

# Automated Object-Oriented Testing with Inferred Program Properties

Tao Xie

Department of Computer Science & Engineering  
University of Washington  
Seattle, WA 98195, USA  
taoxie@cs.washington.edu

## ABSTRACT

Automatic object-oriented test generation tools are powerful: they are able to produce and execute a large number of test inputs that extensively exercise the class under test. However, such a large number of test inputs contain a high percentage of redundant tests, which increase the testing time without increasing the ability to detect faults. Moreover, without a priori specifications, programmers need to manually verify the outputs of these test executions, which is generally impractical. In this paper, we propose to use program properties inferred from test executions to address these two issues. Based on inferred equivalence properties, we detect redundant tests for test minimization or redundant-test avoidance during test generation. Based on various inferred properties, we select valuable tests and abstract test behavior for inspection.

## 1. DESCRIPTION OF PURPOSE

More and more programmers write down unit tests before, during, or after they write object-oriented code. Although manually written tests are very useful, relying on them only is still insufficient to assure high software quality and reliability. The main alternative is to use one of a variety of automatic test generation tools: these are able to produce and execute a large number of test inputs that extensively exercise the class under test. However, there are two important problems with such a large number of test inputs. The first problem is redundant-test problem. Being ignoring object states, test generation tools often generate a high percentage of redundant tests, which increase the testing time without increasing the ability to detect faults. This problem is significant in that redundant tests not only waste the machine time in their generation and execution but also waste human time in their inspection and maintenance. The second problem is test-inspection problem. Without a priori specifications (being common in practice), programmers need to manually verify the outputs of these test executions<sup>1</sup>, which is generally impractical. This problem is significant in that when programmers are unable to inspect these automatically generated tests for correctness, the value of these tests is under-exploited: even if running a test on the program produces wrong results, this behavior might get unnoticed.

<sup>1</sup>Without a priori specifications, existing automatic test generation tools report uncaught exceptions thrown by test executions. These symptoms are useful for robustness checking but still often insufficient for correctness checking

Most existing research on redundant test detection is based on specification or structural coverage [4]. A test is redundant for a test suite iff the specification or structural coverage achieved by the test has been achieved by some other tests in the test suite. However, if we remove redundant tests identified by existing research from a test suite, the quality of the test suite is usually sacrificed. Most existing research on test selection for inspection is primarily based on structural profiles [1]. However, many valuable unit tests are characterized not by structural profiles but variable values.

## 2. GOAL STATEMENT

The main goal of this research is to improve automated object-oriented testing by detecting and avoiding redundant tests, selecting valuable tests for inspection, and abstracting useful test behavior for inspection. Different from existing research [4], removing redundant tests detected by this research shall preserve the quality of the original test suite. Tests selected and behavior abstracted by this research shall be valuable and useful to programmers.

Our research operates in the semantic domain of program properties instead of the syntactic domain of program text, in which most existing research operates. We infer program properties from test executions and use them to improve automated testing. The inferred program properties are usually in the form of formal specifications. This research offers some benefits of formal methods in testing without requiring programmers to write formal specifications.

## 3. APPROACHES

To address the redundant-test problem, we propose a framework for redundant-test detection based on inferred equivalence properties. To address the test-inspection problem, we propose to select tests and abstract tests for inspection based on various inferred properties.

### 3.1 Redundant-Test Detection

We propose *Rostra*, a framework for detecting object-oriented redundant tests based on inferred equivalence properties among object states and method executions [5]. We propose various representations for an object state by using method sequences or object field values. Two object states are equivalent iff their representations are the same. A method execution is characterized by the entry object state of the receiver (called the *method-entry state*) and the object states of method arguments. Two method executions are

equivalent iff their method-entry states and argument object states are equivalent correspondingly. Object-oriented unit tests consist of sequences of method executions. A unit test is redundant for a test suite iff for each method execution of the unit test, there exists an equivalent method execution of some test from the test suite.

We conduct experiments to validate our framework. We measure the percentage of detected redundant tests among automatically generated tests. This measurement shows how much benefit we can achieve by using the framework. We compare the count of uncaught thrown exceptions, branch coverage, and mutant killing ratios of both an original test suite and its minimized test suite. These measurements validate that removing detected redundant tests does not decrease the quality of the original test suite.

### 3.2 Test Selection

We propose two rationales for test selection:

- An automatically generated test is selected when the test exercises a behavior that is not exercised by existing selected tests.
- An automatically generated test is selected when the test exercises a behavior that is universally or commonly exercised by generated tests, or violates a behavior that is commonly exercised by generated tests.

The first rationale splits available tests into two sets: selected and unselected tests, and incrementally grow the set of selected tests until no tests can be selected from the set of unselected tests. In contrast, the second rationale processes all available tests collectively.

Based on the first rationale, we have proposed the *operational violation* approach for selecting automatically generated tests [6]. A behavior is characterized by an inferred property, *operational abstraction* [2], in the form of axiomatic specifications. We select an automatically generated test if the test violates at least one operational abstraction inferred from existing selected tests.

Based on the second rationale, we propose the *statistical algebraic abstractions* approach for selecting automatically generated tests. A behavior is characterized by an inferred property, *algebraic abstraction*, in the form of algebraic specifications. Our property inference differs from previous work [2, 3] in that inferred properties are not required to be true universally among available tests. We select one (special) test that violates a common algebraic abstraction, which has a majority of satisfying instances. We select one (common) test that satisfies a common or universal algebraic abstraction, which has a majority of or all satisfying instances.

When our approaches select a test, we also annotate the test with the reason why it is selected: the behavior that it exercises or violates. This extra information can help programmers to determine the correctness of the test execution efficiently.

We conduct experiments to validate our test selection approaches. We measure the number of all automatically generated tests and the number of selected tests. The first measurement can show whether the inspection cost for all automatically generated tests is expensive; otherwise, we do not need to perform test selection but inspect all tests. The second measurement can show whether the inspection cost for selected tests is affordable. We also measure the percentage of new-behavior-exposing tests among selected tests. This

measurement validates that the selected tests are valuable for inspection.

### 3.3 Test Abstraction

Instead of selecting a subset of tests for inspection, we propose to abstract the behavior of test executions for inspection. Our test selection approaches focus on those local properties inside a method or between two methods, whereas our test abstraction approach focuses on those global properties among multiple methods.

We propose the *observer abstraction* approach for abstracting the behavior of test executions [7]. From test executions, we first construct concrete object state machines, where a state represents a nonequivalent concrete object state and a transition represents a method call. We then map each nonequivalent concrete object state to an abstract state based on the return values of a set of observers (public methods with non-void returns) invoked on the object state. This process produces a set of abstract object state machines called *observer abstractions*. Programmers can inspect observer abstractions for correctness checking, fault isolation, test characterization, and component understanding.

We conduct case studies to validate our test abstraction approach. We investigate whether and how observer abstractions can help programmers to inspect test executions effectively.

## 4. REFERENCES

- [1] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. the 23rd international conference on Software engineering*, pages 339–348, 2001.
- [2] 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.
- [3] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.
- [4] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. the International Conference on Software Maintenance*, pages 34–43, 1998.
- [5] 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*, Sept. 2004.
- [6] 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.
- [7] 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*, Nov. 2004.