Automatically Identifying Special and Common Unit Tests Based on Inferred Statistical Algebraic Abstractions

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

Technical Report UW-CSE-04-08-03
August 2004

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 or even common test inputs. 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 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 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 and identify special and common tests. Our initial experience has shown that many interesting test inputs could be identified among a large number of generated tests.

1 Introduction

In unit testing, the class under test might exhibit special and common 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. Special and common test inputs can be created to exercise some special and common 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, special and common 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 special and common object-oriented unit tests from automatically generated tests without requiring specifications. Programmers can inspect these identified tests for verifying their correctness and understanding program behavior. They can use these identified tests to augment existing tests.

Our new approach is based on dynamically inferred program properties, called *statistical algebraic abstractions*. Different from previous work on dynamic property inference [12, 17], statistical algebraic abstractions inferred by our approach are not necessarily universally true among all test executions; a *statistical algebraic abstraction* is associated with the counts of its satisfying and violating instances during test executions. The abstraction is an equation that abstracts the program's runtime behavior (usually describing interactions among method calls); the equation is syntactically identical to an axiom in algebraic specifications [14]. 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. Then, for each common property,

```
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

we sample and select a special test (violating instance) and a common test (satisfying instance). For each universal property, we sample and select a common test (satisfying instance). Programmers can inspect both the selected tests and their associated properties.

The rest of this paper is organized as follows. Section 2 presents a nontrivial 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 nontrivial 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 [22]. Figure 1 shows declarations of LinkedList's public methods. This implementation uses doubly-linked, circular lists that have a `size` field and a `header` field, which acts as a sentinel node. It inherits a `modCount` field from a super class `AbstractList`; this field records the number of times the list has been structurally modified. 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 generate a large set of tests (10025 tests); among these generated tests, our approach identifies 26 special tests and 62 common tests. These identified tests are associated with 30 universal properties and 52 common properties.

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

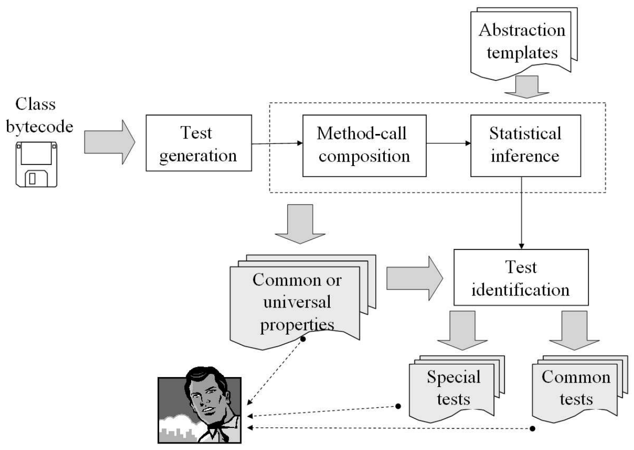

Figure 2. An overview of special and common test identification

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 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 (based on JCrasher [7], a third-party test generation tool), and then dynamically and iteratively invokes different method arguments on each non-equivalent receiver-object state (our previous work [24] 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 develop 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 left-hand side (RHS) contains a constant or information related to the right-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 of a method call is treated as the first method argument (but a constructor does not have a receiver). 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.*

`g(f(S, args1).state, args2).retval` is the same as `g(S′, args2).retval` and `g(f(S, args1).state, args2).state` is the same as `g(S′, args2).retval`.

Figure 3 show 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', all templates are equations. For the sake of brevity, we call these templates as equations without discriminating Template 0'. 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 Template 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 the following forms:

- the method-entry state of the first method call in the LHS, represented as `S`, such as in Template s0, s0' and s5'. For example, an instantiation of Template s0' is `removeFirst(S).state != S` in the LinkedList example.

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

- a constant, represented as `const`, such as in Template r1, s1, r5, and s5. A constant can be `Exception`, indicating throwing an uncaught exception. For example, an instantiation of Template r5 is `add(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 Template r2, s2, r3, and s3. For example, an instantiation of Template s0' 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.
- an 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
== add(contains(S, m0_2).state, m0_1).retval
              [where (m0_1==m0_2)].
```

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

c5: g(f(S, args1).*state*, args2).*retval* == if (h(S)) RHS

where h is a state-preserving public 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 [26].

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)`.

3.2 Test Generation

In previous work [24], we propose a formal framework for detecting equivalent object states and redundant tests. We develop five techniques within this framework. 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 (except that the LHS and RHS of the "==" are of primitive types).

A *method argument list* for a method call[1] is characterized by the method signature and the arguments for the method. Two argument lists are non-equivalent iff their method signatures are different or some of their corresponding arguments are non-equivalent. A method call has two types of inputs: 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 non-equivalent non-equivalent 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`.

We then generate tests to exercise each possible combination of encountered non-equivalent object states and non-equivalent 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 might collect some more new non-equivalent 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 non-equivalent object states. The iterations continue until there are no new non-equivalent 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 [23].

