Analysis and Design of Test Case Prioritization Technique for Regression Testing

Amita
Research Scholar
Department of Computer Science and Engineering
Ganga Institute of Technology & Management, Kablana

Kamal Kumar Ranga
Assistant Professor
Department of Computer Science and Engineering
Ganga Institute of Technology & Management, Kablana

Abstract

Regression testing is a common and necessary task carried out by software practitioners to validate the quality of evolving software systems. Unfortunately, regression testing is often an expensive, time-consuming process, particularly when applied to large software systems. Consequently, practitioners may wish to prioritize the test cases in their regression test suites to execute more important test cases earlier in the regression testing process. One common goal of this test case prioritization is to increase a test suite's rate of fault detection, a measure of how quickly faults are detected by a test suite. Recent research has begun to incorporate the practical issues of varying test costs, test criticalities and fault severities into test case prioritization techniques and metrics measuring the rate of fault detection of these techniques. However, more empirical data is needed to sufficiently examine both the metrics and techniques developed by this research. To address this need, we conducted a case study to further examine the effectiveness of various prioritization techniques that incorporate varying test costs and criticalities. Software practitioners can use the results of this study to help enhance the effectiveness of their regression testing procedures and, in doing so, improve the quality of their software.

Keywords: APFD, Proritization Techniques, RTS, Cost-Cognizant

I. INTRODUCTION

The software industry employs a large amount of resources in testing software. In many cases, this testing is ad-hoc, expensive and time-consuming. A common consequence is unreliable software that is difficult to maintain, arduous to modify and comes at an unnecessarily high cost.

One widespread testing strategy employed by software engineers is to group the individual test cases for a particular software system into a set of test cases called a test suite. This test suite, or a portion of the suite, is run during the development of the software for the purpose of detecting faults within the system. A fault is an observed deviation from the specified functionality of a software system.

Often times, a developer will make changes to a software system after previously testing, and possibly releasing, that system. At this time, the developer is faced with the possibility that his or her changes will unfavorably affect the system, causing it to behave erroneously in cases where it formerly did not. We call such an occurrence a regression fault. Regression testing is a process software engineers carry out for, among other things, the purpose of detecting regression faults. Regression testing is a widespread process within the software industry because during the development process, as changes are made, several versions of a software system may be internally released among developers and after the development process, many Upgrades or patches to a system may be released to consumers. Upgrades to a system add new features that were not included in previous releases of the system, while patches correct faults in the software system that were either not discovered or not resolved during the system's development.

There are many ways to implement regression testing. A developer could simply rerun all the test cases that were run before the modification of the previous version of the software system; this approach is often referred to as the retest-all approach. However, this may involve unnecessary work when changes affect only a portion of a system. Furthermore, some test suites are large and can take days or weeks to fully execute. For example, Elbaum et al. report that one industrial test suite of a system of about 20,000 lines of source code requires a full seven weeks to run.

At times, therefore, it is desirable to run only a subset of the test suite, which can be chosen by using regression test selection techniques or permanently reduce the number of test cases by eliminating redundant test cases from the test suite, which can be accomplished by using test suite reduction techniques or prioritize the test cases within the test suite in a particular order to maximize some goal. Test case prioritization techniques are often used to implement this latter strategy by ensuring that those test cases that best accomplish this goal are executed early in the regression testing process. Potential goals of prioritization techniques can range from detecting faults as early as possible during the testing process (maximizing rate of fault detection) to exercising certain subsets of the system based on their frequency of use or previous history of failure, or achieving code coverage as quickly as possible. It is these prioritization techniques that attempt to maximize rate of fault detection that we focus on in this thesis.

The most commonly cited goal of test case prioritization is to increase a test suite's rate of fault detection. "Fault detection", however, is a relative term because certain faults in a software system can have a greater severity than other faults. For example, a fault that blocks system execution may (all other things being equal) be considered much more severe than a simple cosmetic fault in the system's appearance to the user. Using a different metric, a fault in software functionality that a user makes use of multiple times each day might be considered more severe than a fault in functionality that a user exercises once a month. Furthermore, some test cases have greater costs than other test cases. A test case that requires one hour to fully execute may (again, all other things being equal) be considered much more expensive to run than a test case that requires one minute to execute. Finally, some test cases may be intrinsically more critical to execute than other test cases. A test that exercises a software system operation commonly used by software users may be considered more critical than a test that exercises an operation that users rarely utilize.

