Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1324-1329
DOI: 10.23940/ijpe.18.06.p24.13241329

Test Suite Augmentation via Integrating Black- and White-Box Testing Techniques

Zhiyi Zhang[a,b,*] and Ju Qian[a]

[a]*Collage of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China*
[b]*Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, 210016, China*

Abstract

Test suite augmentation is an important technique for quality assurance of evolved software. In rapid evolution of software development, engineers usually combine black-box and white-box test suites to facilitate understanding and management. This paper presents a research proposal on test suite augmentation via integrating black- and white-box testing techniques. There are two directions of our test suite augmentation. One direction is from black-box to white-box, augmenting functional test suites to satisfy structure coverage criteria. The other direction is from white-box to black-box, augmenting coverage test suite to satisfy functional requirements. A series of evaluation methods is proposed to verify the effectiveness of our augmentation approaches.

Keywords: black-box testing; white-box testing; test suite augmentation; fault detection capability; application scenario

(Submitted on February 27, 2018; Revised on April 15, 2018; Accepted on May 16, 2018)

1. Introduction

For evolved software, existing test suites may not be adequate to validate new test requirements. Test suite augmentation techniques generate new test cases using existing test information to cover new or modified parts of software [10]. Most of the work on test suite augmentation leaves the recent progress of test data generation based on source code [14]. The augmented test suite is designed to satisfy certain coverage criteria [12], such as branch coverage. This type of testing techniques is a kind of white-box.

Developers always use white-box testing to obtain the confidence for the quality of software. White-box testing uses coverage criteria as a proxy method to measure the quality of software [9]. It targets source code to verify whether it fulfills program logic. On the other hand, black-box testing is a widely used approach in which test data is often derived from software requirements [3]. It targets meaningful application scenarios and can exercise requirements very well. Black-box testing pays little attention on code implementation. White-box test data generation is easily automatic because of the great progress on dynamic symbolic execution and search-based strategies. Black-box test cases derived from functional requirements are easily maintained because they are usually generated by humans with domain knowledge. It is natural to combine these two kinds of testing techniques because of their complementary features.

Due to fierce competition, the development cycle is rapidly iterative and raises new requirements for quality assurance. A new trend is to combine black-box and white-box test suites to facilitate understanding and management, as well as to reduce the cost of maintenance [1,2,4]. However, there are some gaps between black-box and white-box, which make it difficult to combine existing test suites. This motivates us to bridge the gaps by test suite augmentation. That is, new test cases are augmented as far as possible to satisfy both white-box and black-box requirements, resulting in a holistic test suite in the development cycle.

* Corresponding author.
E-mail address: zyzhang10@nuaa.edu.cn

In summary, the research question of this paper is the following:

- How to augment test suite via integrating black- and white- box testing techniques?

It is expected to obtain a holistic test suite with advantages of both black-box and white-box testing. The test suite could satisfy software requirements and be meaningful for given application scenarios due to its black-box techniques. The test suite can cover program structure and have high fault detection capability due to its white-box techniques.

In this paper, we propose a systematic approach on test suite augmentation via integrating black-box and white-box test suite to improve software testing for rapidly evolved software. Our research includes two directions. One direction is to augment a black-box test suite to satisfy white-box coverage. The basic idea is to augment a new test case with a similar execution path but exercise different obligations. We assume that similar execution paths represent similar functional requirements. The other direction is to augment white-box test suite to satisfy certain software requirements. The features (such as keywords) of functional requirements will be extracted and will be mapped to program entities to guide the augmentation of white-box test suite.

The main contributions of our research are twofold. (1) We propose a new and systematic approach to augment test suite to achieve high fault detection capability and be meaningful for given application scenarios. (2) We propose a series of evaluation methods to show the effectiveness of our augmentation approach.

2. Related work

2.1. Augmentation Technique

Test suite augmentation uses existing test information to generate new test cases. The new test cases are expected to exercise the modification parts of software or display the change of software execution behavior.

