-art test generation tools. Especially, we are interested to understand the difference in the effectiveness of our regression testing approach, when compared to coverage-based test generation tools. Furthermore, the usability of such automated regression tests to the developer is unknown. After enhancing the usability of our technique, we plan to empirically evaluate it with developers. By the end of this study we will make all the data and source code available open source, to assist future research on the topic.

6. REFERENCES

[1] Apiwattanapong, T., Santelices, R., Chittimalli, P.K., Orso, A., Harrold, M.J.: Matrix: Maintenance-oriented testing requirements identifier and examiner. In: Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings. pp. 137–146. IEEE (2006)

[2] Böhme, M., Oliveira, B.C.d.S., Roychoudhury, A.: Partition-based regression verification. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 302–311. IEEE Press (2013)

[3] Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM 56(2), 82–90 (2013)

[4] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Object distance and its application to adaptive random testing of object-oriented programs. In: Proc. Workshop on Random testing. pp. 55–63. ACM (2006)

[5] Fraser, G., Arcuri, A.: Whole test suite generation. Software Engineering, IEEE Transactions on 39(2) (2013)

[6] Jin, W., Orso, A., Xie, T.: Automated behavioral regression testing. In: Proc. ICST. pp. 137–146. IEEE (2010)

[7] Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). pp. 437–440. San Jose, CA, USA (July 23–25 2014)

[8] McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)

[9] Orso, A., Xie, T.: Bert: Behavioral regression testing. In: Proc. WODA. pp. 36–42. ACM (2008)

[10] Santelices, R., Chittimalli, P., Apiwattanapong, T., Orso, A., Harrold, M.: Test-suite augmentation for evolving software. In: Proc. ASE. pp. 218–227. IEEE (2008)

[11] Shamshiri, S., Fraser, G., McMinn, P., Orso, A.: Search-based propagation of regression faults in automated regression testing. In: Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on. pp. 396–399. IEEE (2013)

[12] Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference. pp. 1367–1374 (2015)

[13] Taneja, K., Xie, T.: Diffgen: Automated regression unit-test generation. In: Proc. ASE. pp. 407–410. IEEE (2008)

[14] Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: express: guided path exploration for efficient regression test generation. In: Proc. ISSTA. pp. 1–11. ACM (2011)

[15] Voas, J.: Pie: A dynamic failure-based technique. Software Engineering, IEEE Transactions on 18(8), 102–112 (1992)

[16] Xu, Z., Cohen, M.B., Rothermel, G.: Factors affecting the use of genetic algorithms in test suite augmentation. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. pp. 1365–1372. ACM (2010)

[17] Xu, Z., Rothermel, G.: Directed test suite augmentation. In: Proc. APSEC. pp. 406 –413 (2009)

[18] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22(2), 67–120 (Mar 2012)