Automatically Identifying Special and Common Unit Tests for Object-Oriented Programs

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

David Notkin
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195
notkin@cs.washington.edu

Abstract

Developers often create common tests and special tests, which exercise common behaviors and special behaviors of the class under test, respectively. Although manually created tests are valuable, developers often overlook some special or even common tests. We have developed a new approach for automatically identifying special and common unit tests for a class without requiring any specification. Given a class, we automatically generate test inputs and identify common and special tests among the generated tests. Developers can inspect these identified tests and use them to augment existing tests. Our approach is based on statistical algebraic abstractions, program properties (in the form of algebraic specifications) dynamically inferred based on a set of predefined abstraction templates. We use statistical algebraic abstractions to characterize program behaviors and identify special and common tests. Our initial experience has shown that a relatively small number of common and special tests can be identified among a large number of generated tests and these identified tests expose common and special behaviors that deserve developers' attention.

1 Introduction

In unit testing of object-oriented programs, the class under test might exhibit special or common program behaviors when it is exercised by different tests. For example, intuitively a bounded-stack class exhibits common behaviors when the stack is neither empty nor full, but might exhibit some special behaviors when the stack is empty or full. Special and common tests can be created to exercise some special and common behaviors 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 behaviors of the class: developers 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 tests to exercise a variety of behaviors of the class. If the class's specifications exist, the execution of these tests can be automatically verified against the specifications. In addition, among generated tests, special or common tests can be identified based on specifications and then these identified tests can be used to augment existing manually created tests. For example, UML use cases [23] can describe high level behaviors with conditionals. Generated tests that exercise conditional behaviors can be identified as special tests. However, in practice, specifications often do not exist. Without specifications, it is impractical for developers to manually inspect and verify the outputs of such a large number of test executions. Consequently developers do not have an efficient way to identify common or special tests.

In this paper, we present a new approach for automatically identifying special and common unit tests from automatically generated tests without requiring specifications. Developers can inspect these identified tests for verifying their correctness and understanding program behavior. Developers can also use these identified tests to augment existing tests.

Our new approach is based on dynamically inferred program properties, called *statistical algebraic abstractions*. An *algebraic abstraction* is an equation that abstracts the program's runtime behaviors (usually describing interactions among method calls); the equation is syntactically identical to an axiom in algebraic specifications [14]. An *instance* of an abstraction is a test that instantiates the left-hand side (LHS) and right-hand side (RHS) of the abstraction. A *satisfying instance* is an instance that satisfies the equality relationship between LHS and RHS defined by the abstraction. A *violating instance* is an instance that violates the equality relationship. A *statistical algebraic abstraction* is associated with the counts of its satisfying and violating instances during test executions. We characterize a *common property* with a statistical algebraic abstraction whose instances are mostly satisfying instances and characterize a *universal property* with a statistical algebraic abstraction whose instances are all satisfying instances. A *conditional universal property* is a universal property whose LHS is associated with a condition. A *common test* is a satisfying instance of a common or universal property. A *special test* is a violating instance of a common property or a satisfying instance of a conditional universal property. For each common property, the first encountered violating instance is selected as a representative of the property's special tests. For each conditional universal property, the first encountered satisfying instance is selected as a representative of the property's special tests. For each common or universal property, the first encountered satisfying instance is selected as a representative of the property's common tests.

In previous work, Ernst et al. developed Daikon [12] to infer operational abstractions that describe the program states at method entry and exit points of a class. In contrast, our approach infers algebraic abstractions, which reveal no internal details of the object representation. Henkel and Diwan's approach [17] also infers algebraic abstractions but their abstractions are universally true among all test executions (so are the operational abstractions inferred by Daikon [12]). Our approach infers *statistical abstractions*, which are not necessarily universally true among all test executions.

The rest of this paper is organized as follows. Section 2 presents a illustrating example. Section 3 illustrates our new approach for identifying special and common tests based on statistical algebraic abstractions. Section 4 presents our initial experience on applying the approach. Section 5 reviews related work, and Section 6 concludes.

2 Example

As an illustrating example, we use a data structure: a LinkedList class, which is the implementation of linked lists in the Java Collections Framework, being a part of the standard Java libraries [25]. Figure 1 shows declarations of LinkedList's public methods. LinkedList has 25 public methods, 321 noncomment, non-blank lines of code, and 708 lines of code including comments and blank lines. Given the bytecode of LinkedList, our approach automatically generates a large set of tests (6777 tests); among these generated tests, our approach identifies 37 special tests and 96 common tests.

```
public class LinkedList {
  public LinkedList() {...}
  public void add(int index, Object element) {...}
  public boolean add(Object o) {...}
  public boolean addAll(int index, Collection c) {...}
  public void addFirst(Object o) {...}
  public void addLast(Object o) {...}
  public void clear() {...}
  public Object remove(int index) {...}
  public boolean remove(Object o) {...}
  public Object removeFirst() {...}
  public Object removeLast() {...}
  public Object set(int index, Object element) {...}
  public Object get(int index) {...}
  public ListIterator listIterator(intindex) {...}
  public Object getFirst() {...}
   ...
}
```