To capture such factors, while providing a way to compare test orders for relative cost-effectiveness, a metric is needed. Elbaum et al. present such a "cost-cognizant" metric, APFDC, which can be used to measure rate of fault detection by numerically rewarding (prioritized) test case orders, taking into account variance in fault costs, test costs and test criticalities. This cost-cognizant metric, however, requires that prioritization techniques be adapted to consider the cost and criticality (determined by the performance goal the technique is attempting to meet) of test cases when prioritizing a test suite. Some cost-cognizant prioritization techniques are suggested in .

II. LITERATURE SURVEY

In year 1997, Wong, W.E. ; Bellcore, Morristown, NJ, USA ; Horgan, J.R. ; London, S. ; Agrawal, H. perfomed a work," A study of effective regression testing in practice " The purpose of regression testing is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change. Regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the software.

In year 2001, Rothermel,G. performed a work, "Prioritizing test cases for regression testing" Test case prioritization techniques schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal. Various goals are possible; one involves rate of fault detection, a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during testing can provide faster feedback on the system under test and let software engineers begin correcting faults earlier than might otherwise be possible.

In year 2002, Jung-Min Kim, performed a work, "A history-based test prioritization technique for regression testing in resource constrained environments" Regression testing is an expensive and frequently executed maintenance process used to revalidate modified software. To improve it, regression test selection (RTS) techniques strive to lower costs without overly reducing effectiveness by carefully selecting a subset of the test suite. Under certain conditions, some can even guarantee that the selected test cases perform no worse than the original test suite. This ignores certain software development realities such as resource and time constraints that may prevent using RTS techniques as intended (e.g., regression testing must be done overnight, but RTS selection returns two days' worth of tests).

In year 2004, Sebastian Elbaum, performed a work, "Selecting a Cost-Effective Test Case Prioritization Techniques" Regression testing is an expensive testing process used to validate modified software and detect whether new faults have been introduced into previously tested code. To reduce the cost of regression testing, software testers may prioritize their test cases so that those which are more important, by some measure, are run earlier in the regression testing process. One goal of prioritization is to increase a test suite's rate of fault detection.

In year2007, Cohen, M.B ; Woolf,K.M, " Combinatoral Interaction Regression Testing: A Study of Test Case Generation and Prioritization" Regression testing is an expensive part of the software maintenance process. Effective regression testing techniques select and order (or prioritize) test cases between successive releases of a program. However, selection and prioritization are dependent on the quality of the initial test suite. An effective and cost efficient test generation technique is combinatorial interaction testing, CIT, which systematically samples all t-way combinations of input parameters.

III. THE TEST CASE PRIORITIZATION

Before describing the software system with which this case study was performed, we provide details regarding the test case prioritization problem, the APFDC metric and the prioritization techniques used in this case study.

A. The Test Case Prioritization Problem

Rothermel et al. formally define the test case prioritization problem:
Given: T, a test suite, PT, the set of permutations of T, and f , a function from PT to the real numbers.
Problem: Find T' E PT such that (VT") (T" E PT) (T" T') [f(T') > f (T")].

To elaborate on this definition, PT represents the set of all possible prioritizations (orders) of the test suite T, and f is a function that takes a test suite and generates an award value based on that test suite's ability to meet some performance goal, such as rate of fault detection. The definition assumes that high award values are more desirable than lower award values.

As stated earlier, test case prioritization techniques are usually developed to maximize a test suite's ability to meet some performance goal. However, there exist various performance goals a software quality engineer may wish to meet when prioritizing a test suite for regression testing. Some of these possible goals are qualitatively described by Rothermel et al. :

- Software engineers may wish to increase the probability of revealing the faults in a software system as early in the regression testing process as possible. This is referred to as a test suite's rate of fault detection, and, in practice, is a common goal of prioritization, as detecting faults is typically considered the primary purpose of testing a software system.
- Software engineers may wish to cover as great a percentage of a software system's (coverable) source code as possible in the shortest amount of time. This is referred to as a test suite's rate of code coverage. Increasing the rate of code coverage of a software system is a common goal in practice because software engineers may be required by regulating agencies to use testing to cover a certain percentage of a software system's source code. Consequently, it may be desirable to accomplish this task as quickly as possible.
- Software engineers may wish to use testing to increase their confidence in the reliability of the software system at as fast a rate as possible. This goal might be accomplished, for example, by using testing to exercise those features of a system that have tended to fail in the past early in the regression testing process.
- Software engineers may wish to detect high-risk faults, i.e. those faults that a software user is most likely to encounter, or that are the most safety-critical, as early in the testing process as possible so that these faults can be quickly resolved.
- Software engineers may wish to execute modified source code, or specific sections of modified code, early in the regression testing process, therefore attempting to detect faults related to these code changes as quickly as possible.

