Automatic Identification of Common and Special Object-Oriented Unit Tests

Tao Xie
taoxie@cs.washington.edu
ACM Member Number: CR80884

Advisor: David Notkin
Department of Computer Science & Engineering
University of Washington, Seattle, WA 98195, USA

1. INTRODUCTION

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

In this paper, we present a new approach for automatically identifying 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 [9, 14], 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 [11]. We dynamically infer statistical algebraic abstractions by instantiating a set of predefined abstraction templates with 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. Then, for each common property, 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.

2. RELATED WORK

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

2.1 Abstraction Generation

Ernst et al. [9] developed the Daikon tool to infer operational abstractions from test executions. Our abstraction inference technique based on abstraction templates 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 developed a tool to infer algebraic specifications for a Java class [14]. 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 equation abstraction whose right hand side (RHS) contains a method call that

is not present in the left hand side (LHS). However, their inferred abstractions are all universal properties, containing no common properties. Their tool does not support conditional abstractions.

2.2 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. [8] 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. [16] 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.

2.3 Test Selection

In partition testing [17], 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 [4]. Given algebraic specifications a priori, several testing tools [10, 3, 7, 15, 5] 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 [13], Hangal and Lam's DIDUCE tool [12], and the operational violation approach in our previous work [20] 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 [9], whereas our approach operates on inferred algebraic specifications.

```
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

Dickinson et al. [6] 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.

3. 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 [18]. Figure 1 shows the 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 generate a large set of tests (6777 tests); among these generated tests, our approach identifies 29 special tests and 79 common tests. These identified tests are associated with 43 universal properties and 45 common properties.

4. APPROACH

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

`add(S, m0).retval == true` and

`indexOf(add(S, i0_1, m1_1).state, m0_2).retval == i0_1`

where the state of a method call's receiver is treated as the first method argument (but a constructor does not have a receiver) and the `.state` and `.retval` expressions denote the state of the receiver after the invocation and the result of the invocation, respectively. We adopt the notation following Henkel and Diwan [14].

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

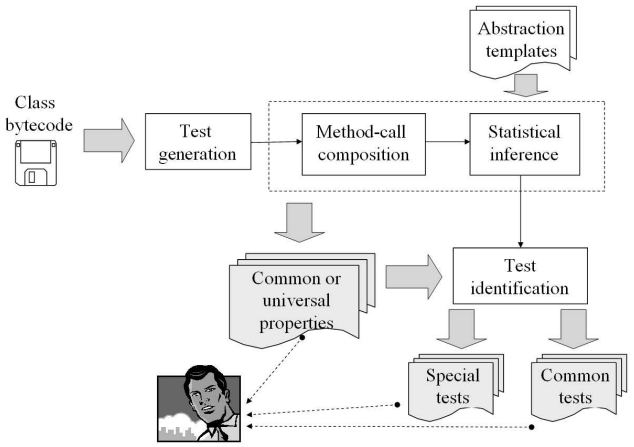

Figure 2: An overview of common and special test identification

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

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

When the underlying abstraction of a universal property is a conditional abstraction (an abstraction whose LHS is associated with a condition), the property is called a *conditional universal property*. For example, a conditional universal property identified by our approach has 474 satisfying counts:

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)]

Special tests additionally include a satisfying instance of a conditional universal property.

Figure 2 shows an overview of our approach. The input to our approach is the bytecode of the (Java) class under test and a set of algebraic abstraction templates pre-defined by us; these templates encode common forms of axioms in algebraic specifications: equality relationships among two neighboring method calls and single method calls (the details of the templates are described in an accompanying technical report [21]). The outputs of the approach are a set of common and special tests and their corresponding properties.

The approach comprises four steps: test generation, method-call composition, statistical inference, and test identification. The step of test generation first generates different representative argument values for each public method of the class, and then dynamically and iteratively invokes different method arguments on each non-equivalent receiver-object state (our previous work [19] develops techniques for determining object-state equivalence). The tests generated in Iteration N have the method call length of N, which is the number of method calls on the object under test after the constructor method call. 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.

5. EVALUATION

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[1]. 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 our previous work on redundant-test detection [19].

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, we terminate the test generation and execution within that iteration. 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 by a factor of three to five 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 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 fal-

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

Table 1: Quantitative results for identifying special and common tests

subject	meth	axiom space	iter	axioms consd	time (sec)	properties univ	c-univ	common	tests generated	special	common	both
BinSearchTree	4	240	3	75	0.93	6	7	7	91	6	14	3
			4	75	1.32	6	7	6	136	5	14	3
			5	75	1.32	6	7	6	136	5	14	3
BinomialHeap	12	2364	3	501	44.36	20	3	52	5272	42	57	1
			4	501	119.78	17	4	51	12440	44	56	1
			5	502	371.23	16	4	53	19888	43	54	1
FibonacciHeap	9	1242	3	287	1.97	21	5	45	173	32	53	4
			4	287	3.59	18	5	53	341	37	55	4
			5	287	7.02	17	4	56	677	39	55	4
HashMap	13	2022	3	381	16.25	73	8	19	2213	15	79	5
			4	381	65.11	73	8	21	7533	17	86	9
			5	381	157.59	73	8	22	15345	18	86	10
HashSet	8	792	3	211	1.85	39	11	17	157	15	45	7
			4	211	2.65	39	11	16	235	14	46	9
			5	211	3.12	39	11	18	261	15	47	10
LinkedList	21	6048	3	796	6.80	44	14	21	729	20	68	3
			4	797	21.88	43	14	33	2241	30	80	9
			5	797	76.71	43	14	31	6777	29	79	8
SortedList	24	7827	3	877	10.19	45	10	24	820	21	70	4
			4	878	33.14	44	10	23	2521	20	70	3
			5	878	110.78	44	9	31	7624	23	74	3
TreeMap	15	1968	3	409	20.31	74	8	20	2911	16	81	6
			4	409	79.94	74	8	21	9421	17	87	9
			5	409	314.46	74	8	20	15991	16	87	9
IntStack	4	252	3	33	0.57	2	0	2	76	2	3	1
			4	33	1.21	2	0	5	241	4	5	2
			5	33	2.84	2	0	5	766	4	5	2
UBStack	10	1077	3	87	0.78	11	1	6	183	6	17	1
			4	87	1.01	11	1	6	274	6	17	3
			5	87	1.23	11	1	5	365	5	16	1

sify 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 (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). 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 117 satisfying count and 3 violating count:

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 315 satisfying count and 45 violating count:

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 $m0\_2$).

One common property for `UBStack`, a bounded stack storing unique elements, has has 47 satisfying count and 6 violating count:

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.

6. CONCLUSION

We have proposed a new approach for automatically identifying special and common tests out of a large number of automatically generated tests, without requiring specifications. 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 evaluation has shown that those tests and properties identified by our approach exposed cases that shall be interesting for programmers to inspect and augment to the existing tests.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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/`.

[19] 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.

[20] 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.

[21] T. Xie and D. Notkin. Automatically identifying special and common unit tests based on inferred statistical algebraic abstractions. Technical Report UW-CSE-04-08-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, August 2004.