Figure 1. A LinkedList implementation

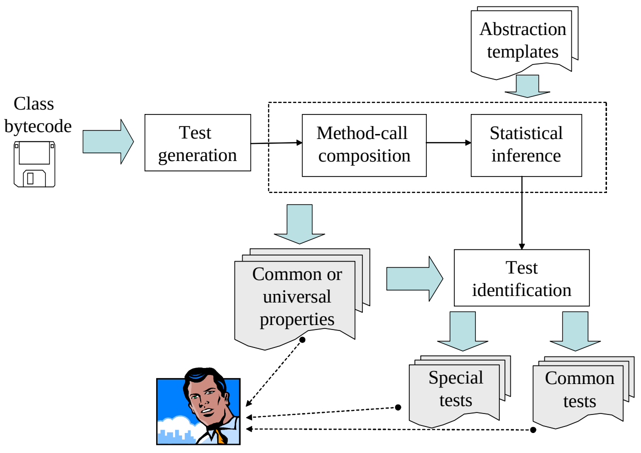

Figure 2. An overview of special and common test identification

3 Approach

Figure 2 shows the overview of our approach for identifying special and common tests. The input to our approach is the bytecode of the (Java) class under test. Our approach relies on 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 (two method calls invoked on the same receiver object in a row) 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. All these four steps are automated. The step of test generation first generates different representative argument values for each public method of the class (based on JCrasher [7], a third-party test generation tool), and then dynamically and iteratively invokes different method arguments on each nonequivalent receiver-object state (our previous work [27] 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. In the next section, we first describe predefined abstraction templates and then illustrate these four steps in details.

3.1 Abstraction Templates

Dwyer et al.'s work [10] and Ernst et al.'s work [12] develop a set of patterns and grammars for temporal properties and operational abstractions, respectively. Inspired by their work, we have developed a set of abstraction templates for algebraic abstractions. We have looked into a non-trivial set of manually written algebraic specifications from the web and found that a majority of manually written axioms are usually equations whose right-hand side (RHS) contains a constant or information related to the left-hand side (LHS). Usually an axiom's LHS or RHS involves method-call pairs besides individual method calls.

We use `f(S, args).state` and `f(S, args).retval` to represent the receiver state and method return after invoking a method `f` on a receiver with argument `args`, where the receiver state of a method call is treated as the first method argument (but a constructor does not have a receiver state). The `.state` and `.retval` expressions denote the state of the receiver (called *method-exit state*) after the invocation and the result of the invocation, respectively. We adopt the notation following Henkel and Diwan [17].

Definition 1 *A* method-call pair ⟨ `f(S, args1), g(S', args2)`⟩, *represented as* `g(f(S, args1).state, args2)`, *is a pair of a method call* `f(S, args1)` *and a method call* `g(S', args2)`, *where these two method calls are invoked in a row on the same receiver, with* `f(S, args1)` *being invoked first.*

Figure 3 shows the algebraic-abstraction templates (s0 - s11) for the method-exit state of a method-call pair or method call. We can derive the algebraic-abstraction templates (r1 - r11) for the return of a method-call pair or method call by replacing the `.state` postfix of s1 - s11 with `.retval` (but s0' and s5' do not have corresponding templates for the method return). Except for Template

s0: f(S, args1).*state*	== S
s0': f(S, args1).*state*	!= S
s1: f(S, args1).*state*	== const
s2: g(f(S, args1).*state*, args2).*state*	== args1.i
s3: g(f(S, args1).*state*, args2).*state*	== args2.i
s4: g(f(S, args1).*state*, args2).*state*	== f(S, args1).*state*
s5: g(f(S, args1).*state*, args2).*state*	== const
s5': g(f(S, args1).*state*, args2).*state*	== S
s6: g(f(S, args1).*state*, args2).*state*	== g(S, args2).*state*
s7: g(f(S, args1).*state*, args2).*state*	== f(g(S, args2).*state*, args1).*state*
s8: g(f(S, args1).*state*, args2).*state*	== f(S, args2).*state*
s9:g(f(S, args1).*state*, args2).*state*	== g(S, args1).*state*
s10:g(f(S, args1).*state*, args2).*state*	== g(f(S, args2).*state*, args1).*state*
s11:g(f(S, args1).*state*, args2).*state*	== f(g(S, args1).*state*, args2).*state*

Figure 3. Algebraic-abstraction templates for method-exit states

s0', all templates are equations. For the sake of brevity, we call these templates as equations without discriminating Template s0'. Basically Template s0 shows `f` is a *state-preserving method*, which does not modify the receiver's object state, and Template s0' shows `f` is a *state-modifying method*, which modifies the receiver's object state. In each of Templates r1, s0, s0' and s1, the LHS of the equation is a single method call. In each of the remaining templates, the LHS of the equation is a method-call pair. The RHS of an equation can be in the following forms:

- the method-entry state of the first method call in the LHS, represented as `S`, such as in Templates s0, s0' and s5'. For example, an instantiation of Template s0' is `removeFirst(S).state != S` in the LinkedList example.
- a constant, represented as `const`, such as in Templates r1, s1, r5, and s5. A constant can be `Exception`, indicating throwing an uncaught exception. For example, an instantiation of Template r1 is `add(S, m0).retval == true`.
- an argument of the first or second method call, represented as `args1.i` or `args2.i` (where `i` indicates the `i`th argument), such as in Templates r2, s2, r3, and s3. For example, an instantiation of Template r2 is `indexOf(add(S, i0_1, m1_1).state, m0_2).retval == i0_1`, where a method parameter is represented as the combination of the first letter of its runtime type name and its parameter order (starting from 0) followed by "_1" if the method is the first one in the method pair or followed by "_2" if the method is the second one.
- a method-exit state or return value of a method-call pair or method call derived from the entities of the LHS, such as the remaining templates.

There are two extensions to abstraction templates: conditional extension and difference extension. The conditional extension adds a condition for the LHS of a template. The existing implementation of our approach considers only conditions that describe the equality relationship among arguments from the first and second method calls in the LHS. The implemented conditional extensions for method-exit state are represented as:

sc1: g(f(S, args1).*state*, args2).*state*
== RHS where (args1.i == args2.j)

We similarly derive conditional extensions for method returns. For example, one instantiation of conditional extensions is

contains(add(S, m0_1).state, m0_2).retval
== true [where (m0_1==m0_2)].

In future work, we plan to support the following conditional extensions:

sc2: g(f(S, args1).*state*, args2).*state* == if (h(S)) RHS

where h is a *boolean observer*, which is a public state-preserving boolean method of the class under test. In previous work, we had used the return values of observers to abstract object states during the construction of state transition diagrams [30].

A difference extension is applicable for those templates whose LHS is a return value with a numeric type, such as `int`. The difference extension that we have implemented are represented as :

rd1: g(f(S, args1).*retval*, args2).*retval* == RHS + const

For example, one instantiation of conditional extensions is

size(add(S, m0_1).state).retval == (size(S).retval + 1)

Henkel and Diwan's inference tool [17] infers 146 axioms for `java.util.ArrayList` [18]. Our abstraction templates are sufficient to be instantiated to form all of these 146 axioms except for two axioms that describe the equivalence relationships between two methods: `add(Object o)` and `add(int index, Object element)` where `index` is 0.

3.2 Test Generation

In previous work [27], we have proposed Rostra, a formal framework for detecting equivalent object states and redundant tests. We have developed five techniques within Rostra. In this paper, we focus on the WholeState technique. The *WholeState* technique represents an object state by using the whole concrete state, which comprises the values of object fields that are reachable from the object. The technique compares object states to determine equivalence by performing a graph isomorphism algorithm on the representations. The "==" in the equations shown in Section 3.1 denotes the equivalence for object states instead of object identities (the object state of a primitive-type variable are considered as its primitive value in the string form). Our previous experience [27, 28] showed that the performance is reasonably acceptable when storing and comparing compressed explicit state information for a single class exercised by relatively short method sequences. Because sometimes two non-isomorphic object states could be equivalent as data structure instances, we can use another two techniques in Rostra that are based on user-defined `equals` methods [20] or use more expensive checking of observational equivalence [3, 9, 17].

A *method argument list* for a method call is characterized by the method signature and the arguments for the method. Two argument lists are nonequivalent iff their method signatures are different or some of their corresponding arguments are nonequivalent. A method call has two types of inputs: the method-entry state and the method argument list. A method call has two types of outputs: the normal return value and method-exit state.

We perform combinatorial test generation on the two types of inputs. We first use a third-party test generation tool, called JCrasher [7], to generate nonequivalent method argument lists. For example, JCrasher generates -1, 0, and 1 for arguments with the integer type and it can generate method sequences creating values for those arguments with non-primitive types. We provide a `MyInput` class as a helper class for JCrasher to generate values for those arguments with the `Object` type. The `MyInput` class contains an integer field `v`, whose value is set through the argument of its constructor. For example, for `add(Object o)`, three arguments can be generated: `MyInput.<init>(-1).state`, `MyInput.<init>(0).state`, and `MyInput.<init>(1).state`, where `<init>` represents a constructor method call.

We then generate tests to exercise each possible combination of encountered nonequivalent object states and nonequivalent method argument lists starting from the states after invoking constructors. In particular, we at first generate and execute tests to exercise the states after invoking constructors (the first iteration). After having executed these tests, we collect some more new nonequivalent object states that are not equivalent to any state exercised before the present iteration. Then we start the next iteration to generate more tests to exercise these new nonequivalent object states. The iterations continue until there are no new nonequivalent object states in the present iteration or we have reached the maximum iteration number. In the illustrating LinkedList example, we choose the maximum iteration number as five. The details of the test-generation algorithm have been presented in our previous work [26].