In order to effectively judge a prioritization technique's success in meeting any performance goal, however, it is necessary to provide a quantitative means of measuring that technique's effectiveness. In the definition of the test case prioritization problem provided above, this quantification takes the form of a function f. In this thesis, that quantification is the APFDC metric.

IV. RESEARCH METHODOLOGY

Previous work has introduced both new metrics and techniques for performing cost-cognizant test case prioritization. However, while laid the groundwork for cost-cognizant test case prioritization, more empirical data is necessary to sufficiently examine the techniques introduced in that work. Furthermore, only investigates three cost-cognizant prioritization techniques; we would like to gather empirical data concerning the effectiveness of additional cost-cognizant techniques for regression testing.

A. Methodology

After preparing our test subject for the case study and creating the necessary additional tools required to obtain the cost-cognizant data and fault information for the subject, our next step was to create prioritized suites using the cost-cognizant techniques described in Section 2.3. For the randomized prioritization technique, however, to control for the influence of variance, we created 20 suites per version.

Having obtained these suites, our remaining task becomes one of calculating APFDC values. We therefore used the fault information for each test case to calculate various APFDC values of each prioritized test suite for versions vl—v10 in a manner to properly address our two research questions, as follows.

1) R(21): Random Test Case Orderings versus Cost-Cognizant Prioritizations

One method of evaluating our cost-cognizant prioritization techniques is to compare these techniques to a randomized test case ordering. To perform this comparison, we measured the $APFD_C$ of the prioritized test suites for versions vl—v10. For the randomized prioritizations, we averaged the $APFD_C$ values obtained on the 20 test suites mentioned earlier. We can analyze these results to compare the performance of each prioritization technique, in terms of cost-cognizant rate of fault detection, against the average $APFD_C$ obtained for the random technique.

2) "Hard to Detect Faults": Random Orderings versus Cost-Cognizant Prioritization"

To compare the performance of our cost-cognizant prioritization techniques when detecting "hard to detect faults" , we measured the APFDC of the same test case orderings created to address RQ1, except that we considered only those faults that were detected by a certain percentage of the test cases in our test suite. We divided the faults of each version of our test subject into three categories: (1) those faults detected by only 1% of all test cases, (2) those faults detected by 0.5% of all test cases, and (3) those faults detected by 0.25% of all test cases. Using this information, we can compare the APFDC values for cost-cognizant prioritization to the average APFDC for the random test case orderings, with respect to the "hard to detect faults" of a software system.

V. RESULT

A. RQ1: Random Test Case Orderings versus Cost-Cognizant Prioritizations

As can be seen, the APFDC values of this table indicate that test suites prioritized by the cost-cognizant implementations of additional function coverage, additional fault index coverage, additional DIFF coverage and additional binary DIFF coverage hereafter referred to as the additional coverage prioritization techniques always outperform a random test case ordering. However, the cost-cognizant implementations of total function coverage, total fault index coverage, total DIFF coverage and total binary DIFF coverage hereafter referred to as the total coverage prioritization techniques do not always outperform a random test case ordering. Of particular note, all prioritized test suites from the total coverage techniques failed to outperform the randomized test suites for versions v2, v3, v4, v7, v8 and v9, and at least one total coverage technique failed to outperform the randomized test suites in vl, v6 and v10. However, there were cases, such as v5, in which every test suite prioritized from a total coverage technique outperformed, based on $APFD_C$ values, the average of the random orderings of the test suite.

B. RQ2: "Hard to Detect Faults": Random Orderings versus Cost-Cognizant Prioritizations

VI. CONCLUSIONS AND FUTURE WORK

Through this thesis, we have provided a further examination of cost-cognizant test case prioritization techniques. We explored the ability of these techniques to make the regression testing process more efficient by investigating their capability to improve the rate of fault detection that might otherwise be achieved by a random test case ordering. Our results suggest that cost-cognizant additional coverage prioritization techniques can improve a test suite's rate of fault detection, but that cost-cognizant total coverage prioritization techniques may not provide this improvement. We also explored the capabilities of cost-cognizant techniques in providing an improved rate of fault detection when the faults in a software system are very difficult to detect, where we found that cost-cognizant test case prioritization techniques generally provide a significant improvement over random test case orderings as faults grow increasingly difficult to detect.

While this study was designed to provide additional empirical evidence concerning the effectiveness of cost-cognizant prioritization techniques, it has also raised several possibilities for future research. First, our results concerning the APFDC of randomized test case orderings seem to indicate that prioritization techniques may not provide an improved rate of fault detection when cost-cognizant metrics are factored into the techniques. These findings, however, are contrary to previous research investigating the rate of fault detection of non-cost-cognizant test case prioritization techniques. Yet, since these results were gathered using a metric where fault severities are considered uniform, further research could provide additional clarity on this matter.

