OCAT: Object Capture based Automated Testing

Hojun Jaygarl
Department of Computer Science
Iowa State University
Ames, IA, USA
jaygarl@cs.iastate.edu

Sunghun Kim
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
hunkim@cse.ust.hk

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

Carl K. Chang
Department of Computer Science
Iowa State University
Ames, IA, USA
chang@cs.iastate.edu

ABSTRACT

Testing object-oriented (OO) software is critical because OO languages are commonly used in developing modern software systems. In testing OO software, one important and yet challenging problem is to generate desirable object instances for receivers and arguments to achieve high code coverage, such as branch coverage, or find bugs. Our initial empirical findings show that coverage of nearly half of the difficult-to-cover branches that a state-of-the-art test-generation tool cannot cover requires desirable object instances that the tool fails to generate. Generating desirable object instances has been a significant challenge for automated test-generation tools, partly because the search space for such desirable object instances is huge, no matter whether these tools compose method sequences to produce object instances or directly construct object instances. To address this significant challenge, we propose a novel approach called Object Capture based Automated Testing (OCAT). OCAT captures object instances dynamically from program executions (e.g., ones from system testing or real use). These captured object instances assist an existing automated test-generation tool, such as a random testing tool, to achieve higher code coverage. Afterwards, OCAT mutates collected instances, based on observed not-covered branches. We evaluated OCAT on three open source projects, and our empirical results show that OCAT helps a state-of-the-art random testing tool, Randoop, to achieve high branch coverage: on average 68.5%, with 25.5% improved from only 43.0% achieved by Randoop alone.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing), Monitors*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'10, July 12–16, 2010, Trento, Italy.
Copyright 2010 ACM 978-1-60558-823-0/10/07 ...$10.00.

General Terms

Experimentation, Reliability, Verification

Keywords

Object Capturing, Object Generation, Object Mutation, Automated Testing

1. INTRODUCTION

Object-oriented (OO) languages are frequently used, and conducting effective testing of OO software is critical to discover existing bugs in the software. Among different types of testing, unit testing has been widely adopted in practice as an important means of assuring software correctness. To reduce manual effort in OO unit testing, automated test-generation tools [26, 28, 33, 37] generate test inputs for a unit (e.g., a class). These test inputs are in the form of method invocations where (1) primitive values, such as integer values, for primitive-type method arguments are automatically generated, and (2) object instances for receivers and non-primitive-type method arguments are automatically generated by composing method-invocation sequences, in short as method sequences.

In unit testing of OO software, one important and yet challenging problem is to generate desirable object instances for receivers and arguments to achieve high code coverage (such as branch coverage) or find bugs. Automatic generation of object instances becomes more difficult and complicated than testing software in procedural languages such as C. For example, the fields of object instances often cannot be directly constructed, but are indirectly constructed via method sequences due to information hiding. Two main types of techniques used by existing automated test-generation tools attempt to address this challenge: direct object construction and method-sequence generation.

Techniques that perform direct object construction, such as Korat [8], directly assign values to object fields of the object instance under construction. If an object field of the object instance is also a non-primitive type (i.e., requiring an object instance), the techniques further construct an object instance for this object field and assign values to object fields of this second-level object instance. Such a procedure is conducted until object-field values of the object instance are assigned to a small pre-defined bound of levels. To avoid generating invalid object instances (e.g., tree object in-

```
80 /**
81  *
82  * @param doc
83  * @param algorithmURI is the URI
       of the algorithm as String
84  */
85 public Algorithm(Document doc,
      String algorithmURI) {
86
87    super(doc);
88
89    this.setAlgorithmURI(algorithmURI);
90 ...
```

Figure 1: Motivating Example - `doc` is insufficient and and the execution of Line 87 throws exceptions. Line 89 cannot be reached unless a desirable `doc` object instance is passed in. The code fragment is from `org.apache.xml.security.algorithms.Algorithm`.

stances with cycles among tree nodes), these techniques require the specification of class invariants for checking and filtering invalid object instances; however, in practice, class invariants are rarely documented. In addition, these techniques require the value domain (i.e., the set of selected values) for each primitive-type object field be manually specified. These factors cause these techniques to be ineffective for testing real-world classes. For example, Korat [8] was evaluated on primarily data structures and applying it requires much manual effort for preparing class invariants and value domains.