3.3 Method-Call Composition

To instantiate the LHS or RHS of most abstractions templates, we need to generate a large number of method-call pairs besides individual method calls. Traditional algebraic-specification-based testing techniques [3, 6, 9, 13, 20] generate neighboring method calls (invoked in a row) on the same receiver as method-call pairs for the LHS or RHS of an algebraic abstraction. For example, the following is a generated test called Test 1 whose line number is marked:

```
Test 1:
1 LinkedList s = new LinkedList( );
2 MyInput m = new MyInput(1);
3 s.add(m);
4 s.get(0);
5 s.size();
6 s.clear();
```

Traditional techniques generate this test to exercise four method-call pairs (each of which may be relevant to the LHS or RHS of an algebraic abstraction): $< 1, 3 >$, $< 3, 4 >$, $< 4, 5 >$, and $< 5, 6 >$, where the line number is used to represent the method call in the line. To reduce the analysis cost, we compose method calls to generate a larger number of synthesized method-call pairs from the same tests; a synthesized method-call pair exhibits the same behavior as their corresponding actual method-call pair even if the two method calls in the synthesized method-call pair are not invoked on the same receiver, or not in a row on the same receiver. We can use a synthesized method-call pair to instantiate an abstract template in the same way as an actual method-call pair.

Before we illustrate the technique of composing method calls to form synthesized method-call pairs, we first introduce the definition of a method execution, which has been informally referred to previously in the paper. We view the method calls on an object as a sequence of object states and state transitions among them. A method call transforms the receiver from the method-entry state to the method-exit state. We use a *method execution* to characterize the runtime information of a method call without considering the receiver's identity.

Definition 2 *A* method execution *⟨ m, s, S_{entry}, a, S_{exit}, r ⟩ is a tuple of a method name m, a method signature s, a method-entry state S_{entry}, method arguments a, a method-exit state S_{exit}, and a return value r. The method execution is produced by a method call m(S_{entry}, a).*

For example, Test 1 produces the following method executions:

1 ⟨ <init>, ( ), $\emptyset$, ( ), S_0, υ ⟩
2 ⟨ MyInput.<init>, (int), $\emptyset$, (1), S_{m0}, υ ⟩
3 ⟨ add, (Object), S_0, (S_{m1}), S_1, true ⟩
4 ⟨ get, (int), S_1, (0), S_2, S_{m2} ⟩
5 ⟨ size, ( ), S_2, ( ), S_3, 1 ⟩
6 ⟨ clear, ( ), S_3, (5), S_4, υ ⟩

where we use $\emptyset$ and υ to represent an empty state and a `void` return value, respectively. We use S_0, S_1, S_2, S_3, and S_4 to represent object states of `LinkedList`, and S_{m0}, S_{m1}, and S_{m2} to represent object states of `MyInput`, A constructor name is shown as `<init>`. We display the class names before method names (e.g. `MyInput` in Line 2) unless the method is of the class under test. We then generate a synthesized method-call pair based on the method-entry states and method-exit states of two method executions.

Definition 3 *A synthesized method-call pair ⟨ `f(S, args1)`, `g(S', args2)`⟩, represented as* `g(f(S, args1).state, args2)` *is a pair of a method call* `f(S, args1)` *and a method call* `g(S', args2)`, *where these two method calls produce two method executions ⟨ f, s_1, S, args1, S_{exit1}, r_1⟩ and ⟨ g, s_2, S', args2, S_{exit2}, r_2⟩, and S_{exit1} and S' are equivalent.*

From the method executions of Test 1, we can produce four synthesized method-call pairs in the same form of those four method-call pairs actually produced at runtime. In addition, we can use the WholeState technique [26] described in Section 3.2 to determine three sets of equivalent object states: $\{S_0, S_4\}$ and $\{S_1, S_2, S_3\}$. Based on the equivalence among object states, we can produce the following three additional synthesized method-call pairs from Test 1: $< 3, 5 >$, $< 3, 6 >$, $< 4, 6 >$, and $< 6, 3 >$. Note that if a method execution throws an uncaught exception, we do not put it as the first method call in a synthesized method-call pair because the method-exit state might be corrupted already.

In algebraic abstractions, the first method call in a method-call pair is usually a method call used to construct or modify the receiver's object state. Therefore for abstraction inference we do not produce synthesized method-call pairs whose first method call is of a state-preserving method. For example, from Test 1, we do not produce <4, 6> for abstraction inference. We dynamically determine whether a method is a state-modifying method. A method is a state-modifying method, if at least one of its previously observed invocations modifies the receiver's object state.

3.4 Statistical Inference

After we collect a method execution, we instantiate the template variables `f` and `args1` in the LHS of r1, s0, s0', and s1 using the method execution's method name and signature. After we generate a synthesized method-call pair, we instantiate the template variables `f`, `args1`, `g`, `args2` in the LHS of r2-11 and s2-11 using the method names and signatures in the synthesized method-call pair. Since the RHS of a template is either a constant or a combination of some variables from the LHS, we instantiate the RHS of a template using a constant or the information from the instantiated LHS. After we have instantiated the LHS and RHS of an abstraction template, we get an algebraic abstraction.