Santelices et al. augmented test suite by combining symbolic execution techniques with a control flow graph or data flow graph [10] and proposed two steps of test suite augmentation [11]: (1) find the behavior of the new software version that is uncovered by the existing test suite; (2) generate the new test cases for the untested behavior. Xu et al. used concolic testing to augment a new test suite [12] and compared it with genetic algorithms during test suite augmentation [12]. They used branch coverage as the measurement. Jamrozik et al. proposed an approach to generate test suites with augmented dynamic symbolic execution [15]. They claimed that DSE usually generates single and simple values for path conditions, which may result in losing some important errors. So, they augmented the path condition by adding boundary values, mutation operators or logic coverage obligations into path conditions, and they produced the solve answer for different values.

Yang et al. used static information to calculate the changes of evolutionary software and generated new path conditions to augment test suite [6,16]. Fraser et al. combined the search algorithm and mutation technology to augment test suite [13]. The experimental results showed that the test suite generated by their method could achieve maximum code coverage with the smallest size. Xu et al. performed a comparative analysis of the role of genetic algorithms and concolic testing in test suite augmentation [17]. They found that test cases augmented using concolic testing are more effective, while using genetic algorithms are more flexible. Yoshida et al. proposed a novel technique for automated and fine-grained incremental generation of unit tests through minimal augmentation of an existing test suite [5]. They analyzed dynamic dependence variables using Reduced Ordered Binary Decision Diagrams. Chen et al. augmented test suite using the data method to cross-combine the most frequently occurring variables based on the coverage criteria [18].

All augmentation techniques described above are restricted to white-box testing. Our augmentation is integrating black-box and white-box. The augmented test cases are expected to satisfy certain functional requirements so that they can be easily maintained and reused.

2.2. Black-Box and White-Box Testing

Gray-box testing partially uses white-box and black-box techniques with incomplete program information. Wang et al. proposed a method that generates test cases directly from a UML activity diagram by using the gray-box method [8]. They took full advantage of the black-box testing to analyze the expected external behavior and the white-box testing to cover the internal structure of software under test. Kicillof et al. used black-box testing to build a behavioral model, and used

conformance of the software under test to provide general guidance for the generation of the tests. White-box testing could generate concrete parameter values, which maximize code coverage using symbolic analysis. Hence, this method could exercise all selected paths in the behavioral mode and all possible coverage paths in the coverage strategy.

There is some research on using black-box and white-box testing directly. Beckert et al. made use of software structure information during black-box testing [2]. They used specifications as inputs for black-box testing, and turned them into white-box testing methods. Barrett et al. analyzed the program's internal and guaranteed a level of test space coverage. They refined initial coverage requirements to explore the regions in a high-dimensional test space and generated test cases that satisfied these requirements and test space coverage [1]. Beydeda et al. integrated white-box and black-box techniques for class-level regression testing [4]. They generated class specification implementation graphs, which contain control and data flow information, and then added data flow edges for black-box testing.

All the techniques described above only focus on one side of black-box or white-box. Our augmentation technique integrates black-box and white-box to obtain a holistic test suite.

3. Our approach and evaluation

Our research includes two directions. One direction is to augment a black-box test suite to satisfy white-box coverage criteria. The other direction is to augment a white-box test suite to satisfy certain software requirements.