The techniques of method-sequence generation [11, 18, 26, 35] produce method sequences that can produce an object instance. For example, random techniques [11, 26] generate random method sequences, sometimes by pruning, based on feedback from previously generated sequences. The evolutionary techniques [18, 35] use genetic algorithms to evolve initial method sequences to ones more likely to cover target branches (e.g., not-yet-covered branches). Because the search space for method sequences is huge, the random technique or the evolutionary technique (whose sequence construction is largely random) is often ineffective in finding method sequences that produce desirable object instances to cover target branches.

The main cause of low code coverage achieved by these tools is the incapability to generate desirable object instances by the preceding types of techniques. For example, Figure 1 shows a not-covered area of an open source project called Apache XML Security[1] after running a state-of-the-art random testing tool, Randoop [26], for 80 minutes. We observe the target constructor `Algorithm` takes a `Document` argument. The first statement `super(doc)` in the constructor body checks the validity of the `Document` argument. Since Randoop fails to generate a desirable (i.e., valid) `Document` object instance, the `super(doc)` throws an exception, and thus the lines after Line 87 in the constructor body remain not-covered.

To address the limitations of previous techniques on generating desirable object instances, we propose a novel approach called Object Capture based Automatic Testing (OCAT). OCAT has three main phases: (1) object capturing, (2) object generation, and (3) object mutation.

In the object-capturing phase, OCAT captures object instances from normal program executions (e.g., ones from system testing or real use). Suppose that we want to test the Eclipse program. It is generally hard to automatically generate desirable object instances; however, if we run and use Eclipse, during execution, there are many object instances being created in the memory heap. Since these object instances reflect real usage, capturing and exploiting them in automated testing could provide potential for being desirable in achieving new branch coverage.

In the object-generation phase, OCAT generates new object instances using a method-sequence generation technique [11, 18, 26, 35] and captured object instances.

Finally, OCAT mutates object instances to satisfy the conditions of not-covered branches. Although a state-of-the-art of test-generation technique is assisted by captured object instances to cover more branches, it is possible there are still not-yet-covered branches. OCAT analyzes the conditions related to not-yet-covered branches and mutates the captured object instances to satisfy the conditions.

OCAT can be viewed as a novel integration of the two preceding types of techniques for generating desirable object instances. The capturing and mutating of object instances in OCAT can be viewed as a type of direct object construction but it does not suffer from the limitations of previous techniques such as Korat [8]. For example, OCAT does not require class invariants but constructs valid object instances by capturing them from normal program executions. OCAT can construct object instances of very large size beyond object instances of size within a small bound. In addition, OCAT mutates captured object instances towards target branches, whereas Korat does not exploit guidance towards the target branch.

OCAT's integration with existing test-generation tools can be viewed as a technique of method-sequence generation since the used test-generation tools explore and generate new method sequences invoked upon the captured object instances. The captured object instances often reflect desirable object instances. Even when the captured instances are not immediately desirable, these are close to desirable object instances and, therefore, invoking method sequences can help derive desirable instances from the object instances.

This paper makes the following main contributions:

- A novel approach, OCAT, captures, generates, and mutates object instances from normal program executions and feeds them to existing test-generation tools for method-sequence generation. OCAT can be viewed as a novel integration of two existing types of techniques for generating desirable object instances: direct object construction and method-sequence generation.
- An implementation of OCAT (for testing Java programs) that is integrated with Randoop [26], a state-of-the-art random testing tool.
- An empirical evaluation of OCAT on three open source projects such as Apache Commons Collections, Apache XML Security, and JSAP (with 114,000 LOC in total). The results show that using captured object instances improves the branch coverage 21.6% on average in comparison to Randoop. Additionally, mutated object instances improve 3.9% of branch coverage on average. All together, OCAT improves 25.5% of branch coverage over Randoop alone.