We next use the actual variable values and state representations in the method execution or synthesized method-call pair to evaluate each generated algebraic abstraction to determine whether they satisfy or violate the abstraction. Unless the RHS of an abstraction is an `Exception` constant, an exception-throwing method execution or synthesized method-call pair in the LHS always violates the abstraction. We consider the method call or method-call pairs instantiating the LHS of an abstraction (called *LHS instance*) as an *instance* of the abstraction[1]. A *satisfying instance* is an instance that satisfies the equality relationship between LHS and RHS defined by the abstraction. A *violating instance* is an instance that violates the equality relationship. We record the statistics of the abstraction satisfactions and violations by instances of the abstraction. In particular, we maintain two counters, a satisfaction counter and a violation counter, for each algebraic abstraction.

Definition 4 *A* statistical algebraic abstraction $\langle\, a, s_a, v_a \,\rangle$ *is a tuple of of an algebraic abstraction a, a count of satisfying instances s_a, and a count of violating instances v_a.*

In addition, we associate two abstraction instances with each statistical abstraction: the first-encountered satisfying instance and the first-encountered violating instance. We use these instances in test selection, which is described in the next section.

A *conditional abstraction* is an abstraction instantiated from a conditional extension of a template. We enumerate all possible conditional abstractions with different combinations of same-type arguments from two method calls in a synthesized method-call pair. A *difference abstraction* is an abstraction instantiated from a difference extension of a template. We transform a difference abstraction to the form of `LHS - RHS == const, args1.i or args2.i`.

To reduce overhead, if we have not encountered any instance that satisfies an abstraction, we do not create or store the entry of the abstraction in the memory. Therefore when the test generation and execution terminates, each abstraction in memory has at least one satisfying instance.

3.5 Identification of Special and Common Tests

After the test generation and execution terminates, we produce a list of statistical algebraic abstractions.

Definition 5 *A* universal property *is a statistical algebraic abstraction $\langle\, a, s_a, v_a \,\rangle$ without any violating instance, that is, v_a is 0. A* conditional universal property *is a universal property whose underlying abstraction is a conditional abstraction.*

[1]We can additionally consider the method call or method-call pairs instantiating the RHS of an abstraction (called *RHS instance*) as a part of the abstraction instance, but we can always derive the RHS instance given an LHS instance and the abstraction.

Definition 6 *A* common property *is a statistical algebraic abstraction with a minority of violating instances. More formally, a common property is a statistical algebraic abstraction $\langle\, a, s_a, v_a \,\rangle$, where $\frac{s_a}{s_a+v_a} \geq t$ ($50\% < t < 100\%$, and t is a user-defined threshold value close to 100%).*

We choose 80% threshold value by default in our approach.

Definition 7 *A* special test *is a violating instance of a common property, or a satisfying instance of a conditional universal property.*

Definition 8 *A* common test *is a satisfying instance of a common or universal property.*

We consider a satisfying instance of a conditional universal property to be a special test instead of a common test because the instance satisfies the condition where there exists an equality relationship between two arguments. We do not select a conditional universal property's violating instances (method call pairs where two argument values are different) as special or common tests because these instances do not exhibit behavior on the RHS of an abstraction template.

For each common property, we select the first-encountered violating instance as a representative of the property's special tests. For each conditional universal property, we select the first-encountered satisfying instance as a representative of the property's special tests. For each common or universal property, we select the first-encountered satisfying instance as a representative of the property's common tests. Since a selected test for one property might be the same as another selected test for another property, we also group those properties associated with the same test together. Developers can inspect these selected tests and their associated satisfied or violated properties.

4 Experience

We have developed a tool, called Sabicu, to prototype our approach and applied the tool on different types of applications, especially those complex data structures. We describe our initial experience on several benchmarks of complex data structures in this section. The full details of the results have been posted on our project web[2]. The first and second columns of Table 1 show the names of the benchmark programs and the number of public methods used for test generation and test identification. Most of these classes are complex data structures that are used to evaluate Korat [4] and later used to evaluate our previous work on redundant-test detection [27].