[1]When we test a class, we in fact test the interface provided by the class. The interface usually contains the public methods of the class. In this paper, we focus on public method calls that are invoked from outside the class.

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 [3,6,9,13,19] extract 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 extract four method-call pairs: $< 1, 3 >$, $< 3, 4 >$, $< 4, 5 >$, 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 transits 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_m, 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. 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 extracted by traditional techniques. In addition, we can use the WholeState technique [23] described in Section 3.2 to determine three sets of equivalent object states: $\{S_0\}$, $\{S_1, S_2, S_3\}$, and $\{S_4\}$[2]. Based on the equivalence among object states, we can produce the following three additional synthesized method-call pairs from Test 1: $< 3, 5 >$, $< 3, 6 >$, and $< 4, 6 >$. 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.

[2]Because the value of `modCount` in S_4 is different from the one in S_0, the WholeState technique determined them not to be equivalent; however our other techniques based on an equals method [23] or observational equivalence [9, 17], we can determine S_0 and S_4 to be equivalent.

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> 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 *abstraction instance*[3]. We record the statistics of the abstraction satisfactions and violations by abstraction instances. In particular, we maintain two counters, a satisfaction counter and a violation counter, for each algebraic abstraction.

Definition 4 *A* statistical algebraic abstraction *⟨ a, s_a, v_a ⟩ 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.

[3]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.

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. We select special tests and common tests based on these abstractions. Before we introduce the definitions of a special test and common test, we first present the definitions of a universal property and a common property. Intuitively a universal property is a statistical algebraic abstraction without any violating instance and a common property is a statistical algebraic abstraction with a minority of violating instances.

Definition 5 *A* universal property *is a statistical algebraic abstraction* $\langle\ a, s_a, v_a\ \rangle$*, where* $\frac{s_a}{s_a+v_a} == 100\%$*.*

For example, our approach identifies the following universal property with 336 satisfying count and 0 violating count (instantiated from an difference extension of Template r5):

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

This universal property shows that invoking `add(Object o)` always increases the list size.

Definition 6 *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. For example, our approach identifies a common property with 11 satisfying count and 1 violating count (instantiated from Template r6):

contains(clear(S).state, m0_2).retval == contains(S, m0_2).retval

This common property shows that when we invoke `clear` and then invoke `contains` to see whether the LinkedList contains an element $m0\_2$, the return value (in fact being `false`) is mostly equal to the return value of directly invoking `contains` with the same element $m0\_2$. In the violating instance, the initial state S contains the element $m0\_2$ so the LHS is true.

As another example, our approach identifies the following common property with 174 satisfying count and 5 violating count (instantiated from Template s0'):

removeFirst(S).state != S

This common property shows that invoking `removeFirst` modifies the receiver's state most of the time but not always. In fact, when we look into the violating instance: removeFirst(<init>().state).state, the instance throws an NoSuchElementException exception (recall that our approach consider an uncaught-exception-throwing method call as violating any abstraction unless the RHS is an exception constant).

When the underlying abstraction of a universal property is a conditional abstraction, the property is called a conditional universal property. For example, a conditional universal properties identified by our approach have 672 satisfying count (instantiated from Template s8):

set(add(S, i0_1,m1_1).state, i0_2,m1_2).state
== add(S, i0_2,m1_2).state [where (i0_1==i0_2)]

This property shows that after we add an element $m1\_1$ to a specific index of the LinkedList and then set the same index with another element $m1\_2$, the resulting state is equivalent to the one resulting from directly adding the element $m1\_2$ to the index.

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.

For each common property, we select the first-encountered violating instance as the representative of the property's special tests. For each conditional universal property, we select the first-encountered satisfying instance as the representative of the property's special tests. For each common or universal property, we select the first-encountered satisfying instance as the 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. Programmers 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 de-

Table 1. Quantitative results for identifying special and common tests

		axiom		axioms	time	properties			tests			
subject	meth	space	iter	consd	(sec)	univ	c-univ	common	generated	special	common	both
BinSearchTree	4	240	3	91	0.85	6	8	6	91	5	11	1
			4	91	1.22	6	8	5	136	4	11	1
			5	91	1.22	6	8	5	136	4	11	1
BinomialHeap	12	2364	3	515	3.11	20	4	44	512	27	39	2
			4	515	12.34	17	4	52	1865	31	42	2
			5	515	37.55	17	4	46	4749	28	40	2
FibonacciHeap	9	1242	3	289	1.30	13	8	54	110	33	39	3
			4	289	1.91	12	3	41	173	27	32	3
			5	289	3.60	9	3	56	341	32	34	3
HashMap	13	2022	3	467	17.83	66	8	98	2605	44	88	6
			4	467	82.61	61	8	90	10137	41	85	10
			5	467	449.42	61	8	82	17277	38	84	10
HashSet	8	792	3	222	2.21	38	11	46	235	20	44	2
			4	222	4.46	37	12	35	469	18	43	5
			5	224	6.98	31	12	20	729	15	41	5
LinkedList	21	6048	3	682	7.48	72	21	46	1009	21	88	1
			4	724	26.04	41	16	69	3249	42	82	0
			5	729	93.10	30	15	37	10025	26	62	0
SortedList	24	7827	3	701	12.17	69	12	55	1135	28	88	1
			4	744	40.89	40	10	59	3655	32	73	0
			5	749	146.96	30	9	60	11278	28	61	0
TreeMap	15	1968	3	535	22.64	67	8	97	3331	44	89	6
			4	535	100.10	62	8	98	12751	44	89	10
			5	535	421.83	62	8	83	17191	40	86	10
IntStack	4	252	3	33	0.49	2	0	2	76	2	3	1
			4	33	1.17	2	0	5	241	4	5	2
			5	33	2.97	2	0	5	766	4	5	2
UBStack	10	1077	3	115	0.71	11	1	5	183	5	16	0
			4	115	0.96	11	1	5	274	5	16	2
			5	115	1.17	11	1	4	365	4	15	0

