2011

Directed Test Suite Augmentation

Zhihong Xu
University of Nebraska - Lincoln, zxu@cse.unl.edu

2011 33rd International Conference on Software Engineering (ICSE)
Digital Object Identifier: 10.1145/1985793.1986008

Directed Test Suite Augmentation

Zhihong Xu
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska 68588-0115, U.S.A.
zxu@cse.unl.edu
http://cse.unl.edu/~zxu

ABSTRACT

Test suite augmentation techniques are used in regression testing to identify code elements affected by changes and to generate test cases to cover those elements. Whereas methods and techniques to find affected elements have been extensively researched in regression testing, how to generate new test cases to cover these elements cost-effectively has rarely been studied. It is known that generating test cases is very expensive, so we want to focus on this second step. We believe that reusing existing test cases will help us achieve this task. This research intends to provide a framework for test suite augmentation techniques that will reuse existing test cases to automatically generate new test cases to cover as many affected elements as possible cost-effectively.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Regression testing, test suite augmentation, empirical studies

1. INTRODUCTION

Software engineers use regression testing to validate software as it evolves. To do this cost-effectively, they often begin by running existing test cases. Existing test cases, however, may not be adequate to validate the code or system behaviors that are present in a new version of a system. *Test suite augmentation techniques* (e.g., [1, 18, 23]) address this problem, by identifying where new test cases are needed and then creating them.

Despite the need for test suite augmentation, most research on regression testing has focused instead on running existing test cases. There has been research on approaches for *identifying affected elements* (code components potentially affected by changes) (e.g., [1, 16, 18]), but these approaches do not then generate test cases, leaving that task to engineers. There has been research on automatically generating test cases given pre-supplied coverage goals (e.g., [7, 19]), but this research has not attempted to integrate the test case generation task with reuse of existing test cases.

In principle, any test case generation technique could be used to generate test cases for a modified program. We believe, however, that test case generation techniques that leverage existing test cases hold the greatest promise where test suite augmentation is concerned. This is because existing test cases provide a rich source of data on potential inputs and code reachability, and existing test cases are naturally available as a starting point in the regression testing context. Further, recent research on test case generation has resulted in techniques that rely on dynamic test execution, and such techniques can naturally leverage existing test cases.

Given the foregoing discussion, our research has an overall goal of providing a framework for test suite augmentation that supports this task cost-effectively for different kinds of programs. It will represent a set of techniques that will not only integrate test case generation techniques with existing test cases, but also consider important factors that affect the cost-effectiveness of the augmentation process.

It is important to investigate our approach on different types of programs since program characteristics may impact how well various techniques work. Therefore a major element of our work will be empirical investigation of augmentation techniques on real software systems. We believe the results will offer useful suggestions for practical use. This research will also offer incentives for researchers who work on test case generation techniques to consider reusing test cases to improve these techniques themselves.

2. BACKGROUND AND RELATED WORK

2.1 Test Suite Augmentation

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, and a wide variety of approaches have been developed for rendering such reuse more cost-effective via regression test selection (RTS) techniques (e.g., [13, 17]) and test case prioritization techniques (e.g., [8]).

Test suite augmentation techniques, in contrast, are not concerned with reuse of T. Rather, they are concerned with the tasks of (1) *identifying affected elements* (portions of P' or its specification for which new test cases are needed), and

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05

then (2) *creating or guiding the creation of test cases that exercise these elements.*

Various algorithms have been proposed for identifying affected elements in software systems following changes. Some of these [3] operate on levels above the code such as on models or specifications, but most operate at the level of code, and in this paper we focus on these. Code level techniques [16] use various analyses, such as slicing on program dependence graphs, to select existing test cases that should be re-executed, while also identifying portions of the code that are related to changes and should be tested. However, these approaches do not provide methods for generating actual test cases to cover the identified code.