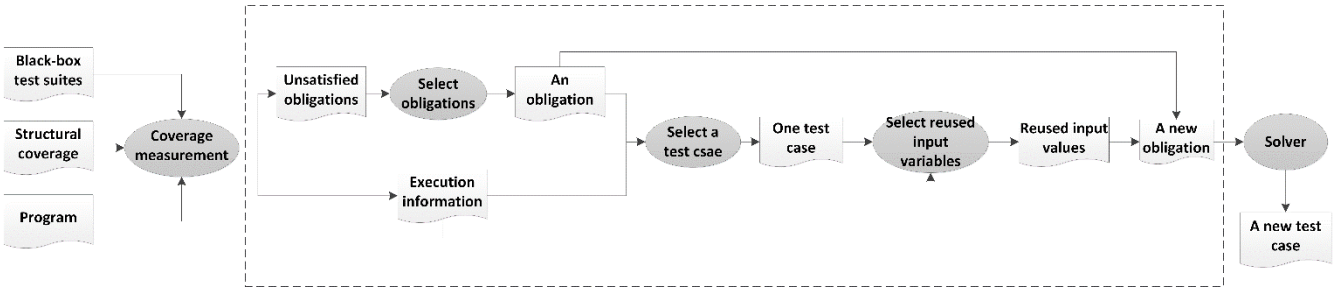

Figure 1. The process of augmentation from black-box to white-box.

3.1. Augmentation from Black-box to White-box

We have an assumption underlying our method:

- The existing test case t exercises meaningful application scenarios of the system under test. A new test case t' is augmented from t. If the execution paths of t and t' are similar, t' can also exercise similar application scenarios that are meaningful.

Figure 1 shows the process of augmentation from black-box to white-box. In summary, our method has the following five steps:

Step 1. Inputs obtaining

Our approach needs three inputs, including a good black-box test suite that is generated manually, a white-box coverage criterion that the augmented test suite should satisfy, and the program under test. We use the black-box test suite generated manually as the existing test suite because it satisfies program requirements and has meaningful application scenarios. According to the assumption mentioned above, the augmented test cases that have similar execution paths with the original black-box test cases can also satisfy software requirements and have meaningful application scenarios.

Step 2. Obligations generation.

We generate all the obligations of the program for a given coverage criterion. An obligation is a source entry in the programming language, such as high-level statements or decisions. For example, an obligation presents a reachable branch in branch coverage.

Step 3. Coverage measurement.

Each black-box test case is run on each obligation and records the execution information. The execution information includes the upper nearest branch point of the obligation and whether the test case satisfies the obligation. We use instrumentation to get this execution information. Also, we can get all unsatisfied obligations that have not been satisfied by the existing test cases.

Step 4. Selection (obligations, test cases and their input variables).

We select an unsatisfied obligation and a test case that have similar execution paths. We could select an unsatisfied obligation randomly. To select a test case that has a similar execution path, for the unsatisfied obligation, if there exists a test case that covers the same decision but not the condition as the unsatisfied obligation, we will use this test case to augment a new test case with a similar execution path. Otherwise, if there is no such test case, according to execution information, we will select a test case that covers the upper nearest branch point for the obligation and use it to augment a new test case. In the worst case, if there is not a test case that covers the same decision as the upper nearest branch point for the obligation, we will select an existing test case randomly to augment a new test case.

Step 5. Augmentation.

A new test case is augmented based on the selected test case. In our approach, we attempt to get a new test case by changing some input values of existing test cases. One method for changing values is using dynamic symbolic execution. According to the execution information, we will find out which variables in the obligation are not satisfied. Then, we select the input variables of the selected test cases that are independent of the unsatisfied variables. Changing these input values will not affect the obligation, and we then reuse the values of the independent variables. Meanwhile, we symbol the remaining input values. Next, we combine the obligation and concreate test case, and use dynamic symbolic execution to solve it. The solved test case will satisfy this obligation. Since these two test cases have similar execution paths, according to the assumption, we believe the new test case will meet similar program requirements and have meaningful application scenarios with the selected test case. If there is no solution, select another test case and augment it again. Repeat steps 4 and 5 until all unsatisfied obligations have been satisfied by at least one test case.

