Exploiting Synergy Between Testing and Inferred Partial Specifications

Tao Xie David Notkin
Department of Computer Science & Engineering, University of Washington
{taoxie, notkin}@cs.washington.edu

Abstract

The specifications of a program can be dynamically inferred from its executions, or equivalently, from the program plus a test suite. A deficient test suite or a subset of a sufficient test suite may not help to infer generalizable program properties. But the partial specifications inferred from the test suite constitute a summary proxy for the test execution history. When a new test is executed on the program, a violation of a previously inferred specification indicates the need for a potential test augmentation. Developers can inspect the test and the violated specification to make a decision whether to add the new test to the existing test suite after equipping the test with an oracle. By selectively augmenting the existing test suite, the quality of the inferred specifications in the next cycle can be improved while avoiding noisy data such as illegal inputs. To experiment with this approach, we integrated the use of Daikon (a dynamic invariant detection tool) and Jtest (a commercial Java unit testing tool). This paper presents several techniques to exploit the synergy between testing and inferred partial specifications in unit test data selection.

1. Introduction

Given that specifications play an important role in a variety of software engineering tasks and that the specifications are often absent from a program, dynamically inferring program specifications from its executions is a useful technique [3]. The output of the dynamic specification inference has been used to aid program evolution in general [3] and program refactoring in particular [7]. Most of the applications can achieve better results if the inferred specifications are closer to the oracle specifications. Like other dynamic analysis techniques, the dynamic specification inference is also constrained by the quality of the test suite for the program. Usually it is unlikely that the inferred properties are true over all possible executions. When properly applied, static verification tools can filter out false positives in the inferred specifications [8].

Different from previous applications that use the final inferred specifications from all the available tests, two recent approaches have begun to use the intermediate partial specifications inferred from a subset. Both are based on the fact that the inferred specifications may change when new tests are added to the existing test suite. The first, called the operational difference (OD) technique, makes use of the differences in inferred specifications between test executions to generate, augment, and minimize the test suites [5]. The second, as implemented in the tool DIDUCE, can continuously check a program's behavior against the incrementally inferred partial specifications during the run(s), and produce a report of all violations detected along the way [4]. This can help detect bugs and track down the root causes. It is noteworthy that "partial specification" also carries the denotation that the specification is not complete or accurate in terms of an oracle specification. Thus there is a convergence of the two meanings when the specifications inferred from the whole test suite are used to approximate the oracle specification.

In this research, we further exploit the synergy between testing and inferred partial specifications. All available tests in this context are a small size of the existing unit test suite plus a large size of the automatically generated unit tests. The purpose is to tackle the problem of selecting automatically generated tests to augment the existing unit test suite. Violations of the inferred partial specifications from the existing unit test suite can help this unit test data selection. Moreover, selectively augmenting the existing test suite can prevent introducing noisy data, e.g. illegal inputs, from negatively affecting the specification inference.

2. Background

The "test first" principle, as advocated by Extreme Programming (XP) development process [1], requires unit tests to be constructed and maintained before, during, and after the source code is written. Developers need to manually generate the test inputs and oracles based on the requirements in mind or in documentation. They need to decide whether enough test cases have been written to cover the features in their code thoroughly. Some commercial tools for Java unit testing, e.g. ParaSoft Jtest [10], attempt to fill the "holes" left by the execution of the manually generated unit tests. These tools can automatically generate a large number of unit tests to exercise the program. However, there are two main issues in automatic unit test generation. First, there are no test oracles for these automatically generated tests unless developers write down some formal specifications or runtime assertions [2]. Second, only a relatively small size of automatically generated tests can be added to the existing unit test suite. This is because the unit test suite needs to be maintainable, as is advocated by the XP approach [1].

Two main unit test selection methods are available. In white box testing (e.g., the residual structural coverage [11]), users select tests that provide new structural coverage unachieved by the existing test suite. In black box testing, the operational difference (OD) technique is applicable in augmenting a test suite [5]. However, the OD technique for this unit test augmentation problem might select a relatively large set of tests because the specification generator's statistical tests usually require multiple executions before outputting a specification clause. Additionally, OD requires frequent generation of specifications, and the existing dynamic specification generation is computationally expensive. Therefore, instead of using OD in the unit test selection, we adopt a specification violation approach similar to DIDUCE [4].