Some recent papers [1, 14, 18, 15] specifically address test suite augmentation. Two of these [1, 18] present an approach that combines dependence analysis and symbolic execution to identify chains of data and control dependencies that, if tested, are likely to exercise the effects of changes. A potential advantage of this approach is a fine-grained identification of affected elements; however, the papers present no specific algorithms for generating test cases. A third paper [14] presents an approach to program differencing using symbolic execution that can be used to identify affected elements more precisely than [1, 18], and yields constraints that can be input to a solver to generate test cases for those requirements. However, this approach is not integrated with reuse of existing test cases. In [15], dynamic symbolic execution (also called concolic testing) is used to generate a test input to address a single change. To some extent this approach reuses test cases to generate new ones, but in this scenario only one change in a program is considered.

2.2 Test Case Generation

While in practice test cases are often generated manually, there has been a great deal of research on techniques for automated test case generation. For example, there has been work on generating test cases from specifications, from formal models and by random or quasi-random selection of inputs (e.g., [5, 20]).

In this work we focus on code-based test case generation techniques, many of which have been investigated in prior work. Among these, several techniques (e.g., [6, 10]) use symbolic execution to find the constraints, in terms of input variables, that must be satisfied in order to execute a target path, and attempt to solve this system of constraints to obtain a test case for that path.

While the foregoing test case generation techniques are static, other techniques make use of dynamic information. Execution-oriented techniques [11] incorporate dynamic execution information into the search for inputs, using function minimization to solve subgoals that contribute toward an intended coverage goal. Goal-oriented techniques [9] also use function minimization to solve subgoals leading toward an intended coverage goal; however, they focus on the final goal rather than on a specific path.

There is another kind of dynamic test generation techniques that uses evolutionary or search-based approaches (e.g., [2, 7, 12]) such as genetic algorithms, tabu search, and simulated annealing to generate test cases. Other work [4, 19] combines concrete and symbolic test execution to generate test inputs. This second approach is known as *concolic testing* or *dynamic symbolic execution*, and has proven useful for generating test cases for C and Java programs.

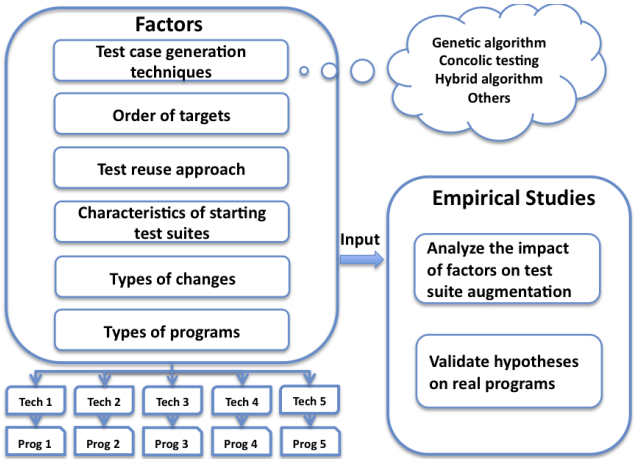

Figure 1: Overview of research

3. GOALS AND APPROACHES

The overall goal of this research is to provide a framework for test suite augmentation techniques that supports the augmentation process cost-effectively. Figure 1 provides an overview of the research. From the research we will be able to select the proper factors to use or tune for particular programs.

3.1 Activities

To achieve this goal, we are performing the following four activities, which are mapped to the boxes in Figure 1. After enumerating these activities, we elaborate on how we expect to complete each activity.

1. Identify factors that could affect the augmentation process
2. Evaluate the impacts of all factors on different test case generation techniques
3. Improve the existing techniques and develop new techniques by considering the impact of different factors
4. Empirically study techniques on real programs

For the first activity, the most important factor is the test generation techniques. There are many test case generation techniques that we could consider, but the techniques we will use are dynamic ones that could leverage exiting test cases, such as genetic algorithms and concolic testing. We will review existing literature on test case generation techniques, and choose appropriate ones to study. This corresponds to the first box "Test case generation techniques" in Figure 1. For the other factors, we will study programs to identify the factors related to the program structure that may affect the augmentation process, such as the order of the targets in the program's flow graph, and the types of changes that occur during evolution. We also need to study test suites to identify the factors related to the existing test suites that may affect the augmentation process. These include factors such as variance in coverage and diversity. Factors of both types may impact the cost-effectiveness of the whole process. This corresponds to other boxes in the "Factors" portion of Figure 1.

