Appears in Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)

Mutually Enhancing Test Generation and Specification Inference

Tao Xie and David Notkin

Department of Computer Science and Engineering
University of Washington at Seattle
WA 98195, USA
{taoxie, notkin}@cs.washington.edu

Abstract. Generating effective tests and inferring likely program specifications are both difficult and costly problems. We propose an approach in which we can mutually enhance the tests and specifications that are generated by iteratively applying each in a feedback loop. In particular, we infer likely specifications from the executions of existing tests and use these specifications to guide automatic test generation. Then the existing tests, as well as the new tests, are used to infer new specifications in the subsequent iteration. The iterative process continues until there is no new test that violates specifications inferred in the previous iteration. Inferred specifications can guide test generation to focus on particular program behavior, reducing the scope of analysis; and newly generated tests can improve the inferred specifications. During each iteration, the generated tests that violate inferred specifications are collected to be inspected. These violating tests are likely to have a high probability of exposing faults or exercising new program behavior. Our hypothesis is that such a feedback loop can mutually enhance test generation and specification inference.

1 Introduction

There are a variety of software quality assurance (SQA) methods being adopted in practice. Since there are particular dependences or correlations among some SQA methods, these methods could be integrated synergistically to provide value considerably beyond what the separate methods can provide alone [28, 32, 35, 11]. Two such exemplary methods are specification-based test generation and dynamic specification inference. Specification-based test generation requires specifications a priori [13, 25, 5]. In practice, however, formal specifications are often not written for programs. On the other hand, dynamic specification inference relies on good tests to infer high quality specifications [10, 31, 21]. There is a circular dependency between tests in specification-based test generation and specifications in dynamic specification inference.

In addition, when formal specifications are not available, automatic test generation, such as white-box test generation or random test generation, does not sufficiently address output checking. Without specifications, output checking is limited to detecting a program crash, or an exception is thrown but not caught. In other words, there is a lack of test oracles in automatic test generation without specifications *a priori*.

In this research, without *a priori* specifications, we want to mutually enhance test generation and specification inference. At the same time, from a large number of generated tests, we want to have a way to identify valuable tests for inspection. Valuable tests can be fault-revealing tests or tests that exercise new program behavior. The solution we propose is a method and tools for constructing a feedback loop between test generation and specification inference, using and adapting existing specification-based test generation and dynamic specification inference techniques. We implement the method for three types of inferred specifications: axiomatic specifications [10], protocol specifications [31], and algebraic specifications [21]. We demonstrate the usefulness of the method by initially focusing on the unit test generation and the specification inference for object-oriented components, such as Java classes.

2 Background

2.1 Formal Specifications

A formal specification expresses the desired behavior of a program. We model the specification in a style of *requires*/*ensures*. *Requires* describe the constraints of using APIs provided by a class. When *requires* are satisfied during execution, *ensures* describe the desired behavior of the class. *Requires* can be used to guard against illegal inputs, and *ensures* can be used as test oracles for correctness checking.

Axiomatic specifications [22] are defined in the granularity of a method in a class interface. Preconditions for a method are *requires* for the method, whereas post-conditions for a method are *ensures* for the method. Object invariants in axiomatic specifications can be viewed as the pre/post-conditions for each method in the class interface. The basic elements in *requires*/*ensures* consist of method arguments, returns, and class fields.

Protocol specifications [6] are defined in the granularity of a class. *Requires* are the sequencing constraints in the form of finite state machines. Although extensions to protocol specifications can describe *ensures* behavior, there are no *ensures* in basic protocol specifications. The basic elements in *requires* consist of method calls, including method signatures, but usually no method arguments or returns.