We ran Sabicu on a Linux machine with a Pentium IV 2.8 GHz processor with 1 GB of RAM running Sun's JDK

[2]http://www.csc.ncsu.edu/faculty/xie/sabicu/

Table 1. Quantitative results for identifying special and common tests

		potential		axioms	time	properties			tests			
benchmark	meth	axioms	iter	consd	(sec)	univ	c-univ	common	generated	special	common	both
BinSearchTree	4	240	3	75	0.93	6	10	7	91	6	14	3
			4	75	1.29	6	10	6	136	5	14	3
			5	75	1.30	6	10	6	136	5	14	3
BinomialHeap	12	2364	3	505	44.37	22	6	56	5272	45	63	1
			4	505	115.25	21	5	53	12440	45	61	1
			5	506	392.97	21	5	51	21456	42	59	1
FibonacciHeap	9	1242	3	290	2.06	15	6	72	173	51	59	8
			4	290	3.67	13	6	81	341	55	63	9
			5	290	7.17	12	6	80	677	52	62	7
HashMap	13	2022	3	381	15.33	81	9	19	2213	15	88	5
			4	381	61.28	81	9	18	7533	14	92	9
			5	381	163.21	81	9	19	15345	15	92	10
HashSet	8	792	3	211	1.78	43	15	18	157	15	48	7
			4	211	2.61	43	15	16	235	14	49	9
			5	211	2.91	43	15	16	261	14	50	10
LinkedList	21	6048	3	848	6.82	56	20	24	729	22	81	5
			4	848	21.62	55	18	43	2241	39	96	8
			5	848	74.01	55	18	39	6777	37	96	8
SortedList	24	7827	3	939	10.41	56	14	28	820	23	84	6
			4	939	32.61	55	14	35	2521	30	88	4
			5	939	108.35	55	14	44	7624	33	95	4
TreeMap	15	1968	3	411	20.21	84	9	20	2911	16	92	6
			4	411	82.85	84	9	18	9421	14	95	9
			5	411	295.86	84	9	17	16291	13	95	9
IntStack	4	252	3	35	0.53	5	0	2	66	2	4	2
			4	35	1.06	5	0	5	201	4	6	3
			5	35	2.29	5	0	5	606	4	6	3
UBStack	9	942	3	80	0.75	10	2	7	169	7	17	1
			4	80	0.97	10	2	7	253	7	17	3
			5	80	1.18	10	2	6	337	6	16	1

1.4.2. In particular, we ran Sabicu on the benchmarks with three different maximum iteration numbers: 3, 4, and 5 (test-generation iterations are described in Section 3.2). To avoid taking too long during one iteration, we set a time-out of five minutes for each iteration; if within five minutes Sabicu could not finish generating and running tests to fully exercise the new nonequivalent object states, we terminate the test generation. The fourth column shows the maximum iteration number where the data in the same row are produced. We compute the size of potential axioms to be explored (all possible instantiations of the abstraction templates by the public methods), which is shown in the third column of Table 1. The fifth column shows the number of axiom candidates (statistical abstractions) that our prototype considered and kept in memory during test generation and execution.

We have observed that the number of axiom candidates is not very large and they often remain stable across iterations. The sixth column shows the real time (in seconds) spent on test generation, execution, and identification. We have observed that for relatively large programs the real time grows by a factor of three to five when increasing the maximum iteration by one. Columns 7, 8, and 9 show the number of universal properties, conditional universal properties, and common properties, respectively. The last four columns show the number of all generated tests, identified special tests, identified common tests, and tests identified to be both special and common with respect to different properties, respectively. We have observed that a higher maximum iteration number (more tests) can falsify universal properties inferred from earlier iterations but usually cannot produce more universal properties because the maximum iteration number of three shall be able to instantiate all possible universal properties (described by our abstraction templates). However, the number of conditional universal properties or common properties can be increased or decreased when we increase the maximum iteration number. On one hand, a universal property can be demoted to be common properties or conditional universal properties[3]. On the other hand, a property does not have a high enough number of satisfying instances can be promoted to be a common property when more satisfying instances are generated in a higher iteration. Although the number of all generated tests increases over iterations, the number of identified special and common tests remains relatively manageable; although the absolute number of identified tests is relatively high for large benchmarks, the average number of identified tests for each method is not high.

We manually inspect identified tests and their associated properties; we especially focus on special tests. Because of space limit, we will describe only several identified tests in this section. One common property for LinkedList has 117 satisfying count and 3 violating count (instantiated from Template s7):

removeLast(addFirst(S, m0_1).state).state
 == addFirst(removeLast(S).state, m0_1).state

In the common test of this property, the LinkedList state S in the abstraction holds at least one element. But in the special test, S holds no element.

Another common property for LinkedList has 408 satisfying count and 42 violating count (instantiated from Template s7):

remove(removeLast(S).state, m0_2).state ==
 removeLast(remove(S, m0_2).state).state

In the common test of this property, the LinkedList state S in the abstraction holds only one element (being $m0\_2$). But in the special test, S holds two elements (the last element being $m0\_2$).

To investigate whether the inferred axioms are complete and consistent, we can compare them with algebraic specifications that a developer would generate. Among the 10 benchmarks, the last two benchmarks are equipped with algebraic specifications specified or inferred by other researchers. All the three axioms inferred by Henkel and Diwan [17] for IntStack (an unbounded stack storing integer elements) are also inferred by Sabicu as universal axioms. Stotts et al. [24] wrote 12 axioms for UBStack (a bounded stack storing unique integer elements)[4], eight of which are universal axioms and four of which are conditional axioms. Sabicu successfully infers all the manually written universal axioms and infers two common axioms that correspond to two of the four manually written conditional axioms.

