Won **the third place** in 2004 ACM SIGPLAN Student Research Competition at OOPSLA 04.
Appears in *Proceedings of the 19th Annual ACM SIGPLAN OOPSLA Companion 04*, Vancouver, Canada

# Automatic Identification of Common and Special Object-Oriented Unit Tests

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

## ABSTRACT

Common and special test inputs can be created to exercise some common and special behavior of the class under test, respectively. Although manually created tests are valuable, programmers often overlook some special test inputs. If programmers write down specifications, special or common tests can be automatically generated and selected by tools. However, specifications are not commonly written in practice. This research develops a novel approach for automatically identifying common and special unit tests for a class without requiring any specification. Given a class, our approach automatically generates test inputs and identifies common and special tests among the generated tests. Programmers can inspect these identified tests and use them to augment existing (manual) tests. Our approach is based on statistical algebraic abstractions, program properties (in the form of algebraic specifications) dynamically inferred from test executions. We use statistical algebraic abstractions to characterize program behavior. A test is identified to be common if the test exercises a behavior that is universally or commonly exercised by generated tests, or to be special if the test violates a behavior that is commonly exercised by generated tests.

**Categories and Subject Descriptors:** D.2.5 [Testing and Debugging]: Testing tools

**General Terms:** Reliability.

**Keywords:** Test Selection, Dynamic Inference.

## 1. INTRODUCTION

In unit testing, the class under test might exhibit common and special program behavior when it is exercised by different test inputs. For example, intuitively a bounded-stack class exhibits common behavior when the stack is neither empty nor full, but might exhibit some special behavior when the stack is empty or full. Common and special test inputs can be created to exercise some common and special behavior of the class under test, respectively. Although manually written unit tests for classes play an important role in software development, they are often insufficient to exercise some important common or special behavior of the class: programmers often overlook some special or boundary values and sometimes even fail to include some common cases. The main complementary approach is to use one of the automatic unit test generation tools to generate a large number of test inputs to exercise a variety of behaviors of the class. With a priori specifications, the executions of these test inputs can be automatically verified. In addition, among generated tests, common and special tests can be identified based on specifications and then these identified tests can be used to augment existing manual tests. However, in practice, specifications are often not written by programmers. Without a priori specifications, it is impractical for programmers to manually inspect and verify the outputs of such a large number of test executions. Consequently programmers do not have an efficient way to identify common and special tests. In this paper, we present a new approach for automatically identifying common and special tests from automatically generated tests without requiring specifications. Programmers can inspect these identified tests for correctness and use them to augment existing manual tests. To illustrate the approach, in this paper we use a bounded stack class `BStack` whose public methods are shown in Figure 1. The capacity of the stack is hardcoded as three for simplicity.

```
public class BStack {
  public BStack() { ... }
  public void push(int i) { ... }
  public void pop(){ ...  }
  public int top() { ...  }
  public boolean isMember(int i) { ... }
  public boolean isEmpty(){ ... }
  public boolean equals(Object s) { ... }
}
```

**Figure 1: The interface of a `bounded stack` class**

## 2. APPROACH

Our new approach is based on statistical algebraic abstractions. An *algebraic abstraction* is an equation that abstracts a program's runtime behavior. It is syntactically identical to an axiom in algebraic specifications. For example, two algebraic abstractions of `BStack` are

`isMember(BStack().state,` $i_2$`).retval == false` and

`isEmpty(push(S,` $i_1$`).state).retval == false`

where the receiver of a method call is treated as the first method argument (but the constructor `BStack()` does not have a receiver) and the `.state` and `.retval` expressions denote the state of the receiver after the invocation and the result of the invocation, respectively. We adopt the notation following Henkel and Diwan [2].

An instance of an algebraic abstraction is a test that is able to instantiate the left-hand side and right-hand side of the equation in the algebraic abstraction. A satisfying instance of an algebraic abstraction is an instance that sat-

Copyright is held by the author/owner.
*OOPSLA'04,* Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

isfies the equation in the algebraic abstraction. A violating instance of an algebraic abstraction is an instance that violates the equation in the algebraic abstraction. A *statistical algebraic abstraction* is an algebraic abstraction that is associated with the counts of its satisfying and violating instances. We dynamically infer statistical algebraic abstractions from test executions.

A *common property* is a statistical algebraic abstraction whose instances are mostly satisfying instances (by default, our approach sets the percentage threshold of satisfying instances as 80%, which can be configured by the user). A *universal property* is a statistical algebraic abstraction whose instances are all satisfying instances. A *special test* is a violating instance of a common property and a *common test* is a satisfying instance of a common or universal property. For each common property, we sample and select a special test and a common test. For each universal property, we also sample and select a common test.

