Improving Effectiveness of Automated Software Testing in the Absence of Specifications

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract

Program specifications can be valuable in improving the effectiveness of automated software testing in generating test inputs and checking test executions for correctness. Unfortunately, specifications are often absent from programs in practice. We present a framework for improving effectiveness of automated testing in the absence of specifications. The framework supports a set of related techniques, including redundant-test detection, non-redundant-test generation, test selection, test abstraction, and program-spectra comparison. The framework has been implemented and empirical results have shown that the developed techniques within the framework improve the effectiveness of automated testing by detecting high percentage of redundant tests among test inputs generated by existing tools, generating non-redundant test inputs to achieve high structural coverage, reducing inspection efforts for detecting problems in the program, and exposing behavioral differences during regression testing.

1 Introduction

To reduce the laborious human effort in software testing, developers can conduct automated software testing by using tools to automate some activities in software testing. Software testing activities typically include generating test inputs, running test inputs, and verifying test executions. Developers can use some existing frameworks or tools such as the JUnit testing framework [5] to write unit-test inputs and their expected outputs. Then the JUnit framework can automate running test inputs and verifying actual outputs against the manually written assertions. To reduce the burden of manually creating test inputs, developers can use some existing test-input generation tools [1, 3, 11] to generate test inputs automatically. After developers modify a program, they can conduct *regression testing* by rerunning the existing test inputs in order to assure that no regression faults are introduced. Even when expected outputs are not created for the existing test inputs, the actual outputs produced by the new version can be automatically compared with the ones produced by the old version in order to detect behavioral differences. There are two major challenges in automated software testing:

Generate test inputs effectively. In testing object-oriented problems, a test input typically consists of a sequence of method calls on the objects of the class; inputs for method calls consist of not only method arguments but also receiver-object states, which are sometimes *structurally complex* inputs, such as linked data structures that must satisfy complex properties. Method sequences can be generated to construct desired object states indirectly [11]; however, it is generally expensive to enumerate all possible method sequences even given a small number of argument values and a small bound on the maximum sequence length.

Verify test executions effectively. Test inputs can be automatically generated but the expected outputs for generated test inputs often cannot be generated without specifications. When no expected outputs are available, developers often rely on program crashes or uncaught exceptions [3] as symptoms for unexpected behavior; however, it is limited in exploiting these generated test inputs by verifying only whether the program crashes or throws uncaught exceptions. In regression testing, developers can compare the actual outputs of a new version of the program with the actual outputs of a previous version; however, behavioral differences between versions often cannot be propagated to the observable outputs that are compared between versions.

Although specifications can be used to improve the effectiveness of generating test inputs and checking program correctness when running test inputs without expected outputs, specifications often do not exist in practice. Our research focuses on developing a framework for improving

effectiveness of automated testing in the absence of specifications. The framework includes techniques and tools for improving the effectiveness of generating test inputs and inspecting their executions for correctness, two major challenges in automated software testing.

2 Framework

Figure 1 shows our framework for improving effectiveness of automated testing. The framework consists of two groups of components. The first group of components —the redundant-test detector [14] and non-redundant-test generator [14, 15]— addresses the issues in generating test inputs. The second group of components (the test selector [17], test abstractor [18, 19], and program-spectra comparator [20]) infers program behavior dynamically in order to address the issues in checking the correctness of test executions. The second group of components further sends feedback information to the first group to guide test generation [16].

2.1 Redundant-Test Detector

Existing unit-test-generation tools generate a large number of test inputs to exercise different sequences of method calls in the interface of the class under test. Different combinations of method calls on the class under test result in a combinatorial explosion of tests. Because of resource constraints, existing test-generation tools often generate different sequences of method calls whose lengths range from one [3] to three [11]. However, sequences of up-to-three method calls are often insufficient for detecting faults or satisfying test adequacy criteria. In fact, a large portion of these different sequences of method calls exercise no new method behavior; in other words, the tests formed by this large portion of sequences are *redundant tests*. We have defined redundant tests by using method-input values (including both argument values and receiver-object states). When the method-input values of each method call in a test have been exercised by the existing tests, the test is considered as a redundant test even if the sequence of method calls in the test is different from the one of any existing test. We have developed a redundant-test detector, which can post-process a test suite generated by existing test-generation tools and output a reduced test suite containing no redundant tests. Our approach not only presents a foundation for existing tools that generate non-redundant tests [2, 10] but also enables any other test-generation tools [3, 11] to avoid generating redundant tests by incorporating the redundant-test detection in their test generation process. Our experimental results have shown the effectiveness of the redundant-test detection tool: about 90% of the tests generated by a commercial testing tool [11] are detected and reduced by our tool as redundant tests.