Algebraic specifications [18] are also defined in the granularity of a class. *Ensures* are the *AND* combination of all axioms in algebraic specifications. In the *AND* combination, each axiom, in the form of *LHS*=*RHS*, is interpreted as "if a current call sequence window instantiates *LHS*, then its result is equal to *RHS*". The basic elements in *ensures* consist of method calls, including method signature, method arguments and returns, but no class fields. Therefore, algebraic specifications are in a higher-level abstraction than axiomatic specifications are. Usually there are no explicit *requires* in algebraic specifications. Indeed, sequencing constraints, which are *requires*, can be derived from the axiom whose *RHS* is an error or exception [4].

2.2 Dynamic Specification Inference

Dynamic specification inference discovers operational abstractions from the executions of tests [20]. An operational abstraction is syntactically identical to a formal specification. The discovered operational abstractions consist of those properties that hold for all the observed executions. These abstractions can be used to approximate specifications or indicate the deficiency of tests.

Ernst et al. [10] develop a dynamic invariant detection tool, called Daikon, to infer likely axiomatic specifications from executions of test suites. It examines the variable values that a program computes, generalizes over them, and reports the generalizations in the form of pre/post-conditions and class invariants.

Whaley et al. [31] develop a tool to infer likely protocol specifications from method call traces collected while a Java class interface is being used. These specifications are in the form of multiple finite state machines, each of which contains methods accessing the same class field. Ammons et al. [1] develop a tool to infer likely protocol specifications from C method call traces by using an off-the-shelf probabilistic finite state automaton learner. Hagerer et al. [19] present the regular extrapolation technique to discover protocol specifications from execution traces of reactive systems.

Henkel and Diwan [21] develop a tool to derive a large number of terms for a Java class and generate tests to evaluate them. The observational equivalence technique [3, 9] is used to evaluate the equality among these terms. Based on the evaluation results, equations among these terms are proposed, and are further generalized to infer axioms in algebraic specifications.

2.3 Specification-Based Test Generation

We categorize specification-based test generation into test generation for functionality and test generation for robustness. Test generation for functionality generates tests that satisfy *requires*, and checks whether *ensures* are satisfied during test executions. Test generation for robustness generates tests that may not satisfy *requires*, and checks whether a program can handle these test executions gracefully, such as throwing appropriate exceptions.

We divide the test generation problem into three sub-problems: object state setup, method parameter generation, and method sequence generation. Object state setup puts the class under test into particular states before invoking methods on it. Method parameter generation produces particular arguments for methods to be invoked. Method sequence generation creates particular method call sequences to exercise the class on certain object states. Axiomatic specifications provide more guidance on both method parameter generation and object state setup, whereas algebraic specifications provide more guidance on method sequence generation. Protocol specifications provide more guidance on both object state setup and method sequence generation.

Dick and Faivre develop a tool to reduce axiomatic specifications to a disjunctive normal form and generate tests based on them [8]. Boyapati et al. develop a tool to generate tests effectively by filtering the test input space based on preconditions in axiomatic specifications [5]. Gupta develop a structural testing technique to generate test inputs to exercise some particular post-conditions or assertions [16]. A commercial Java unit testing tool, ParaSoft Jtest [24], can automatically generate test inputs to perform white box testing when no axiomatic specifications are provided, and perform black box testing when axiomatic specifications are equipped.

There is a variety of test generation techniques based on protocol specifications, which is in the form of finite state machines [25]. There are several test generation tools based on algebraic specifications. They generate tests to execute the *LHS* and *RHS* of an axiom. The DAISTS tool developed by Gannon et al. [13] and the Daistish tool developed by Hughes and Stotts [23] use an implementation-supplied equality method to compare the results of *LHS* and *RHS*. A tool developed by Bernot et al. [3] and the ASTOOT tool developed by Doong and Frankl [9] uses observational equivalence to determine whether *LHS* and *RHS* are equal.

