Search-Based Propagation of Regression Faults in Automated Regression Testing

Sina Shamshiri*, Gordon Fraser*, Phil McMinn*, Alessandro Orso†
*Dept. of Computer Science, University of Sheffield, UK
{sina.shamshiri, gordon.fraser, p.mcminn}@sheffield.ac.uk
†Georgia Institute of Technology, US
orso@cc.gatech.edu

***Abstract*—Over the lifetime of software programs, developers make changes by adding, removing, enhancing functionality or by refactoring code. These changes can sometimes result in undesired side effects in the original functionality of the software, better known as *regression faults*. To detect these, developers either have to rely on an existing set of test cases, or have to create new tests that exercise the changes. However, simply executing the changed code does not guarantee that a regression fault manifests in a state change, or that this state change propagates to an observable output where it could be detected by a test case. To address this propagation aspect, we present EvoSuiteℝ, an extension of the EvoSuite unit test generation tool. Our approach generates tests that propagate regression faults to an observable difference using a search-based approach, and captures this observable difference with test assertions. We illustrate on an example program that EvoSuiteℝ can be effective in revealing regression errors in cases where alternative approaches may fail, and motivate further research in this direction.**

***Keywords*-regression testing; search based software engineering; automated regression testing; search based testing**

I. Introduction

Throughout the lifetime of a software program, it evolves and undergoes numerous changes, as the developers may intend to add, enhance, or refactor functionality. These modifications to the code can result in unintentional bugs commonly known as *regression faults*. In order to increase the confidence of developers over such modifications, regression testing is used to check that the main functionality is still intact and any change in the software behaviour is intentional. A common assumption is that developers have a set of test cases prepared before making modifications to software, and they re-run the tests after making changes to the code. Any test failing informs the developers of a regression fault.

However, creating such test suites is known to be expensive and time-consuming, and the quality of the existing test suite will be the main determinant of the success of regression testing. In practice, developers create test suites that only partially cover the structure of the program. Even assuming there is a high coverage test suite, this cannot guarantee that regression faults will be detected.

To address these issues, many attempts have been made to generate regression tests automatically. Although it is well understood how to generate test cases that reach and exercise changed code, the task of making the tests *propagate* any state change induced by a regression fault to an observable output is still an open issue. In particular, current approaches to achieving propagation attempt to exhaustively calculate execution paths and therefore suffer from scalability issues [1] and have to restrict the depth of the exploration.

In this paper, we propose an approach named EvoSuiteℝ as an extension of EvoSuite automatic test suite generator [2] in order to create a regression test suite. EvoSuiteℝ tries to find regression faults between two versions of a Java class, using a search-based approach, aiming to direct the regression faults to propagating the output. Our approach optimises towards better test suites by guiding the search to reach maximum behavioural differences.

The main contribution of this paper is our approach in finding regression faults via a *search-based technique* that seeks to propagate and detect regression faults by:

1) Reaching maximum code coverage by optimising test cases towards better coverage in both versions of the class.
2) Maximising state differences between the two versions of the given class in order to propagate state infections to observable differences.
3) Maximising control-flow differences between the two versions to propagate regression faults.

Additionally, a case study is presented with a non-trivial example, in which our approach successfully propagates the regression fault.

II. Background and Motivating Example

In order to detect a regression fault in a program [3], a test should be able to first, reach and execute fault. Secondly, the execution should result in a change and therefore infect the state of the program. Finally, it should propagate to an observable difference in the output.

To discuss how different approaches in regression testing tackle these three aspects, we present a simple, but non-trivial, motivating example. To construct such an example, a case was considered that a method has to be called for a number of times for the regression fault to be propagated to the output. The class `CreditCard`, as shown in Figure 1, considers a scenario in which the credit card company allows a customer to withdraw as long as their minimum

```java
public class CreditCard {

  private int minRepayment = 0;
  private int balance = 0;

  public boolean withdraw(int amount) {
    if(canWithdraw() == false)
      return false;

    minRepayment+= 5 * 2 ;
    balance += amount;
    return true;
  }

  public boolean canWithdraw() {
    return (minRepayment <= 200);
  }
}
```

Figure 1. In the `CreditCard` class, the highlighted change will not immediately affect the program behaviour. However, after 21 calls to `withdraw()`, `canWithdraw()` would return false, resulting in a regression fault.

repayment reaches a certain amount. The minimum payment is calculated as a fixed amount that increases based on the number of withdrawals. Over time, the credit card company decides to double the minimum repayment rate, however the developers forget to suitably raise the withdrawal limit. While the program is changed, a regression fault will not show up unless `withdraw()` is called between 20 and 40 times.