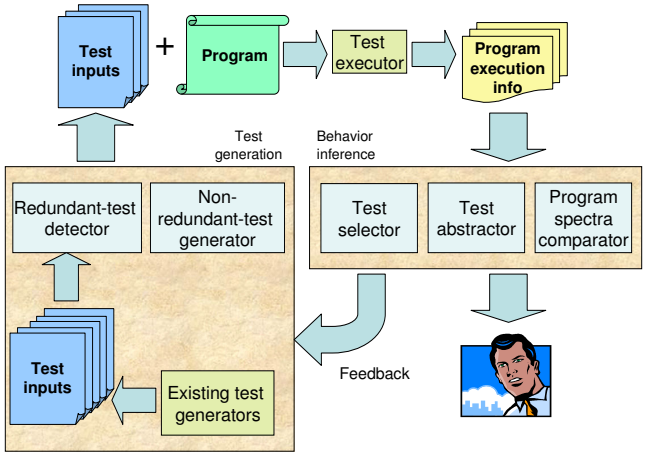

Figure 1. Framework for improving effectiveness of automated testing

2.2 Non-Redundant-Test Generator

Based on the notion of avoiding generating redundant-tests, we have developed a non-redundant-test generator, which explores the concrete or symbolic receiver-object state space by using method calls (through normal program execution or symbolic execution). The test generator based on concrete-state exploration faces the state explosion problem. Recently, symbolic execution [8] has been used to directly construct symbolic states for receiver objects [7, 13]; however, the application of symbolic execution requires the user to provide specially constructed class invariants [9]. Without requiring any class invariant, our test generator can also use symbolic execution of method sequences to explore the symbolic receiver-object states and prune this exploration based on novel state comparisons (comparing both heap representations and symbolic representations). Our extension and application of symbolic execution in state exploration not only alleviate the state explosion problem but also generate relevant method arguments for method sequences automatically by using a constraint solver. Our experimental results have shown the effectiveness of the test generation based on symbolic-state exploration: it can achieve higher branch coverage faster than the test generation based on concrete-state exploration.

2.3 Test Selector

Because it is infeasible for developers to inspect the actual outputs of a large number of generated tests, we have developed a test selector to select a small valuable subset of generated tests for inspection. These selected tests exercise new behavior that has not been exercised by the ex-

isting test suite. In particular, we use Daikon [4] to infer program behavior dynamically from the execution of the existing (manually) constructed test suite. We next feed inferred behavior in the form of specifications to an existing specification-based test-generation tool [11]. The tool generates tests to violate the inferred behavior. These violating tests are selected for inspection, because these violating tests exhibit behavior different from the behavior exhibited by the existing tests. Developers can inspect these violating tests together with the violated properties, equip these tests with expected outputs, and add them to the existing test suite. Our experimental results have shown that the selected tests have a high probability of exposing anomalous program behavior (either faults or failures) in the program.

2.4 Test Abstractor

Instead of selecting a subset of generated tests for inspection, our test abstractor summarizes and abstracts the receiver-object-state transition behavior exercised by all the generated tests. Because the concrete-state transition diagram for receiver objects is too complicated for developers to inspect, the test abstractor uses a state abstraction technique based on the observers in a class interface; these observers are the public methods whose return types are not void. An abstract state for a concrete state is represented by the concrete state's observable behavior, consisting of the return values of observer-method calls on the concrete state. The abstract states and transitions among them are used to construct succinct state transition diagrams for developers to inspect. Our evaluation has shown that the abstract-state transition diagrams can help discover anomalous behavior, debug exception-throwing behavior, and understand normal behavior in the class interface.