The paper is organized as follows. Section 2 illustrates challenges in OO testing for motivating our OCAT approach. Section 3 presents our approach. Section 4 describes our empirical evaluation. Section 5 discusses issues of our approach. Section 6 discusses related works and Section 7 concludes.

[1] `http://santuario.apache.org/`

Table 1: Categories of main causes for not-covered branches from 10 source files in three open source projects

Cause of not-covered branches	# of branches (%)
Insufficient object	135 (46.3%)
String comparison	61 (20.9%)
Container object access	39 (13.4%)
Array comparison[6]	25 (8.6%)
Exception branches	18 (6.1%)
Environmental setting	9 (3.1%)
Non-deterministic branch	4 (1.3%)

2. CHALLENGES IN STRUCTURAL TESTING

In this section, we discuss the main challenges that we empirically observe for a state-of-the-art random testing tool, called Randoop [26], to achieve structural coverage such as branch coverage. The analysis of these challenges helps motivate our OCAT approach. We select Randoop in this preliminary empirical study, for two main reasons: (1) Randoop can be applied on any real-world code base in a totally automatic fashion without any manual effort; (2) Randoop has been shown to outperform systematic and pure random test-generation tools in terms of achieved branch coverage [26].

We run the extended version of Randoop [20] [2] on Apache Commons Collections[3], Apache XML Security[4], and JSAP[5] (details in Section 4.1), until either achieved branch coverage levels off without much further increase, or Randoop runs out of memory and cannot continue to run. Then, we randomly select 10 source code files from each subject, and manually investigate the causes of not-covered branches.

Table 1 shows categories of main causes of not-covered branches by Randoop. The main cause of not-covered branches is that Randoop is unable to generate desirable object instances required to cover certain branches. Nearly 50% of the not-covered branches in this study are due to this cause. As shown in Figure 1, the generated object instance '`doc`' does not satisfy the desirable condition for covering the branches in `Algorithm`.

The second main cause is *string comparison*. Most string comparisons consist of simple constraints such as equality (e.g., `equals()` and `equalsIgnoreCase()`), size (e.g., `length()`), and substring (e.g., `contains()`, `substring()`, and `charAt()`). However, it is difficult for Randoop to randomly find a desirable string to satisfy such constraints, since the input space of string type values is huge. Recent approaches on string generation with symbolic execution [7, 16, 22] can alleviate this issue.

The third main cause is *container object access*. It is not easy to create a certain size of a container with necessary elements. Similarly, the fourth main cause, *array comparisons*, includes accessing array elements and checking their size; randomly creating an array with desirable elements or size is difficult. Indeed, recent symbolic execution approaches such as Pex [33] can effectively handle method arguments as array elements in symbolic execution.

[2]This extended Randoop generally achieves higher code coverage than the original Randoop by generating array inputs and applying adaptive random selection of object instances and methods.

[3]`http://commons.apache.org/`

[4]`http://santuario.apache.org/`

[5]`http://martiansoftware.com/jsap`

[6]We put *access of a reference array type* into the "container object access" category.