Second, our results indicate that cost-cognizant test case prioritization techniques yield test suites with higher rates of fault detection than random test case orderings when the faults of a software system are "hard to detect" . Furthermore, the superiority of test case prioritization techniques appears to be amplified as these faults grow increasingly difficult to detect. This is a relatively untested finding, however, and further research should be committed towards investing this issue further.

Finally, an additional strategy for investigating the effectiveness of cost-cognizant prioritization techniques might be to compare the APFDC values of the cost-cognizant adaptations of our techniques with those values of the non-cost-cognizant adaptations. A study designed similar to that of this thesis could be effective in investigating cost-cognizant techniques through this comparison, provided that non-cost-cognizant implementations of each prioritization technique are available. Conducting such a study on multiple test subjects with multiple test suites, both of varying sizes, would increase the power of such a study.

It is our hope that the results of this study will contribute to the advancement of efficient methods of regression testing through test case prioritization, and will help to improve the overall quality of future software.

REFERENCES

[1] IEEE Standards Association. Software engineering standards. In Std. 1061: Standard for Software Quality Methodology, volume 3. Institute of Electrical and Electronics Engineers, 1999 edition, 1992.
[2] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. IEEE Transactions on Software Engineering, 21(9):705-716, September 1995.
[3] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. Philosophy for software measurement. J. Sys. Software, 12(3):277—281,1990.
[4] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In Proceedings of the 21st International Conference on Software Engineering, pages 345-354, May 1999.
[5] B. Bruegge and A. H. Dutoit. Object-Oriented Software Engineering: Conquering Complex and Changing Systems. Prentice Hall, Upper Saddle River, New Jersey, 2000.
[6] T. Chen and M. Lau. Dividing strategies for the optimization of a test suite. Info. Proc. Let., 60(3):135-141, March 1996.
[7] Y. Chen, D. Rosenblum, and K. Vo. Testtube: A system for selective regression testing. In Proc. Int'l Symp. Softw. Testing and Analysis, pages 102-112, August 2000.
[8] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In Proceedings of the International Symposium on Software Testing and Analysis, pages 102-112, August 2000.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In Proceedings of the 23rd International Conference on Software Engineering, pages 329-338, May 2001.
[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, 28(2):159-182, February 2002.
[11] S. G. Elbaum and J. C. Munson. A standard for the measurement of C complexity attributes. Technical Report TR-CS-98-02, University of Idaho, February 1998.
[12] S. G. Elbaum and J. C. Munson. Code churn: A measure for estimating the impact of code changes. In Proceedings of the International Conference on Software Maintenance, pages 24-31, November 1998.
[13] S. G. Elbaum and J. C. Munson. Software evolution and the code fault introduction process. 4(3):241-262, September 1999.
[14] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In Proceedings of the 20th International Conference on Software Engineering, pages 188-197, April 1998.
[15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology, 2(3):270-285, July 1993.
[16] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, Ohio State University, March 1997.
[17] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In Proceedings of the International Symposium on Software Testing and Analysis, pages 135145, August 2000.
[18] R. A. Johnson and D. W. Wichorn. Applied Multivariate Analysis. Prentice Hall, Englewood Cliffs, New Jersey, 3rd edition, 1992.
[19] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using complexity metrics. J. on Selected Areas in Comm., 8(2):253-261, February 1990.
[20] H. Leung and L. White. Insights into regression testing. In Proc. Conf Softw. Maint., pages 60-69, October 1989.
[21] J. C. Munson. Software measurement: Problems and practice. Annals of Software Engineering, 1(1):255-285, 1995.
[22] J. Musa. Software Reliability Engineering. McGraw-Hill, New York, N.Y., 1999.
[23] J. D. Musa, A. Iannino, and K. Okumoto. Software Reliability, Measurement, Prediction, Application. McGraw-Hill, New York, 1987.
[24] A. P. Nikora and J. C. Munson. Software evolution and the fault process. In Proceedings of the 23rd Annual Software Engineering Workshop. NASA/Goddard Space Flight Center, 1998.
[25] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In Proceedings of the Twelfth International Conference on Testing Computer Software, pages 111-123, June 1995.
[26] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. Communications of the ACM, 41(5):81-86, May 1988.
[27] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), June 1988.
[28] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In Proceedings of the 24th International Conference on Software Engineering, May 2002 (to appear).
[29] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology, 6(2):173210, April 1997.