Our statistical inference differs from previous work on specification inference (axiomatic-specification inference [1] and algebraic-specification inference [2]) in that the abstractions inferred by our approach are not required to be universally true among all test executions.

Figure 2 shows an overview of our approach. The input to our approach is the bytecode of the (Java) class under test and a set of algebraic abstraction templates pre-defined by us; these templates encode common forms of axioms in algebraic specifications: equality relationships among two neighboring method calls and single method calls. The outputs of the approach are a set of common and special tests and their corresponding properties. The approach comprises four steps: test generation, method-call composition, statistical inference, and test identification. The step of test generation first generates different representative argument values for each public method of the class, and then dynamically and iteratively invokes different method arguments on each non-equivalent receiver-object state (our previous work [3] develops techniques for determining object-state equivalence). The step of method-call composition monitors and collects method executions to compose two method calls `m1` and `m2` forming a method-call pair if `m1`'s receiver-object state *after* invoking `m1` is equivalent to `m2`'s receiver-object state *before* invoking `m2`. The composed method-call pair is used in the step of statistical inference as if the two method calls in the pair were invoked in a row on the same receiver. The step of statistical inference uses method-call pairs and single method calls to instantiate and check against the abstraction templates. This step produces a set of common or universal properties. The step of test identification identifies common and special tests based on these properties.

## 3. PRELIMINARY RESULTS

We have implemented the approach and performed preliminary experiments on several data structures, including the `BStack` class and some nontrivial classes experimented in our previous work [3]. This section illustrates some results for `BStack`[1]. Our approach identifies 19 common tests and 10 special tests (4 tests are both common and special) out of 361 automatically generated tests. Our approach infers 9 universal properties and 20 common properties. Universal properties include the two abstractions shown in Section 2.

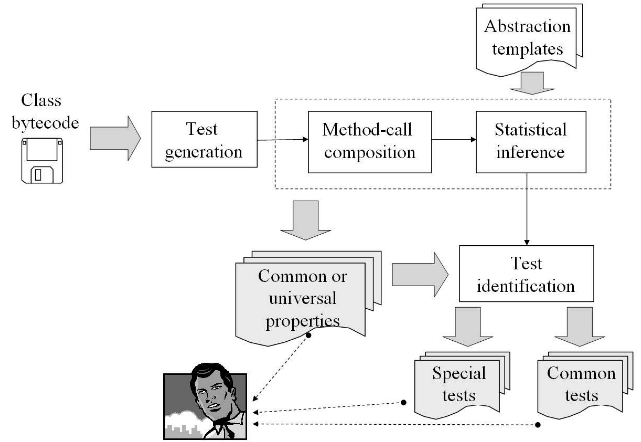


**Figure 2: An overview of common and special test identification**

Common properties include the following one:

`isMember(push(S,` $i_1$`).state,` $i_2$`).retval==true where` $i_1$`==`$i_2$

This property has 96 satisfying instances and 24 violating instances. Henkel and Diwan's approach [2] cannot infer such a property because this property is not universally true among test executions. Intuitively this property states that if we push an element into the stack and then query the stack with the same element using `isMember`, the return value of `isMember` is `true`. It seems to be the correct behavior to us. But there are a small percentage of special tests violating them, one of them is:

```
uniqueBoundedStack u1 = new uniqueBoundedStack();
u1.push(-1);
u1.push(-1);
u1.push(-1);
u1.push(0);
u1.isMember(0);
```

After inspecting this special test, we could understand that after pushing three -1 into the stack, the stack is already full. So `push(0)` does not allow 0 to be in the stack and then invoking `isMember(0)` does not return `true`. This special test exercises a special case: a full stack. Other special tests also exercise other special cases such as empty stacks and one-element stacks.

## 4. CONCLUSION

This research proposes a novel approach of automatically identifying common and special object-oriented unit tests. The approach formalizes the notion of common and special behavior by using inferred statistical algebraic abstractions and provides an operational way for defining and identifying common and special tests. Our preliminary results show that the approach is promising to achieve the goal of identifying interesting tests. In ongoing work, we are improving the implementation and conducting more experiments.

## 5. REFERENCES


[1] 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.

[2] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[3] 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.


[1] Full details of the experimental results are described in http://www.cs.washington.edu/homes/taoxie/sabicu/