2.5 Program-Spectra Comparator

In regression testing, comparing the actual outputs of two program versions is limited in exposing the internal behavioral differences during the program execution, because internal behavioral differences often cannot be propagated to observable outputs. A program spectrum is used to characterize a program's behavior [12]. We propose a new class of program spectra, called *value spectra*, to enrich the existing program spectra family, which primarily includes structural spectra (such as path spectra [6, 12]). Value spectra capture internal program states during a test execution. A *deviation* is the difference between the value of a variable in a new program version and the corresponding one in an old version. We have developed a program-spectra comparator that compares the value spectra from an old version and a new version, and uses the spectra differences to detect behavior deviations in the new version. Furthermore, value spectra differences can be used to locate deviation roots, which are program locations that trigger the behavior deviations. Inspecting value spectra differences can allow developers to determine whether program changes introduce intended behavioral differences or regression faults. Our experimental results have shown that comparing value spectra can effectively expose behavioral differences between versions even when their actual outputs are the same, and value spectra differences can be used to locate deviation roots with high accuracy.

2.6 Feedback Loop

Dynamic behavior inference requires a good-quality test suite to infer behavior that is close to what shall be described by a specification (if it is manually constructed). On the other hand, specification-based test generation can help produce a good-quality test suite but requires specifications, which often do not exist in practice. There seems to be a circular dependency between dynamic behavior inference and (specification-based) test generation. To exploit the circular dependency and alleviate the problem, we propose a feedback loop between behavior inference and test generation. The feedback loop starts with an existing test suite (constructed manually or automatically) or some existing program runs. By using one of the behavior-inference components (the test selector, test abstractor, or program-spectra comparator), we first infer behavior based on the existing test suite or program runs. We then feed inferred behavior to a specification-based test-generation tool or a test-generation tool that can exploit the inferred behavior to improve its test generation. The new generated tests can be used to infer new behavior. The new behavior can be further used to guide test generation in the subsequent iteration. Iterations terminate until a user-defined maximum iteration number has been reached or no new behavior has been inferred from new tests. This feedback loop provides a means to producing better tests and better approximated specifications automatically and incrementally. In addition, the by-products of the feedback loop are a set of selected tests for inspection; these selected tests exhibit new behavior that has not been exercised by the existing tests.

3. Conclusion

We have proposed a framework for improving effectiveness of automated testing in the absence of specifications. A set of techniques and tools have been developed within the framework. First, we have defined redundant tests based on method input values and developed a tool for detecting redundant tests among automatically generated tests; these identified redundant tests increase testing time without increasing the ability to detect faults or increasing develop-

ers' confidence on the program under test. Second, we have developed a tool that generates only non-redundant tests by executing method calls symbolically to explore the symbolic-state space. Symbolic execution not only allows us to reduce the state space for exploration but also generates relevant method arguments automatically. Third, we have used Daikon [4] to infer behavior exercised by the existing tests and fed the inferred behavior in the form of specifications to a specification-based test generation tool [11]. Developers can inspect those generated tests that violate these inferred behavior, instead of inspecting a large number of all generated tests. Fourth, we have used the returns of observer methods to group concrete states into abstract states, from which we construct succinct observer abstractions for inspection. Fifth, we have defined value spectra to characterize program behavior, compared the value spectra from an old version and a new version, and used the spectra differences to detect behavior deviations in the new version. We have further used value spectra differences to locate deviation roots. Finally, putting behavior inference and test generation together, we can construct a feedback loop between these two types of dynamic analysis, starting with an existing test suite (constructed manually or automatically) or some existing program runs. The feedback loop produces better tests and better approximated specifications automatically and incrementally.

4. Acknowledgments

I would like to thank my Ph.D. advisor, David Notkin, and my collaborators in my Ph.D. research: Darko Marinov, Amir Michail, Wolfram Schulte, and Jianjun Zhao.

References

[1] Agitar Agitator 3.0, 2005. `http://www.agitar.com/`.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[3] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[5] E. Gamma and K. Beck. JUnit, 2003. `http://www.junit.org`.

[6] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[7] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[9] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[10] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[11] Parasoft Jtest 4.5, 2003. `http://www.parasoft.com/`.

[12] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. 6th ESEC/FSE*, pages 432–449, 1997.

[13] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[14] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[15] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.

[16] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

[17] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[18] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, pages 290–305, Nov. 2004.

[19] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems*, pages 39–46, October 2004.

[20] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.