Early work on regression testing only used existing test suites in order to find regression faults. As it may be expensive to run a complete test suite after making any change to the software, Elbaum et al. [4] proposed prioritising parts of the test suite that can achieve coverage more quickly. Similarly, Harrold et al. [5] proposed test suite minimisation that selected a subset of the set suite that would provide the same level of coverage. Since such approaches relied solely on existing test suites, success would only be by chance and depended on the quality of the tests. Another aspect of the example (Figure 1) is the fact that, if the developers already have a test suite with 100% structural coverage, this still would not guarantee that the regression would be detected.

Test suite augmentation is another aspect in regression testing that has drawn the interest of many researchers. Xu and Rothermel [6] introduced DTSA, which was able to add new test cases to the existing test suite for the modified parts of the program, to create a coverage-adequete test suite. Teneja et al. [7] generated regression tests using dynamic symbolic execution while employing techniques such as pruning the unchanged paths to further optimize their search strategy. These approaches aimed to solved the reachability aspect while leaving propagation to chance.

Other approaches in the area looked at using external or randomly generated test suites in order to find regression faults. Orso and Xie [8] presented BERT as a behavioural regression testing tool, that operated in three stages. The first stage used either a set of test cases randomly generated by third-party tools (e.g. [9], [10]) or an existing test suite. Secondly, it executed the tests over both the old and new versions of the class under the test while capturing their outcome, and finally analysed the observed behavioural difference and presented the result back to the user. A similar approach taken by Taneja and Xie [11] attempted to add new branches to the code, that if covered by a random testing tool would expose a regression fault. The resulting tool DiffGen synthesised a test driver, that would try to execute the branches on both versions. While regression testing with randomly generated test suites was found to be useful, the quality of the test is still determined by the existing or generated tests, and reaching any of the three elements is by chance. With the `CreditCard` example, these approaches can only succeed if the provided test suites can identify a behavioural difference.

The previous approaches did not however focus on addressing the propagation of regression faults. Santelices et al. [12] used symbolic execution of the different versions of the program, to augment the existing test suite with new test cases that would consider the change-propagation paths. More recently, Santelices and Harrold [1] proposed an approach for propagation-based testing of changes. These approaches have scalability limitations, as the number of possible execution paths exponentially grow and therefore a boundary has to be defined.

While previous attempts made in regression testing were found to be effective, they mostly have shortcomings such as a) relying on third-party test generation tools while not handling fault propagation, as execution of a regression fault may not propagate to a change in the behaviour [8] [11], b) relying solely on existing test suites and therefore leaving reachability, infection and propagation to chance [8] [4] [5], c) augmenting test suites while aiming to only achieve coverage [6] [7], and d) in cases that consider propagation, they either have scalability limitations due to high computation costs, or do not consider a sequence of method calls that may lead to reaching a fault [12] [1].

III. Propagating Regression Faults with Search-based Testing

EvoSuiteℝ attempts to solve the previous shortcomings by using a search-based approach in finding regression faults. The main objective is to generate test cases that can propagate to different states between the versions of the program under test. To achieve this, EvoSuiteℝ performs the following steps.

A. Overview

EvoSuiteℝ is an extension of the EvoSuite platform [2] and therefore relies on evolutionary algorithms. Using a genetic algorithm [13], an initial population of random test suites is first created. Over time individuals are randomly mutated and reproduced; these operations depend on the underlying representation. The probability of an individual for being selected for reproduction depends on its *fitness value*, which estimates the distance towards an optimal solution. Therefore the best test suite always has the best fitness value compared to its ancestors [2].