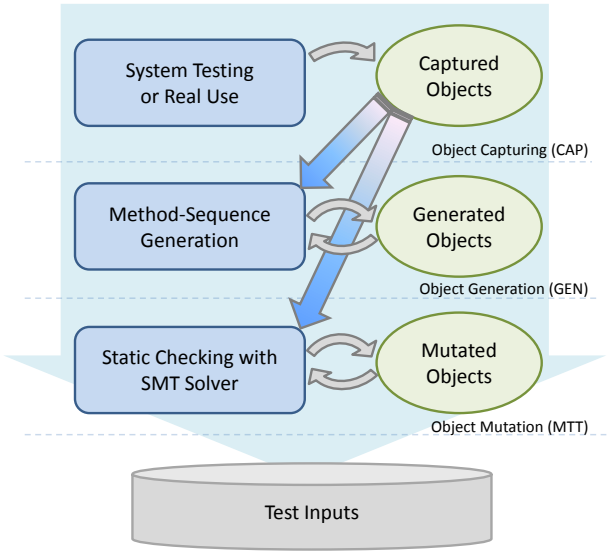

Figure 2: Overview of our OCAT approach

Exception branches are related to a `try..catch` statement, or a branch that checks exception types. These branches are relatively hard to cover automatically , since these branches have a particularity of exception handling code that handles run-time errors. To determine a desirable condition, a tool needs to know how exceptions are defined and where exceptions have been thrown (e.g., an exception propagation path).

The last two main causes are branches related to an environmental setting (e.g., environment variables and file-system structure) and non-deterministic execution (e.g., multi-threading and user interactions), which are difficult for automatic testing approaches to address. Indeed, the former cause can be alleviated through recent approaches on mock objects [25, 34] and the latter cause can be alleviated through recent approaches on concurrent testing approaches [27].

The top three main causes are all related to a lack of desirable object instances, and few existing approaches can effectively address these causes, especially when testing real-world code bases. Similarly, Thummalapenta et al. [32] identified that creating desirable object instances is a main challenge in other automated testing techniques, such as symbolic execution [23, 28].

The OCAT approach proposed in this paper tackles the challenge of generating desirable object instances in automated test generation.

3. OCAT APPROACH

In this section, we describe the OCAT approach in detail. Figure 2 shows the overview of OCAT's three phases: object capturing (CAP), object generation (GEN), and object mutation (MTT). In the CAP phase, OCAT conducts bytecode instrumentation of the program under monitoring and then executes the instrumented system instead of the original one with the given program executions. During these executions, the instrumented system captures encountered object instances and serializes them into a file. In the GEN phase, OCAT generates object instances by feeding the captured object instances into an existing automated test-generation tool that generates method sequences; these generated method sequences derive object instances from the captured object instances.

```
public Algorithm(Document doc,
   String algorithmURI) {

   //instrumented
   ObjCapture.capture(new Object[] {this,
      doc, algorithmURI});

   super(doc);

   this.setAlgorithmURI(algorithmURI);
}
```

Figure 3: Instrumented code - this code is instrumented from the code shown in Figure 1

In the MTT phase, to cover those branches not yet covered by the GEN phase, OCAT mutates captured object instances to cover not-yet-covered branches.

We next describe each OCAT phase in detail.

3.1 Object Capturing (CAP)

The key idea of OCAT is that capturing object instances from normal program executions can be used directly or indirectly for generating unit tests of the classes under test to achieve high structural coverage, such as branch coverage. Some example types of objects are (1) classes under test (e.g., receiver classes), (2) arguments of a method under test, and (3) objects needed to directly or indirectly construct the first two types of objects. To capture these types of objects, OCAT requires normal program executions such as those from system tests or user interactions with systems.

Our CAP phase consists of two parts: *instrumentation* and *object serialization*. In the first part, *instrumentation*, OCAT inserts object-capturing code at each method's entry point of the program under monitoring. The inserted code invokes a `capture` procedure shown in Algorithm 1 and passes arguments to OCAT's serialization module. Figure 3 shows an instrumented code example. The `capture` procedure captures object instances along with their types.

The next part is *object serialization*, described in Algorithm 1. First, we run the instrumented program to serialize object instances through normal program executions. If the instrumented `capture` procedure is called, OCAT receives a type and a concrete state of the object instance to be serialized. Then, OCAT keeps types and states of all serialized object instances to maintain an object repository without redundant object instances with the same state [39]. OCAT serializes object instances in memory first and serializes the object instances into files whenever the number of object instances reaches a preset limit or the program under monitoring finishes its execution.