One of the inferred common axiom for UBStack has 47 satisfying count and 6 violating count (instantiated from Template r2):

top(push(S, i0_1).state).retval == i0_1

This property shows the bounded feature of the stack implementation; if a stack is unbounded, this property would be a universal property. In the special test for this property, the UBStack state S is already full; pushing an element on a full stack does not change the stack state. Invoking `top` subsequently does not return the element that was just pushed. The corresponding manually written conditional axiom is the following:

top(push(S, i0_1).state).retval ==
 if isFull(S) then top(S) else i0_1

Another inferred common axiom for UBStack has 42 satisfying count and 11 violating count (instantiated from Template r6):

isFull(push(S, i0_1).state).retval == isFull(S).retval

This property also shows the bounded feature of the stack implementation. In the special test for this property, the UBStack state S is almost full; pushing one more element on it could produce a full stack. The corresponding manually written conditional axiom is the following:

isFull(push(S, i0_1).state).retval ==
 if getSize(B) $>=$ maxSize(B)-1 then true else false

However, we cannot infer common axioms corresponding to two manually written axioms:

pop(push(S, i0_1).state).retval ==
 if isFull(S) then pop(S) else S

and

getSize(push(S, i0_1).state).retval ==
 if (getSize(S) == maxSize(S)) then maxSize(S)
 else getSize(S) + 1

because the corresponding statistical abstractions could not get overwhelming satisfying count to be surfaced as common axioms.

Overall we have found that conditional universal properties are not too many but often indicate important interactions between two methods. For example, the uniqueness of the UBStack's elements is reflected by a conditional universal property:

push(push(S, i0_1).state, i0_2).state == push(S, i0_2).state
 [where (i0_1==i0_2)]

This property shows that pushing the same element twice in a row has the same effect of pushing the element once; this property is missing among the axioms written by Stotts et al. [24], because Stotts et al. wrote these axioms for a bounded stack in general instead of a bounded stack that stores unique elements.

We also found that some universal properties are not really universally satisfiable because the generated tests are not sufficient enough to violate them. However, we cannot afford to generate exhaustive tests with higher bound [27, 28] (reflected by the maximum iteration number). In future work, we plan to use universal properties or conditional uni-

[3]A universal property can be demoted to a conditional one because we do not infer or report a conditional universal property that is inferred by a universal property

[4]Although UBStack is one of the programs used by Stotts et al. [24], Stotts et al. wrote axioms for a bounded integer stack, which does not necessarily have the feature of storing unique elements

versal properties to guide generating a focused set of tests for these properties instead of a bounded exhaustive set.

Although we manually inspected identified tests and found that they can expose different common and special behaviors that we have not noticed before, it is still unclear whether developers (code owners) would like to invest effort in inspecting these identified tests for inclusion in their existing test suite. In future work, we plan to conduct case studies to gather feedback on how developers assess the identified tests. In addition, more experiments are needed to assess the fault detection capability of identified tests comparing to all the generated tests or those tests selected using other test selection techniques. Finally, we primarily applied our approach on data structures and we plan to investigate how generalizable our approach is beyond data structures.

5 Related Work

Our work is mainly related to three lines of research: abstraction generation (also called specification inference), statistical program analysis, and test selection.

Abstraction Generation. Ernst et al. [12] developed the Daikon tool to infer operational abstractions from test executions. Our abstraction template technique is inspired by their use of grammars in abstraction inference. Their abstractions are universal properties, whereas statistical algebraic abstractions in our approach include both universal and common properties. During the inference process, Daikon immediately throws out falsified abstractions to reduce the candidate space but our approach cannot do so because falsified abstractions may become common properties if most of all the encountered instances (including later-encountered ones) are satisfying ones. Our approach does not eliminate a potential axiom if the so-far observed weight of evidence is against it (having far more violating instances than satisfying instances), because further-generated tests may yield enough satisfying instances to make it a common property. Keeping track of statistical algebraic abstractions is more tractable than keeping track of statistical operational abstractions, because the candidate space of operational abstractions is much larger. In addition, the inference of algebraic abstractions is less sensitive to the actual values that are put and stored in a container object; therefore, our approach would not infer some less meaningful properties such as one saying that an integer of 0 is inserted into a data structure all the time or most of the time.

Henkel and Diwan developed a tool to infer algebraic specifications for a Java class [17]. Their tool generates a large number of terms, which are method sequences, and evaluates these terms to find equations, which are then generalized to axioms. Since their technique does not rely on abstraction templates, their technique is able to infer more types of abstractions than the ones predefined in our approach. For example, their technique can infer an abstraction whose RHS contains a method call that is not present in the LHS. However, their inferred abstractions are all universal properties, containing no common properties. Their tool does not support conditional abstractions. Their later work [19] developed an interpreter for the algebraic specifications of a Java class, and this interpreter acts like a prototype implementation for the class. The abstractions inferred by either their earlier tool or our tool can be fed into this interpreter for debugging algebraic specifications.