Our approach is implemented by integrating Daikon and Jtest. Daikon [3], a dynamic invariant detection tool, is used to infer specifications from program executions of test suites. The probability limit for justifying invariants is set by Daikon users. The probability is Daikon's estimate of how likely the invariant is to occur by chance. It ranges from 0 to 100% with a default value of 1%. Smaller values yield stronger filtering. Daikon includes a MergeESC tool, which inserts inferred specifications to the code as ESC/Java annotations [12]. ParaSoft Jtest [10], on the other hand, is a commercial Java unit testing tool, which automatically generates unit test data for a Java class. It instruments and compiles the code that contains Java Design-by-Contract (DbC) comments, then automatically checks at runtime whether the specified contracts are violated. We modified MergeESC to enable Daikon to insert the inferred specifications into the code as DbC comments. Since ESC/Java has better expressiveness than Jtest's DbC, a perl script is written to filter out the specifications whose annotations cannot be supported by Jtest's DbC. After being fed with a Java class annotated with DbC comments, Jtest uses them to automatically create and execute test cases and then verify whether a class behaves as expected. It suppresses any problems found for the test inputs that violate the preconditions of the class under test. But it still reports precondition violations for those methods called indirectly from outside the class. Note that DIDUCE tool reports all precondition violations [4]. By default, Jtest tests each method by generating arguments for them and calling them independently. In other words, Jtest basically tries the calling sequences of length 1 by default. Tool users can set the length of calling sequences in the range of 1 to 3. If a calling sequence of length 3 is specified, Jtest first tries all calling sequences of length 1 followed by all those of length 2 and 3 sequentially.

3. Specification Violation Approach

This section describes the specification violation approach. Section 3.1 introduces the basic technique of the approach. Section 3.2 presents the precondition guard removal technique to improve the effectiveness of the basic technique. Section 3.3 describes the iterative process of applying these techniques. A preliminary experiment is conducted on a Java class of the bounded stack that is used to store unique elements of integer [13]. Detailed experimental results for this example are described in [14].

3.1. Basic Technique

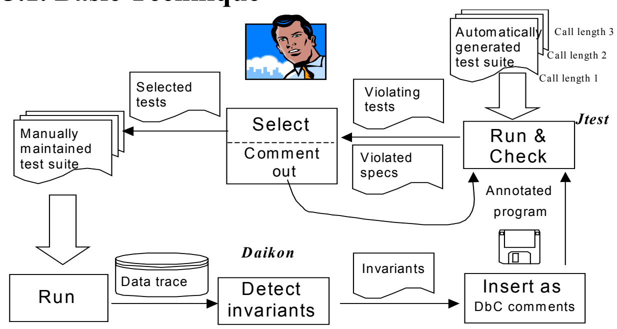

Figure 1. An overview of the basic technique

In our approach, partial specifications are inferred from program executions of the existing unit test suite by using Daikon (Figure 1). The partial specifications are inserted into the code as DbC comments. The resulting code is fed to Jtest. Initially, Jtest's calling sequence length is set to 1 and Jtest is run to automatically generate and execute test data. When a certain number of specification violations have occurred before Jtest exhausts its testing repository, it stops generating test data and reports specification violations. For each reported specification violation, i.e., the violated specification and the violating test, developers inspect them to decide whether to equip the test with an oracle and add it to the existing test suite. Then developers disable each violated specification by commenting them out and rerun Jtest repeating the above procedure until no specification violations are reported. The whole process is iteratively applied by setting the length of calling sequences as 2 and subsequently 3.

The rationale behind the basic technique is that if a new test violates an inferred partial specification, it is likely that this test exercises a new feature of the program uncovered by the existing test suite. This technique guarantees that the new test does not overlap with any others from the existing test suite in terms of the violated specification. In addition, the violating tests have a relatively high probability of exposing faults in the code if there are any. It is because that running the existing test suite on the code exhibits the normal behavior reflected by the inferred specifications and the violating tests might make the code exhibit the abnormal behavior.

The symptoms of specification violations can be that the boolean value of a specification predicate is false or exceptions are thrown. In order for the inferred specifications to be violated, we set the probability limit to be 100%. The specification violations indicate deficiencies of the existing test suite. However, some violations might not be very helpful for the unit test selection. For example, the existing test suite for the stack implementation only push the integer element of 2 or 3 into the stack and thus one of the inferred specifications is that the stack element is 2 or 3. The automatically generated tests that push the element of 1 into the stack violate this specification. Since the element of 1 is not so different than 2 or 3 for the purpose of testing this stack implementation, developers might not select the violating test to the existing test suite.

3.2. Precondition Guard Removal

In our basic technique, when the existing test suite is deficient, the inferred preconditions might be so restrictive as to filter out those legal test data inputs in Jtest test data generation and execution. This over-restrictiveness of preconditions also makes static verification of inferred specifications less effective [8]. Even if a static verifier could confirm an inferred post-condition specification given some over-restrictive preconditions, it is hard to tell whether it is generalizable to the actual preconditions.