3 Feedback Loop Framework

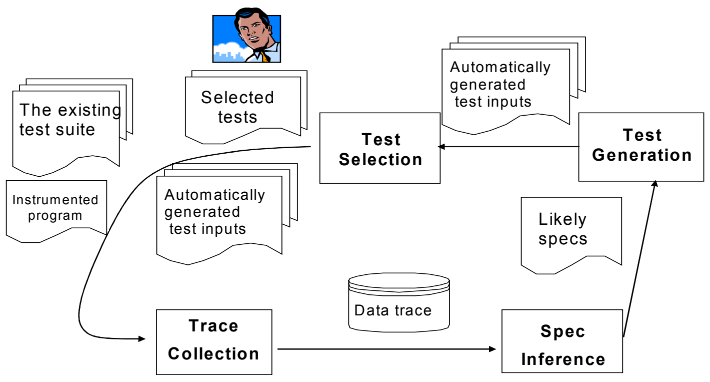

Fig. 1. An overview of the feedback loop framework

Our approach can be viewed as a black box into which a developer feeds a program and its existing tests, and from which the developer gets a set of valuable tests, inferred specifications, and reasons why these tests are valuable. Then the developer can inspect the valuable tests and inferred specifications for problems. Our feedback loop framework consists of multiple iterations. Each iteration is given the program, a set of tests, and specifications inferred from the previous iteration (except for the first iteration). After each iteration, a complete set of new tests, a valuable subset of new tests, reasons for being valuable, and new inferred specifications are produced. The subsequent iteration is given the original tests augmented by the complete set of new tests or the valuable subset of new tests, as well as the new inferred specifications. Optionally the developer can specify some iteration-terminating conditions, such as a stack size being equal to the maximum capacity, or the number of iterations reaching the specified number. The iterations continue until user-specified conditions are satisfied and there is no new test that violates specifications inferred in the previous iteration.

Figure 1 shows an overview of the feedback loop framework. The framework defines four stages for each iteration: trace collection, specification inference, test generation, and test selection. Human intervention is only needed for inspecting selected tests and inferred specifications in the end of the feedback loop. But human intervention may be incorporated in the end of each iteration and should improve results.

In the trace collection stage, the given tests are run on the instrumented Java program and traces are collected from the executions. Object states are defined by some particular relevant class field values. The values of method arguments, returns, and object states are recorded at the entry and exit of a method execution. To collect object states, we instrument invocations of *this.equals(this)* at the entry and exit of each public method in the Java class file. Then we monitor the class field values accessed by the execution of *this.equals(this)*. These values are collected as object states. The collected method arguments, returns, and object states are used in the specification inference stage and the test generation stage.

In the specification inference stage, the collected traces are used to infer specifications. The axiomatic and protocol specification inference techniques in Section 2.2 are used in this stage. Instead of using the algebraic specification inference technique based on observational equivalence [21], we develop a tool prototype to infer algebraic specifications based on an implementation-supplied equality method. Since it is expensive to execute the equality method to compare object states among all method executions, we use object states collected in the trace collection stage to compare the object states offline. Based on a set of pre-defined axiom-pattern templates, the tool looks for equality patterns among collected object states, method arguments, and returns of methods. We infer algebraic specifications by using these equality patterns as axioms.

In the test generation stage, inferred specifications are used to guide test generation. Jtest [24] is used to automatically generate tests based on axiomatic specifications. In protocol and algebraic specification-based test generation, we grow new object states and method parameters based on the collected traces in the present iteration. In addition, we generate the method sequences based on inferred protocol and algebraic specifications.

Because inferred preconditions in axiomatic specifications may be overconstrained, only generating test inputs that satisfy them would leave some interesting legal test inputs out of scope. One solution is to remove all the inferred preconditions before the specifications are used to guide test generation. Then both legal and illegal test inputs can be generated. Allowing some illegal inputs can still be useful in testing program robustness. However, removing inferred preconditions makes test generation based on preconditions unguided. In future work, we plan to investigate techniques to remove or relax parts of inferred preconditions. There are similar overconstrained problems with protocol specifications. To address these problems, we can deliberately generate some method sequences that do not follow the transitions in the inferred finite state machines. For example, we can generate test sequences to exercise the complement of the inferred finite state machines. The test generation based on inferred algebraic specifications also needs some adaptations. If not all the combinations of method pairs are exercised, we need to generate tests to exercise those uncovered method pairs besides those method pairs in the inferred axioms.