One challenge here is there are many types of objects in program executions and the same type of objects can also have many instances. We observed that most of these object instances have isomorphic states, and these instances do not contribute to increasing code coverage. Therefore, it is desirable to capture only object instances with non-isomorphic states for each class type. To check state isomorphism of object instances, OCAT uses a concrete state representation. Xie et al. [39] defined a state representation of a program heap. We adopt a part of their definition to define a state of an object instance, being a subset of a program heap state.

Let P be the set consisting of all primitive values, including `null`. Let O be a set of objects whose fields form a set F.

Algorithm 1 Pseudo-code for object capturing

```
Map container keeps object states to avoid having redundant objects
Set container keeps object instances to be serialized
01 Map<type, state> mapStates
02 Set<object> setObjs
03
04 procedure capture(objs[])
05 begin
06  for each obj ∈ objs[]
07   //get a type and linearized state of the object instance obj
08   type ← getType(obj);
09   state ← representState(obj);
10
11   //check whether the obj has been serialized
12   if (!isSerialized(type, state)) then
13    mapStates.add(type, state);
14    setObjs.add(obj);
15   end
16  end
17
18  //check whether enough number of obj has been serialized
19  if (hasEnoughInstances(setObjs)) then
20   // serialize object instances and clear the setObjs set
21   serializeToFileandClearSet_Thread(setObjs);
22  end
23 end
```

DEFINITION 1. *A* state *is an edge-labeled graph* $\langle O, E\rangle$*, where*

$$E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}.$$

State isomorphism is defined as graph isomorphism based on node bijection [8].

DEFINITION 2. *Two states* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there is a bijection* $\rho : O_1 \rightarrow O_2$ *such that:*

$$\begin{aligned} E_2 \quad = \quad & \left\{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\right\} \cup \\ & \left\{\langle \rho(o), f, o'\rangle | \langle o, f, o'\rangle \in E_1, o' \in P\right\}. \end{aligned}$$

These preceding definitions of state representation and isomorphic states help prune redundant object instances: if two object instances have isomorphic states, then these two instances are considered as redundant. Other than using object-state isomorphism to detect redundant object instances, two alternative ways could be used. One way is to use an abstract state representation [37]. However, an abstract state representation is too coarse-grained, since it ignores the field values in an object instance and checks only the structural shape of the object instance. Another way is to use `equals()`; however, the outcomes of executing `equals()` depend on its implementation, and `equals()` can be implemented in various ways by different developers. For example, `equals()` may be implemented, based upon an object instance's reference value. Although reference values of two instances are different, these two instances could have the same states. In this case, using `equals()` identifies that these two instances have different states.

According to our empirical study in Section 4, using an abstract state representation incurs storage of few object instances and using `equals()` incurs storage of too many isomorphic object instances. Consequently, both aforementioned alternative ways are

```
FastTreeMap var0 = (FastTreeMap) Serializer.
      loadObject("capobj/FastTreeMap/hash_32");
String var1 = (String) Serializer.
      loadObject("capobj/String/hash_98");
BeanMap var2 = (BeanMap) Serializer.
      loadObject("capobj/BeanMap/hash_32");
Integer var3 = (Integer) Serializer.
      loadObject("capobj/Integer/hash_808");

var0.add(var3,var1);
var0.putAll((java.util.Map)var2);
```

Figure 4: A generated method sequence with captured object instances by FRT

inadequate to identify different object states for OCAT. Therefore, we use the concrete state representation.

3.2 Object Generation (GEN)

This section describes the GEN phase, which generates object instances by invoking method sequences with captured object instances.

After object instances are captured and serialized from a program under monitoring, we de-serialize and use them as test inputs. Particularly, we leverage a method-sequence generation technique by using the captured object instances in two ways. First, the captured object instances can be directly used. Second, the captured object instances contribute to the creation of other necessary object instances for testing. Let C be a set of captured object instances by OCAT. Consider two target methods m_i of class i and m_j of class j. Consider two sets of desirable object instances D_{m_i} and D_{m_j} that cover code in method m_i and m_j, respectively. Let R_{m_j} be a set of object instances returned by invoking m_j on D_{m_j}. If $D_{m_i} \subseteq C$, code in method m_i can be directly covered by using the captured object instances. If $D_{m_i} \not\subseteq C$, but $D_{m_i} \subseteq R_{m_j}$ and $D_{m_j} \subseteq C$, then code in m_i can be indirectly covered by feeding the object instances returned by invoking m_j on the captured object instances.