After identifying techniques and factors, we will investigate them empirically. Since different techniques have different attributes, we expect the identified factors to have

different impacts on the techniques, that impact the whole process. In order to achieve this, we will consider all meaningful and necessary combinations of techniques and factors and evaluate the combinations on several types of object programs. In our controlled experiments, we want to choose some programs with a lot of test cases that can help form different initial test suites that we can use to study the related factors. As independent variables we will use combinations of techniques and factors. As dependent variables we will use measures of efficiency (e.g., execution time) and effectiveness (e.g., structural coverage or fault detection effectiveness). The results may not only provide suggestions for practical use, but also give us some insights toward the third activity. This corresponds to the "Analyze the impact of factors on test suite augmentation" box in the "Empirical Studies" portion of Figure 1.

By looking at the results from Activity 2, we may find that different techniques have different strengths under particular conditions. For different programs and different types of changes, we will be able to find the best way to tune the techniques and apply them on those programs cost-effectively. Also, we may find ways to combine these techniques to fully take advantage of their strengths and achieve the best performance. In this sense, we will create some new techniques including the consideration of the factors. This corresponds to the "Test case generation techniques" box in "Factor" portion and the cloud of Figure 1.

We will apply these techniques on a selection of non-trivial programs. This corresponds to the "Validate hypotheses on real programs" of "Empirical studies" section of Figure 1.

Finally by looking at the results from our empirical studies, we will be able to select cost-effective techniques for different programs for test suite augmentation, which is the goal of the research.

3.2 Scope

The research just defined will be scoped in a number of ways to make its completion feasible in a reasonable amount of time.

We will limit the number of techniques and number of factors that will be investigated in the first two activities. Adjustments to these activities may be made as we acquire a better understanding of adapting, creating, and empirically evaluating techniques; however, at this time, we expect to identify a minimum of two techniques in Activity 1. We also expect that a minimum of three factors will be identified to be influential on test suite augmentation processes. Since we have limited number of techniques and factors, there are six combinations to be evaluated in Activity 2. We also expect that at least two hybrid techniques will be developed in Activity 3. We will validate our hypotheses on at least two non-trivial programs in Activity 4.

4. PRELIMINARY WORK

We have completed preliminary work towards each of the first two activities listed in Section 3. We briefly discuss that work here.

4.1 Applicable Test Case Generation Techniques

We have identified two techniques: genetic algorithms and concolic testing, which are appropriate for our study purpose, since both are dynamic techniques and also can use existing test cases to generate more new test cases. Reusing test cases [24] when using genetic algorithm has been applied to duplicate the coverage of an existing test suite that is not for regression testing, so we are the first to consider reusing test cases in genetic algorithm to improve the coverage of the old test suite on the new version of the program. Concolic testing usually starts with a random test case and works with only one test case at a time. We are also the first to consider reusing test cases in concolic testing to improve the coverage of an existing test suite on the new version of the program. We have used concolic testing for augmentation in a simple way [23] and found it was more effective and more efficient than using concolic testing on the new program from scratch.

4.2 Factors

We have also identified several factors that could influence the test suite augmentation process by affecting the techniques we mentioned above. The first factor is the manner in which existing test cases and newly generated test cases are reused. Since we begin augmentation with existing test cases and then we generate some new test cases during the process, picking the right subset for each technique to use is important, since having more test cases to use may improve the effectiveness, but at that same time may cost more. The second factor is the order in which the targets are considered. When we work on covering some targets we may incidentally cover other targets or we may generate some useful test cases for use on later targets. In both cases, the right order can help improve the performance. The third factor is the characteristics of the initial test suites. If the initial test suites have higher coverage on the new version of the program, we will require less effort to augment them. Also different sizes and diversity of the initial test suites will bring the test generation techniques different power in augmentation and then affect the cost-effectiveness.

4.3 Evaluation

To evaluate the impact of the factors identified above on the test case generation techniques and the differences between the two test case generation techniques in the test suite augmentation context, we conducted two empirical studies. In the first study, we used a genetic algorithm and examined only one factor, on a single subject [21]. In this study, we found that different methods of reusing test cases resulted in different coverage and cost, which meant the test reuse approach had impact on the cost-effectiveness of test suite augmentation when we use genetic algorithm. To complete our study, we conducted a more thorough study considering two techniques and the factors mentioned above together [22]. In the second study, we used two test reuse approaches: old, which is only using the test cases in the existing test suite, and old plus new, which is using not only the test cases in the existing test suite but also newly generated test cases from working on previous targets, and two different orders: depth first ordering, which tries to consider a target only after its predecessors have been considered, and random order. We evaluate their impact on the two different techniques mentioned above.