B. Representation

Individuals in the population are represented by regression chromosomes, which extend the chromosomes used by EvoSuite [2] such that execution of a test case leads to two traces for the two versions of a tested class. Details on the mutation and crossover operators can be found in [2]. The traces serve as input for the fitness function.

C. Fitness Function

Since the direction of the evolution of the generated test suites depends on the fitness function, the fitness value is the key towards reaching better test suites. Three main measurements were considered as our fitness guidance. At first, it is important that the test suites are able to execute and cover the code under the test, therefore focusing on reachability. Thereafter, aiming to direct regression faults to propagate to an observable difference, first the state of the program after the execution of each statement has to be observed and compared to find execution differences. Secondly, the fitness should guide the search to cause diversions at branches, therefore creating a change in the control-flow.

The overall fitness value is calculated by combining the value of the following measurements:

1) Structural Coverage: Before being able to propagate the regression faults to an observable behavioural difference, it is important to reach and execute the faulty code, so as to solve the reachability aspect of regression testing. Furthermore, as every part of the class may potentially contribute towards propagation, we aim to cover *all* the code. Therefore, one of the measures for assessing the fitness of the test suite is the level of coverage it has over the class and its branches. To measure this value, based on the stored execution traces the value of the branch coverage fitness function [2] is calculated for both versions and added.

2) Object Distance: It is expected that while executing a regression fault, if the two versions of the software are generating different values (either internally inside the objects or externally as observable return values), increasing state difference between two versions of a software would raise the chance of the fault to propagate to an observable output (i.e., the public API of the class). Therefore, another measure used in calculating the fitness is the observable difference based on the stored execution traces. In order to calculate this value, initially the execution traces are analysed, and the objects resulted from the execution of each statement in the test suite are recursively inspected to create a set of comparable primitive data values. To capture the values for non-primitive data types, the Java Reflection Framework is used to access the values inside objects in a recursive manner, regardless of their access modifiers. Finally, the observed execution values are compared between the two versions and a fitness value is returned based on the calculated numerical object distance [14] between the two versions. The maximum calculated distance will be normalised and added to the total distance. The inverse of the total distance is finally added to the main fitness value.

3) Control-flow Distance: In order to further assist the regression fault to propagate, we identified another measurement to better guide the search. Generally during the execution, if a test case results in a diversion at a particular branch between the versions, it raises the chance of the regression fault to propagate to the output, due to the change in the control flow. As each part of the control flow may contribute towards propagation, we aim to cause a diversion in the control flow at as many branches as possible, and we thus implemented a control-flow distance measure for each branch of the code. This measurement decreases as the branches get closer to diversion. To achieve this measurement, first similar branches between the classes get mapped together using JDiff [15]. Afterwards, the control-flow distance value is calculated at branch level. For each method call executed on a test suite, the branch state is observed in both versions of the code. The function `distance` calculates the distance of a branch to becoming true or false – also known as *branch distance* [13] – for both versions and compares them. Additionally, the actual branch distance value is normalised between 0 and 1 denoted as ω, such that $0 \leq \omega(b_i) \leq 1$. If a diversion is observed at a branch between the two versions, the minimum fitness value 0 is returned. Otherwise, the fitness function for a test suite with method calls $C = \{c_1, \ldots, c_n\}$ and branches $B = \{b_1, \ldots, b_m\}$ is calculated as follows:

$$\text{cfgDis}(B) = \sum_{i=1}^{m} (min_{j=1}^{m}(1 - \text{distance}(c_j, b_i) + \omega(b_i)))$$

The fitness is expected to decrease until a diversion is observed, for which the best fitness value 0 is returned. Finally, the sum of the minimum distance values for each branch among all method executions will be normalised and added to the total fitness.

D. Finalising and Assertions

After the search finishes, EvoSuiteℝ will pick the test suite with the best fitness value and will analyse the test suite on both versions for one last time. At this stage, the test statements will be executed on both versions of the class under test, and if the test can reveal a difference between the two versions, EvoSuiteℝ adds assertions to the test. This enables the developer to see that a behavioural difference exists, and to fix the code until it passes.