In the test selection stage, the generated tests are executed, and checked against the inferred specifications. Two types of tests are selected for inspection. The first type of test is the test whose execution causes an uncaught runtime exception or a program crash. If the test is a legal input, it may expose a program fault. The second type of test is the test whose execution violates *ensures* in axiomatic specifications or algebraic specifications. If the *ensures* violated by the test are overconstrained ones, this may indicate the insufficiency of the existing tests. If the violated *ensures* are actual ones and the test is a legal input, it may expose a program fault. These selected tests are collected as the candidates of valuable tests. In the end of the feedback loop, the developer can inspect these selected tests and their violated specifications for problems. If a selected test input is an illegal one, the developer can either add preconditions to guard against this test input in the subsequent iteration, or adopt defensive programming to throw appropriate exceptions for this test input. If a selected test input is a legal one and it exposes a program fault, the developer can fix the bug that causes the fault, and augment the regression test suite with this test input after adding an oracle for it. If a selected test input is a legal one and it does not expose a fault but exercise certain new program behavior, the developer can add it to the regression test suite together with its oracle. Besides selecting these two types of tests, the developer can also select those tests that exercise at least one new structural entity, such as statement or branch.

In our experiments with the feedback loop for axiomatic specifications, the number of selected tests is not large, which makes the human inspection effort affordable [34]. In addition, the selected tests have a high probability of exposing faults or exercising new program behavior. We observed the similar phenomena in our preliminary experiment with the feedback loop for algebraic specifications.

The selected tests, or all the newly generated tests from the present iteration are used to augment the existing tests in the subsequent iteration. The inferred specifications from the present iteration are also used in the specification inference stage of the subsequent iteration. In this specification inference stage, conditional specifications might be inferred to refine some of those specifications that are violated by the generated tests in the present iteration.

4 Related Work

There have been several lines of work that use feedback loops in static analyses. Ball and Rajamani construct a feedback loop between program abstraction and model checking to validate user-specified temporal safety properties of interfaces [2]. Flanagan and Leino use a feedback loop between annotation guessing and theorem proving to infer specifications statically [12]. Wild guesses of annotations are automatically generated based on heuristics before the first iteration. Human interventions are needed to insert manual annotations in subsequent iterations. Giannakopoulou et al. construct a feedback loop between assumption generation and model checking to infer assumptions for a user-specified property in compositional verification [14, 7].

Given crude program abstractions or properties, these feedback loops in static analyses use model checkers or theorem provers to find counterexamples or refutations. Then these counterexamples or refutations are used to refine the abstractions or properties iteratively. Our work is to construct a feedback loop in dynamic analyses, corresponding to the ones in static analyses. Our work does not require users to specify properties, which are inferred from test executions instead.

Naumovich and Frankl propose to construct a feedback loop between finite state verification and testing to dynamically confirm the statically detected faults [26]. When a finite state verifier detects a property violation, a testing tool uses the violation to guide test data selection, execution, and checking. The tool hopes to find test data that shows the violation to be real. Based on the test information, human intervention is used to refine the model and restart the verifier. This is an example of a feedback loop between static analysis and dynamic analysis. Another example of a feedback loop between static analysis and dynamic analysis is profile-guided optimization [30]. Our work focuses on the feedback loop in dynamic analyses.