Statistical Program Analysis. Different from the preceding abstraction inference techniques, Ammons et al. infer protocol specifications for a C application program interface by observing frequent interaction patterns of method calls [1]. Their inferred protocol specifications are either common or universal properties. They identify those executions that violate the inferred protocol specifications for inspection. Both their and our approaches use statistical techniques to infer frequent behavior. Their approach operates on protocol specifications, whereas our approach operates on algebraic specifications. Their later work [2] uses concept analysis to automatically group the violating executions into highly similar clusters. They found that by examining clusters instead of individual executions, developers can debug a specification with less work. Our approach selects one representative test from each subdomain defined by statistical algebraic abstractions, instead of presenting all violating or satisfying tests to developers. This can also reduce the inspection effort for a similar reason.

Engler et al. [11] infer bugs by statically identifying inconsistencies from commonly observed behavior. We dynamically identify special tests, which might expose bugs, based on deviations from common properties. Liblit et al. [21] use remote program sampling to collect dynamic information of a program from executions experienced by end users. They use statistical regression techniques to identify predicates that are highly correlated with program failures. In our approach, we use statistical inference to identify special tests and common tests.

Test Selection. In partition testing [22], a test input domain is divided into subdomains based on some criteria, and then we can select one or more representative inputs from each subdomain. Our approach is basically a type of partition testing. We divide test input domain for a method-call pair or method call into subdomains based on each inferred statistical algebraic abstraction: satisfying tests and violating tests.

When a priori specifications are provided for a program, Chang and Richardson use specification coverage criteria to select a candidate set of test cases that exercise new aspects of the specification [5]. Given algebraic specifications a priori, several testing tools [3, 6, 9, 13, 20] generate and select a set of tests to exercise these specifications. Unlike these black-box approaches, our approach does not require specifications a priori. Indeed, after specifications are written, a specification-based testing approach can be fully automated, whereas our approach involves human inspection of the selected tests. We plan to investigate investment tradeoffs of these approaches and the combination of two approaches, because human-written specifications might not be complete to capture different behaviors of a program.

Harder et al.'s operational difference approach [16], Hangal and Lam's DIDUCE tool [15], and the operational violation approach in our previous work [29] select tests based on a common rationale: selecting a test if the test exercises a certain program behavior that is not exhibited by previously executed tests. The approach in this paper is based on a different rationale: selecting a test as a special test if the test exercises a certain program behavior that is not exhibited by most other tests; selecting a test as a common test if the test exercises a certain program behavior that is exhibited by all or most other tests. Our approach is less sensitive to the order of the executed tests than these previous approaches. In addition, these three previous approaches operates on inferred operational abstractions [12], whereas our approach operates on inferred algebraic specifications.

Dickinson et al. [8] use clustering analysis to partition executions based on structural profiles, and use sampling techniques to select executions from clusters for observations. Their experimental results show that failures often have unusual profiles that are revealed by cluster analysis. Although our approach shares a similar rationale with their approach, our approach operates on black-box algebraic abstractions instead of structural behavior.

6 Conclusion

We have proposed a new approach for automatically identifying special or common tests out of a large number of automatically generated tests. The approach is based on statistically true (not necessarily universally true) program properties, called statistical algebraic abstractions. We have developed a set of abstraction templates, which we can instantiate to form commonly seen axioms in algebraic specifications. Based on the predefined abstraction templates, we perform a statistical inference on collected method calls and method-call pairs to obtain statistical algebraic abstractions. We have developed a way to characterize special or common tests based on statistical algebraic abstractions. We sample and select special tests and common tests together with their associated abstractions for inspection. Our initial experience has shown that those tests and properties identified by our approach exposed many common and special behaviors that deserve developers' attention.

Acknowledgments

We thank Darko Marinov and the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work was supported in part by the National Science Foundation under grant ITR 0086003 and the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] G. Ammons, D. Mandelin, R. Bodik, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2003.

[3] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[4] 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.

[5] J. Chang and D. J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.

[6] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7(3):250–295, 1998.

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

[8] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proc. 8th ESEC/FSE*, pages 246–255, 2001.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420, 1999.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. 18th ACM symposium on Operating Systems Principles*, pages 57–72, 2001.

[12] 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.

[13] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[14] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering*, pages 291–301, 2002.

[16] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. 25th International Conference on Software Engineering*, pages 60–71, 2003.

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

[18] J. Henkel and A. Diwan. Case study: Debugging a discovered specification for java.util.ArrayList by using algebraic interpretation. Technical Report CU-CS-970-04, University of Colorado at Boulder, 2004.

[19] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proc. 26th International Conference on Software Engineering*, pages 449–458, 2004.

[20] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for oo programs in the presence of side-effects. In *Proc. International Symposium on Software Testing and Analysis*, pages 53–61, 1996.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[22] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[23] OMG. Unified Modeling Language Specification (v1.5), 2003.

[24] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.

[25] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[26] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.

[27] 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.

[28] 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.

[29] 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.

[30] 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.