Any method-sequence generation technique can derive object instances from captured object instances. In this paper, we use a *feedback-directed random test generation* (FRT) technique [26] that randomly generates method sequences and verifies their validity by execution. FRT generates method sequences starting from a set of primitive-type declarations with predefined values. In our case, FRT starts with a set of method calls that de-serialize captured object instances and declarations of other primitive-type input values. The de-serialized captured object instances provide a good basis to lead the method-sequence generation technique to generate desirable object instances.

In particular, first, FRT selects one of the methods under test. To determine input arguments and a receiver for this selected method, FRT searches sequences from a sequence pool, which contains previously constructed sequences. If FRT finds sequences that construct the same type of objects as the type of one of the arguments and the receiver of the selected method, FRT merges these sequences, and appends the selected method to the end of the merged sequence to make a new sequence. In this way, FRT incrementally extends sequences. To verify the newly created sequence (the one just extended) and obtain feedback (return values/generated object instances), FRT executes the sequence. From this execution, FRT checks whether the sequence is illegal by observing whether exceptions are thrown. From the feedback, FRT performs state matching of object instances to discard sequences that create redundant object instances. By repeating this sequence-construction process (method selection, sequence selection, merging, extension, and execution), FRT incrementally generates new object instances.

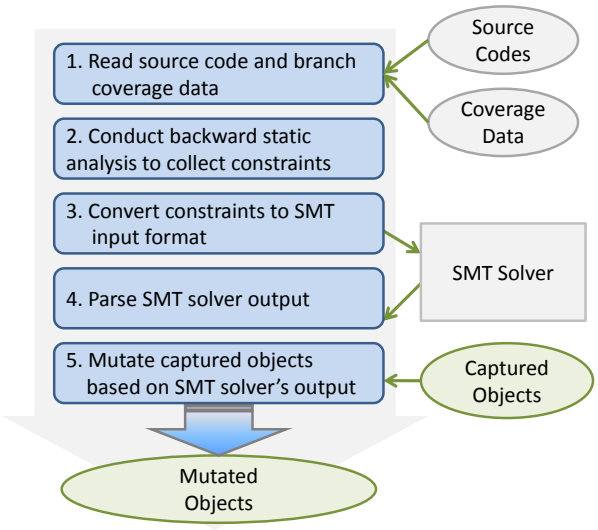

Figure 5: Overview of object mutation

Figure 4 presents an example of a generated sequence with object instances captured by FRT. A captured instance of `FastTreeMap` is modified by invoking `add()` and `putAll()` methods with other captured object instances. First, the example loads a `FastTreeMap` instance from captured object instances as a receiver `var0`. Then, `var1`, `var2`, and `var3` are loaded as input arguments for the last two subsequent method calls in Figure 4. In particular, the method call `var0.add()` adds the loaded instance to `FastTreeMap` and `var0.putAll()` adds all elements of a captured instance of `BeanMap`.

Using captured object instances as initial inputs reduces the huge search space of desirable object instances in the method-sequence generation process, since the captured object instances are likely close to desirable object instances. Therefore, captured object instances make the method-sequence generation approach effective to produce desirable object instances, and construct and execute method sequences with the captured object instances to achieve high code coverage. Our empirical results (Section 4) show that using captured object instances with FRT significantly increases the code coverage achieved by FRT alone.

3.3 Object Mutation (MTT)

Generating object instances by invoking method sequences with captured object instances may not cover all branches. The MTT phase statically analyzes conditions of not-covered branches after GEN phase. Then, the MTT phase generates method sequences for not-covered branches by purposely mutating captured object instances.