Peled et al. present the black box checking [29] and the adaptive model checking approach [15]. Black box checking tests whether an implementation with unknown structure or model satisfies certain given properties. Adaptive model checking performs model checking in the presence of an inaccurate model. In these approaches, a feedback loop is constructed between model learning and model checking, which is similar to the preceding feedback loops in static analyses. Model checking is performed on the learned model against some given properties. When a counterexample is found for a given property, the counterexample is compared with the actual system. If the counterexample is confirmed, a fault is reported. If the counterexample is refuted, it is fed to the model learning algorithm to improve the learned model. Another feedback loop is constructed between model learning and conformance testing. If no counterexample is found for the given property, conformance testing is conducted to test whether the learned model and the system conform. If they do not conform, the discrepancy-exposing test sequence is fed to the model learning algorithm, in order to improve the approximate model. Then the improved model is used to perform model checking in the subsequent iteration. The dynamic specification inference in our feedback loop is corresponding to the model learning in their feedback loop, and the specification-based test generation in our feedback loop is corresponding to the conformance testing in their feedback loop. Our feedback loop does not require some given properties, but their feedback loop requires user-specified properties in order to perform model checking.

Gupta et al. use a feedback loop between test data generation and branch predicate constraint solving to generate test data for a given path [17]. An arbitrarily chosen input from a given domain is executed to exercise the program statements relevant to the evaluation of each branch predicate on the given path. Then a set of linear constraints is derived. These constraints can be solved to produce the increments for the input. These increments are added to the current input in the subsequent iteration. The specification inference in our work is corresponding to the branch predicate constraints in their approach. Our work does not require users to specify a property, whereas the work of Gupta et al. requires users to specify the path to be covered.

5 Conclusion

We have proposed a feedback loop between specification-based test generation and dynamic specification inference. This feedback loop can mutually enhance both test generation and specification inference. The feedback loop provides aids in test generation by improving the underlying specifications, and aids in specification inference by improving the underlying test suites. We have implemented a feedback loop for axiomatic specifications, and demonstrated its usefulness [33, 34]. We have developed an initial implementation of feedback loop for algebraic specifications, and plan to do more experiments and refine the implementation. In future work, we plan to implement and experiment the feedback loop for protocol specifications. At the same time, the following research questions are to be further investigated. In the first iteration, the inferred specifications can be used to generate a relatively large number of new tests. In the subsequent iterations, the marginal improvements on tests and specifications come from the specification refinement and object state growth. We need to explore effective ways to maximize these marginal improvements. We also plan to investigate other SQA methods, such as static verification techniques, in evaluating the quality of the inferred specifications in iterations [27].

6 Acknowledgment

We thank Michael Ernst, Andrew Petersen, and Darko Marinov for fruitful discussion. This work was supported in part by the National Science Foundation under grant ITR 0086003. We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

1. Ammons, G., Bodik, R., Larus, J. R.: Mining specification. In Proceedings of Principles of Programming Languages, Portland, Oregon, (2002) 4-16
2. Ball, T., Rajamani, S. K.: Automatically Validating Temporal Safety Properties of Interfaces. In Proceedings of SPIN Workshop on Model Checking of Software, (2001) 103-122
3. Bernot, G., Gaudel, M. C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal, Nov. (1991) 387-405
4. Bieman, J.M., Olender, K.M.: Using algebraic specifications to find sequencing defects. In Proceedings of 4th International Symposium on Software Reliability Engineering, (1993) 226–232
5. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In Proceedings of the 2002 International Symposium on Software Testing and Analysis, Rome, Italy, (2002) 123-133
6. Campbell, R. H., Habermann, A. N.: The Specification of Process Synchronization by Path Expressions. vol. 16, Lecture Notes in Computer Science, (1974) 89-102