3.2. Augmentation from White-box to Black-box

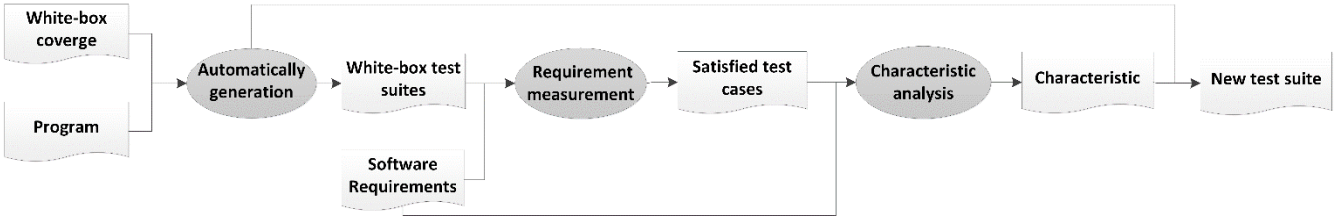

Figure 2. The process of augmentation from white-box to black-box.

Our main idea of this research is to analyze existing white-box test cases that satisfy software requirement, find their character and use them to augment new test cases. Figure 2 shows the process of augmentation from white-box to black-box. In summary, our method has the following four steps:

Step 1. White-box test suites obtaining

We select one or more coverage strategies, such as branch coverage or MC/DC. Then, for each coverage strategy, a test suite will be automatically generated to satisfy it. Also, these test cases can come from the test case pool of the previous version.

Step 2. Requirement measurement.

We build the model of black-box requirements and measure each test case by using the model to find out which software requirements have been satisfied. So, we can get the test cases that meet software requirements.

Step 3. Characteristic analysis.

We analyze these test cases that meet software requirements and discover the characteristics of these test cases, such as the potential interactions that exist between parameters and the constraints between parameters and values. After that, we can figure out some potential interactions between the software structure and requirements.

Step 4. Augmentation.

We use characteristics that are discovered in Step 3 during the generation of the white-box test suite. For example, add the potential interactions that exist between parameters when using dynamic symbolic execution so that the generated test cases can satisfy some software requirements.

From this method, we could generate a white-box test suite that meets software requirements. Moreover, these test cases can be used as existing black-box test suite in our method mentioned in Section 3.1.

3.3. Evaluation methods

We use three evaluation methods as following:

1. Coverage ratio. This is to evaluate the effectiveness in respect to the white-box and our augmented test suite.
2. Percentage of meaningful test cases for application scenarios. This is to evaluate the effectiveness in respect to the black-box and our augmented test suite.
3. Mutation score. This is to evaluate the fault detection capability of test suites.

In evaluation method 1, we calculate two aspects of coverage. One aspect is the percentage of white-box coverage for the existing black-box test suite, such as branch coverage or MC/DC. Especially, we calculate the percentage of important but uncovered functions. The other is the percentage of same coverage for our augmented test suite. We use this evaluation to prove that only using the black-box test suite cannot satisfy the test requirement of program structure.

In evaluation method 2, we calculate two aspects of percentage of meaningful test cases for application scenarios. One aspect is percentage of test cases with meaningful application scenarios in the automatically generated white-box test suite. The other is the percentage of test cases with meaningful application scenarios in the augmented test suite. We use this evaluation to prove that only using the white-box test suite cannot satisfy the test requirement of program requirements.

In evaluation method 3, we calculate the mutation score of simply combing black-box and white-box test suite, as well as the augmented test suite respectively. We use this evaluation to prove that the fault detection capability of the test suite that simply combines the black-box and white-box test suite is not strong enough, so our research is significant.

4. Conclusions

In this paper, we explore our research work about test suite augmentation by integrating black-box and white-box test suites. Our research includes two directions. One direction is from black-box to white-box, augmenting functional test suites to satisfy certain structure coverage criteria. The other direction is from white-box to black-box, augmenting coverage test suites to satisfy certain functional requirements. From this method, we could get a test suite that not only meets the software requirement, tests the program structure and implements details well, but also boasts high fault-finding and meaningful application scenarios.

Acknowledgements

The work is supported, in part, by The National Key Research and Development Program of China (Grant No. 2018YFB1003902), the Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), the Ministry of Industry and Information Technology (Grant No. XCA17007-04) and by the Science and Technology Planning Project of Jiangsu Province (Grant No. BY2016003-02).