scribe 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[4]. The first and second columns of Table 1 show the name 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 [24].

We ran Sabicu on a Linux machine with a Pentium IV 2.8 GHz processor with 1 GB of RAM running Sun's JDK 1.4.2. In particular, we ran Sabicu on the benchmarks with three different maximum iteration numbers: 3, 4, and 5. To avoid taking too long during one iteration, we set a timeout 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, the We estimate the size of axiom space to explore based on the number of methods and the number of abstraction templates. The third column of Table 1 shows our estimation. The fourth column shows the maximum iteration number where the data in the same row are produced. 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 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 to be around three to five times when setting one more maximum iteration. Columns 7, 8, and 9 show the number of universal properties, conditional universal properties, and common properties, respectively. The last four columns show the

[4]http://www.cs.washington.edu/homes/taoxie/sabicu/

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[5]. 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 interesting identified tests that we observed during inspection in this section. One common property for LinkedList has 171 satisfying count and 21 violating count (instantiated from Template s6):

addFirst(remove(S, m0_1).state, m0_2).state ==
addFirst(S, m0_2).state [where (m0_1==m0_2)]

In the common test of this property, the LinkedList state S in the abstraction does not hold the element to be removed ($m0\_1$ or $m0\_2$). But in the special test, S holds the element to be removed.

Another common property for LinkedList has 204 satisfying count and 21 violating count (instantiated from Template r5):

contains(remove(S, m0_1).state, m0_2).retval == false
[where(m0_1==m0_2)]

In the common test of this property, the LinkedList state S in the abstraction hold no or only one element to be removed ($m0\_1$ or $m0\_2$). But in the special test, S holds more than one (same) element to be removed. This property shows that LinkedList can hold multiple equivalent elements unlike a set (the property would be a universal property for a set implementation).

One common property for UBStack, a bounded stack storing unique elements, has has 47 satisfying count and 6 violating count (instantiated from Template r5):

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

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 (that does not exist in the stack already) on a full stack does not change the stack state. Invoking `isMember` with the same element as the argument does not get a `false` return value.

We have found that conditional universal properties are not too many but often indicate interesting and important interactions between two methods. Indeed, even without using our approach, programmers can use heuristics for generating tests to exercise two neighboring method calls whose arguments share the same type. However, our approach can help find most interesting call pairs among them automatically. 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 (reflected by the maximum iteration number). In future work, we plan to use universal properties or conditional universal properties to guide generating a narrowed set of tests for these properties instead of a bounded exhaustive set.

Although we manually inspected identified tests and found many interesting behaviors exposed by them, it is still unclear how well these identified tests can detect faults. In future work, we plan to do experiments to assess the fault detection capability of identified tests comparing to all the generated tests or those tests selected using other test selection techniques.

5 Related Work

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

5.1 Abstraction Generation

Ernst et al. [12] develop 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 contain both universal and common properties. 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.

Henkel and Diwan develop a tool to infer algebraic specifications for a Java class [17]. Their tool generates a large

[5]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

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 [18] develops 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.

5.2 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, programmers 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 programmers. 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. [20] 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.

5.3 Test Selection

In partition testing [21], 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, 19] generate and select a set of tests to exercise these specifications. Unlike these black-box approaches, our approach does not require specifications a priori.

Harder et al.'s operational difference approach [16], Hangal and Lam's DIDUCE tool [15], and the operational violation approach in our previous work [25] 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. Different from these previous approaches, our approach is not sensitive to the order of the executed tests. 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 and 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 develop 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 develop a way to characterize special and 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 interesting cases.

Acknowledgments

We thank Darko Marinov for providing Korat benchmarks and valuable feedback on an earlier version of the paper. We acknowledge support through 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. A tool for writing and debugging algebraic specifications. In *Proc. 26th International Conference on Software Engineering*, pages 449–458, 2004.

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

[20] 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.

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

[22] 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/`.

[23] 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.

[24] 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.

[25] 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.

[26] 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.