7. Cobleigh, J. M., Giannakopoulou, D., Pasareanu, C. S.: Learning Assumptions for Compositional Verification. In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (2003) 331-346
8. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In Proceedings of International Symposium of Formal Methods Europe, (1993) 268-284
9. Doong, R., Frankl, P. G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering, 3(2), Apr. (1994) 101-130
10. Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol. 27, no. 2, Feb. (2001) 1-25
11. Ernst, M. D.: Static and dynamic analysis: Synergy and duality. In Proceedings of ICSE Workshop on Dynamic Analysis. Portland, Oregon, May (2003) 24-27
12. Flanagan, C., Leino, K. R. M.: Houdini, an annotation assistant for ESC/Java. In Proceedings of International Symposium of Formal Methods Europe, March (2001) 500-517
13. Gannon, J., McMullin, P., Hamlet, R.: Data-Abstraction Implementation, Specification, and Testing. ACM Transactions on Programming Languages and Systems, Vol.31, No. 3, July (1981) 211-223
14. Giannakopoulou, D., Pasareanu, C. S., Barringer, H.: Assumption Generation for Software Component Verification. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, (2002) 3-12
15. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Apr. (2002) 357-370
16. Gupta, N.: Generating Test Data for Dynamically Discovering Likely Program Invariants. In Proceedings of ICSE 2003 Workshop on Dynamic Analysis, May (2003) 21-24
17. Gupta, N., Mathur, A. P., Soffa, M. L.: Automated test data generation using an iterative relaxation method. In Proceedings of International Symposium on Foundations of Software Engineering, November (1998) 231-244
18. Guttag, J. V., Horning, J. J.: The algebraic specification of abstract data types. Acta Information, 10, (1978) 27-52
19. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model Generation by Moderated Regular Extrapolation. In Proceedings of International Conference on Fundamental Approaches to Software Engineering, Heidelberg, Germany, April (2002) 80-95
20. Harder, M., Mellen, J., Ernst, M. D.: Improving test suites via operational abstraction. In Proceedings of the International Conference on Software Engineering, Portland, Oregon, May (2003) 60-71
21. Henkel, J., Diwan, A.: Discovering algebraic specifications from Java classes. In Proceedings of European Conference on Object-Oriented Programming, July (2003) 431-456
22. Hoare, C. A. R.: An axiomatic basis for computer programming. Communications of the ACM, 12(10), (1969) 576–580
23. Hughes, M., Stotts, D.: Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In Proceedings of the International Symposium on Software Testing and Analysis, San Diego, California, (1996) 53-61
24. Parasoft Corporation, Jtest manuals version 4.5, http://www.parasoft.com/, October 23, 2002
25. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. In Proceedings of the IEEE, vol. 84, 8, (1996) 1090-1123
26. Naumovich, G., Frankl, P. G.: Toward Synergy of Finite State Verification and Testing. In Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, June (2000) 89-94

27. Nimmer, J. W., Ernst, M. D.: Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In Proceedings of First Workshop on Runtime Verification, Paris, France, ENTCS 55 (2), (2001)
28. Osterweil, L. J., et al.: Strategic directions in software quality. ACM Computing Surveys, (4), Dec. (1996) 738-750
29. Peled, D., Vardi, M. Y., Yannakakis, M.: Black Box Checking. In Proceedings of FORTE/PSTV, Beijing, China, Kluwer, (1999) 225-240
30. Pettis, K., Hansen, R. C.: Profile guided code positioning. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, White Plains, NY, USA, (1990) 16-27
31. Whaley, J., Martin, M. C., Lam, M. S.: Automatic Extraction of Object-Oriented Component Interfaces. In Proceedings of the International Symposium on Software Testing and Analysis, July (2002) 218-228
32. Xie, T., Notkin, D.: Macro and Micro Perspectives on Strategic Software Quality Assurance in Resource Constrained Environments. In Proceedings of 4th International Workshop on Economics-Driven Software Engineering Research, May (2002) 66-70
33. Xie, T., Notkin, D.: Exploiting Synergy between Testing and Inferred Partial Specifications. In Proceedings of ICSE 2003 Workshop on Dynamic Analysis, Portland, Oregon, (2003) 17-20
34. Xie, T., Notkin, D.: Tool-Assisted Unit Test Selection Based on Operational Violations. In Proceedings of 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada, Oct. (2003) 40-48
35. Young, M.: Symbiosis of Static Analysis and Program Testing. In Proceedings of 6th International Conference on Fundamental Approaches to Software Engineering, Warsaw, Poland, April (2003) 1-5