References

1. A. Barrett and D. Dvorak, 2009, "A Combinatorial Test Suite Generator for Gray-box Testing," In *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology* (SMC-IT), pp. 387-393, Pasadena, USA, July 2009
2. B. Beckert, and C. Gladisch, "White-box Testing by Combining Deduction-based Specification Extraction and Black-box Testing," in *Proceedings of the first International Conference on Tests and Proofs* (TAP), pp. 207-216, Zurich, Switzerland, February 2007
3. B. Beizer and J. Wiley, "Black Box Testing: Techniques for Functional Testing of Software and Systems," *IEEE Software*, vol.13, no. 8, pp.98, 1996
4. S. Beydeda, and V. Gruhn, "Integrating White-and Black-box techniques for Class-level Regression Testing," In *Proceedings of 25th International Computer Software and Applications Conference* (COMPSAC), pp. 357-362, Chicago, USA, October 2001
5. W. Chen, K. Hsieh, L. Wang, and J. Bhadra, "Data Driven Test Plan Augmentation for Platform Verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 23-29, 2017
6. G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276-291, 2013
7. K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux, "Generating Test Suites with Augmented Dynamic Symbolic Execution," In *Proceedings of 7th International Conference on Tests and Proofs* (TAP), pp. 152-167, Budapest, Hungary. June 2013
8. N. Kicillof, W. Grieskamp, N. Tillmann and V. Braberman, "Achieving Both Model and Code Coverage with Automated Gray-box Testing," In *Proceedings of the 3rd international workshop on Advances in model-based testing* (A-MOST), pp. 1-11, London, United Kingdom, July 2007
9. T. Ostrand, "White-Box Testing," Encyclopedia of Software Engineering, John Wiley & Sons, Inc., 2002
10. R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite Augmentation for Evolving Software," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (ASE), pp. 218-227, L'Aquila, Italy, September 2008
11. R. Santelices, "Automated Scalable Test-suite Augmentation for Evolving Software," In *Proceedings of 31st International Conference on Software Engineering* (ICSE) Companion Volume, pp. 379-382, Vancouver, Canada, May 2009
12. Z. Xu, Y. Kim, M. Kim, G. Rothermel and M. B. Cohen, "Directed Test Suite Augmentation: Techniques and Tradeoffs," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE), pp. 257-266, Santa Fe, USA, November 2010
13. Z. Xu, Y. Kim, M. Kim, M. B. Cohen and G. Rothermel, "Directed Test Suite Augmentation: An Empirical Investigation," *Software Testing Verification & Reliability*, vol. 25, no. 2, pp. 77-114, 2015
14. Z. Xu and G. Rothermel, "Directed Test Suite Augmentation," in *Proceedings of the 16th Asia-Pacific Software Engineering Conference* (APSEC), pp. 406-413, Penang, Malaysia, November 2009
15. G. Yang, S. Khurshid, S. Person and N. Rungta, "Property Differencing for Incremental Checking," in *Proceedings of the 36th International Conference on Software Engineering* (ICSE), pp. 1059-1070, Hyderabad, India, May 2014
16. G. Yang, S. Person, N. Rungta and S. Khurshid, "Directed Incremental Symbolic Execution," *ACM Transactions on Software Engineering & Methodology*, vol. 24, no.1, article. 3, September 2014
17. H. Yoshida, S. Tokumoto, M .R. Prasad, I. Ghosh and T. Uehara, "FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs," in *Proceedings of the 25th International Symposium on Software Testing and Analysis* (ISSTA), pp. 106-117, Saarbrücken, Germany, July 2016
18. L. Wang, J. Yuan, X. Yu, J. Hu, X. Li and G. Zheng, 2004. "Generating Test Cases from UML Activity Diagram Based on Gray-box Method," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference* (APSEC), pp. 284-291, Busan, Korea, December 2004