To assure better quality of the unit under test, we need to exercise the unit under more circumstances than what is constrained by the inferred preconditions. Before the code that is annotated with DbC comments is fed to Jtest, all precondition comments are removed. In the preliminary experiment, we observed that precondition guard removal techniques reported more violations and exposed more faults than the basic technique (Section 3.1). Indeed, removing precondition guards produces more false positives by allowing some illegal inputs. Yet the tool only reports those illegal inputs that cause postcondition or invariant violations.

3.3. Iterations

After the new test augmentations using the 3.1 and 3.2 techniques, all the violating tests with legal inputs, whether selected or unselected, can be further run together with the existing ones to infer new specifications. Although those unselected violating tests with legal inputs might not exercise any interesting new features, running them in the specification inference can relax the violated specifications to reduce the false positives in the next iteration. The same process described in Section 3.1 and 3.2 is repeated until there are no specification violations or no test data selected from the violating tests. In the preliminary experiment, most of the specification violations were observed in the first iteration, and all specification violations were observed before the third iteration.

4. Effect of Inferred Specifications on Test Generation

In previous sections, we showed that the inferred specifications can be used to select unit test data and improve the specification quality. Furthermore, we observed that the inferred specifications also had an effect on Jtest's automatic test generation. As is described in Jtest's manual [6], if the code has preconditions, Jtest tries to find inputs that satisfy all of them. If the code has postconditions, Jtest creates test cases that verify whether the code satisfies these conditions. If the code has invariants, Jtest creates test cases that try to make them fail. The preliminary experiment showed that preconditions have greater impacts on Jtest's test generation than either postconditions or invariants. Sometimes Jtest, equipped with specifications, could automatically generate tests that achieve better code coverage than the one without specifications. For the test length of two, the former Jtest automatically generated more tests for the stack implementation than the latter one. It suggests that inferred specifications are able to guide Jtest to generate better tests.

5. Concluding Remarks

In sum, selecting automatically generated tests to augment the existing unit test suite is an important step in the unit testing practice. Inferred partial specifications act as a proxy for the existing test execution history. A new test that violates an inferred specification is a good candidate for developers to inspect for test data selection. The violating test also has a high probability to expose faults in the code. Instead of considering the test augmentation as a one-time phase, it should be considered as a frequent activity in software evolution, if not as frequent as regression unit testing. When a program is changed, the specifications inferred from the same unit test suite might change as well, giving rights to possible test violations. Tool-assisted unit test augmentation can be a means to evolving unit tests and assuring better unit quality. Moreover, augmenting unit test suite in a controlled way can lead to better quality of inferred specifications. In future work, we plan to apply the specification violation techniques in connecting system testing and unit testing. Specifications are to be inferred from system testing and specification violations by the generated unit tests are used to guide unit test data selection. Also, the partial specifications inferred from testing done by component providers are to be delivered as component metadata [9], which will aid component users to perform test augmentations. Finally, we plan to apply the specification violation techniques in other kinds of inferred specifications, e.g. sequencing constraints or protocols.

6. Acknowledgement

We thank Michael Ernst and the Daikon project members at MIT for their assistance in our installation and use of the Daikon tool. This work was supported in part by the National Science Foundation under grant ITR 0086003. The authors wish to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

7. References

[1] K. Beck. Extreme programming explained. Addison-Wesley, 2000.

[2] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Proceedings of 16th European Conference Object-Oriented Programming (ECOOP), 2002, pp. 231-255.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol. 27, no. 2, Feb. 2001, pp. 1-25.

[4] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In Proceedings of the International Conference on Software Engineering, May 2002, pp. 291-301.

[5] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In Proceedings of the International Conference on Software Engineering, (Portland, Oregon), May 6-8, 2003.

[6] Jtest manuals version 4.5. Parasoft Corporation, October 23, 2002.

[7] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In Proceedings of ICSM 2001, November, 2001, pp. 736-743.

[8] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In Proceedings of RV'01, First Workshop on Runtime Verification, (Paris, France), July 23, 2001.

[9] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks, In Proceedings of the 2nd International Workshop on Engineering Distributed Objects, November 2000, pp. 129-144.

[10] ParaSoft Corportation. http://www.parasoft.com/

[11] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In Proceedings of ICSE 1999, pp. 277-284.

[12] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, Palo Alto, California, Oct 12, 2000.

[13] P. D. Stotts, M. Lindsey, A. Antley. An informal formal method for systematic JUnit test case generation. XP/Agile Universe 2002, pp 131-143.

[14] T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications, University of Washington Department of Computer Science and Engineering technical report UW-CSE-03-04-02, (Seattle, WA), April 2003.