The two factors we considered had different impacts on the two test generation techniques. The test reuse approach had more impact on concolic testing. For both techniques, old plus new cost more, but improved the effectiveness of

only concolic testing. Order of targets had more impact on the genetic algorithm. The order had no impact on effectiveness for either technique since we aimed to cover all targets anyway, but it improved the efficiency of the genetic algorithm. If these results generalize, we would suggest that in practice we should use depth first ordering and old test cases when we use a genetic algorithm for augmentation while we should use random order and old plus new test cases when we use concolic testing. From our study, there are implications for researchers. Empiricists need to specify target order and test reuse approach. Additionally, research into other orders and approaches may be useful.

5. EXPECTED CONTRIBUTIONS AND RESEARCH PLAN

Through the activities described in Section 3, this research is expected to make the following contributions:

1. Bring the notion of test reuse into test suite augmentation for regression testing.
2. Identify several factors which could impact the cost-effectiveness of the augmentation process.
3. Give researchers new insights into test suite augmentation.
4. Develop a framework for test suite augmentation techniques to enable them to work effectively and efficiently.

In the future, we are going to complete the activities listed in Section 3.1. First, we are going to experiment on additional larger applications to check if the results we have now are generalizable. Second, we will also study the other factors more closely, such as types of programs. Third, we are going to improve the existing techniques by considering the factors using the results from empirical studies. Fourth, we will investigate various programs to determine the strengths and weaknesses of techniques and find ways to combine them to build up some hybrid techniques. Fifth, we will apply these techniques on some real programs to see how they work. Finally, a framework for test case generation techniques will be provided to help engineers choose techniques appropriate for different programs.

6. REFERENCES

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Test.: Acad. Ind. Conf. Pract. Res. Techn.*, pages 137–146, Aug. 2006.

[2] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *Proc. Int'l. Symp. Softw. Test. Anal.*, July 2004.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proc. Conf. Comp. Comm. Sec.*, pages 322–335, Oct 2006.

[5] T. Y. Chen and R. Merkel. Quasi-random testing. *IEEE Trans. Rel.*, 56(3):562–568, 2007.

[6] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, Sept. 1976.

[7] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado. A tabu search algorithm for structural software testing. *Comp. Op. Res.*, 35(10):3052–3072, 2008.

[8] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[9] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Meth.*, 5(1):63–86, Jan. 1996.

[10] A. Gotlieb, B. Botella, and M. Reuher. Automatic test data generation using constraint solving techniques. In *Proc. Int'l. Symp. Softw. Test. Anal.*, Mar. 1998.

[11] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–897, Aug. 1990.

[12] C. Michael, G. McGraw, and M. Shatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, Dec. 2001.

[13] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. Int'l. Symp. Found. Softw. Eng.*, Nov. 2004.

[14] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. Int'l. Symp. Found. Softw. Eng.*, Nov. 2008.

[15] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Proc. Int'l Conf. on Auto. Soft. Eng.*, 2010.

[16] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. Int'l Symp. Softw. Test. Anal.*, 1994.

[17] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, Apr. 1997.

[18] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. Int'l Conf. Auto. Softw. Eng.*, Sept. 2008.

[19] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. Int'l Symp. Found. Softw. Eng.*, pages 263–272, Sept. 2005.

[20] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java Pathfinder. In *Proc. Int'l Symp. Softw. Test. Anal.*, pages 97–107, July 2004.

[21] Z. Xu, M. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Gen. Evol. Comp. Conf*, July 2010.

[22] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proc. Int's Symp. Found.of Softw. Eng.*, Nov. 2010.

[23] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proc. Asia-Pacific Softw. Eng. Conf.*, Dec. 2009.

[24] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proc. Int'l. Conf. Softw. Test. Anal.*, pages 140–150, July 2007.