IV. Case Study

For an initial assessment of the competency of our approach, we used the `BankAccount` [8], [16] example and our approach succeeded in identifying the different behaviour. Being successful in propagating the regression faults in such basic examples, we used the motivating example from Section II as a case study. Additionally, in order to test the effectiveness of our propagation strategy, we have experimented our approach without the propagation fitness measures (i.e., just with branch coverage). The results of the experiments show that, our approach was not only effective in finding the regression, but it also outperformed a search-based regression testing with coverage as the only fitness measure.

In order to restrict the duration of the search, we used a time limit as the search budget. Considering that an evolutionary algorithm is used for searching, the number of generations created may affect the chance of generating better test suites that can identify the regression faults. Therefore, to evaluate the performance of our approach, we used the three time budgets of short, medium and long with durations of respectively 3, 5 and 10 minutes. To compare with alternative approaches where setting time limits was not allowed, the number of randomly generated test cases has been modified to respectively 200, 500 and 2000, to achieve the same duration. In order to ensure that the generated result of the software has not been by chance, each method-budget pair has been tested for at least 30 times.

Table I lists the methods used to test the `CreditCard` class versions, with their success rate in finding the regression fault respective to the used search budget. Based on the results, running EvoSuiteℝ with a higher search budget results in higher success rate, meaning that the given guidance is effective in finding regression faults.

Additionally, it can be seen that with respect to the increase in search budget, the success rate of search-based testing with coverage as the only measurement did not significantly increase. This is while the combined fitness values were able to reach a higher success rate due to having a better guidance. In comparison to the existing regression testing approach BERT, even with very large number of test cases, remains unable to find the regression fault due to the nature of random testing.

Table I
Success rate in finding the regression

Method	**Short**	**Medium**	**Long**
EvoSuiteℝ	66%	75%	90%
EvoSuiteℝ with Coverage only	3%	12%	13%
BERT with Randoop	0%	0%	0%

V. Conclusion and future work

In this paper, we presented EvoSuiteℝ, as a new approach to automated regression testing and test suite generation. EvoSuiteℝ uses search-based techniques to optimise towards test suites that are able to find regression faults in the program. Our approach not only generates test suites, but also aims to solve the propagation aspect of regression testing.

Our initial experiments provide initial, encouraging evidence that our approach succeeds in propagation. As a next step, we will experiment with different ways of combining the fitness measurements introduced in this paper. The fitness function discussed in this paper addresses the reachability and propagation aspects of regression testing. We will investigate how to extend this by addressing also state infection, for example using dynamic symbolic execution. Furthermore, a large empirical study has to be conducted to examine the effectiveness of our approach in practice.

References

[1] R. Santelices and M. Harrold, "Applying aggressive propagation-based strategies for testing changes," in *Proc. ICST*, 2011, pp. 11–20.

[2] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proc. QSIC*. IEEE, 2011, pp. 31–40.

[3] J. Voas, "Pie: A dynamic failure-based technique," *Software Engineering, IEEE Transactions on*, vol. 18, no. 8, pp. 102–112, 1992.

[4] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, 2000.

[5] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, 1993.

[6] Z. Xu and G. Rothermel, "Directed test suite augmentation," in *Proc. APSEC*, 2009, pp. 406 –413.

[7] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "express: guided path exploration for efficient regression test generation," in *Proc. ISSTA*. ACM, 2011, pp. 1–11.

[8] A. Orso and T. Xie, "Bert: Behavioral regression testing," in *Proc. WODA*. ACM, 2008, pp. 36–42.

[9] Agitar, "*JUnit Factory* URL: http://www.agitar.com/developers/junit_factory.html," 2013, Last visited on 14.01.2013.

[10] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Proc. OOPSLA*, 2007, pp. 815–816.

[11] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *Proc. ASE*. IEEE, 2008, pp. 407–410.

[12] R. Santelices, P. Chittimalli, T. Apiwattanapong, A. Orso, and M. Harrold, "Test-suite augmentation for evolving software," in *Proc. ASE*. IEEE, 2008, pp. 218–227.

[13] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

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

[15] T. Apiwattanapong, A. Orso, and M. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, no. 1, pp. 3–36, 2007.

[16] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Proc. ICST*. IEEE, 2010, pp. 137–146.