Figure 5 shows an overview of our MTT phase. (1) OCAT identifies not-yet-covered branches by analyzing source code and branch coverage information. (2) OCAT conducts static analysis to collect constraints starting from the not-yet-covered branches in a backward traversal manner of code analysis. (3) OCAT solves the collected constraints by using a Satisfiability Modulo Theories (SMT) solver [12]. (4) OCAT uses the solved solution from the SMT

```
public static final int HARD = 0;
protected int keyType;

...

public Object getKey() {
     return (parent.keyType > HARD)
           ? ((Reference) key).get()
           : key;
 }
```

Figure 6: Object Mutation Example

solver as a concrete input value of the method that has the target not-covered-branch. (5) If the solution is related to a member field of an object input, OCAT loads and mutates a captured object instance. OCAT uses the Java reflection API [31] to modify a value of the corresponding member field of a captured object, based on the solution of the SMT solver.

After mutating object instances, OCAT generates method sequences to test the target method with the mutated object inputs. To generate a method sequence, OCAT searches captured object instances for other necessary object instances for a receiver and arguments to invoke the method under test. If there is no proper object instance in the pool of captured object instances, OCAT directly creates the corresponding object instances by invoking one of the constructors of the class for the objects. If the constructor needs object instances as arguments again, OCAT recursively searches the pool or directly creates argument object instances within a preset depth of recursion.

After all, to verify whether a mutated object instance satisfies the condition of a target branch, OCAT concretely executes the generated sequence.

The following steps illustrate the example of mutating an `AbstractRefereceMap` object in Apache Commons Collection (see Figure 6):

1. The coverage report indicates that the true branch of "`parent.keyType > HARD`" has not been covered. OCAT parses the report and source code to determine the predicate of the not-covered branch.
2. OCAT checks variables in the predicate to determine whether they are input arguments including arguments, a receiver, or object fields of a receiver and arguments. Here, `HARD` is a constant field, whose value is 0 and `parent.keyType` is a protected member field of the receiver.
3. OCAT converts the predicate to the input format of Yices [12], a SMT solver:
   `(define var::int)(assert+ (> var 0))(check)`
4. Yices outputs `(= var 1)` from the input and OCAT parses the output.
5. OCAT randomly selects an instance of `AbstractReferenceMap` and modifies its `parent.keyType` to 1.

To compute input argument values from solved constraints, we do not consider predicates that involve local variables or method invocations in the target method body. However, applying sophisticated analysis and heuristics such as inter-procedural alias analysis [17] and directed call analysis [9] may help increase code coverage. Applying these techniques for OCAT remains as our future work.

When modifying member-field values of an object instance, OCAT does not change a private field value as a default setting since modifying a private field value might break class invariants and make

```
<org.apache.commons.collections.FastTreeMap
serialization="custom">
  <unserializable-parents/>
  <tree-map>
    <default/>
    <int>0</int>
    <string>first</string>
    <string>First Item</string>
    <string>second</string>
    <string>Second Item</string>
  </tree-map>
  <org.apache.commons.collections.FastTreeMap>
    <default>
      <fast>true</fast>
      <map>
        <no-comparator/>
        <entry>
          <string>first</string>
          <string>First Item</string>
        </entry>
        <entry>
          <string>second</string>
          <string>Second Item</string>
        </entry>
      </map>
    </default>
  </org.apache.commons.collections.FastTreeMap>
</org.apache.commons.collections.FastTreeMap>
```

Figure 7: An example of a serialized object instance

the object instance invalid. Indeed, our empirical results in Section 4 show that only a few cases that a variable in a target predicate is related to a private field. However, it is good to have an optional functionality to handle this kind of occasional situations. To avoid invalid object instances caused by modifying private field values, OCAT provides an option of allowing developers to provide a predicate method (also called `repOk()` [8]) that checks class invariants of a class. Programming best practices suggest that a programmer provides such a method when writing a class. After mutating an object instance, OCAT checks whether `repOk()` returns `true` to ensure state validity of the object instance [24]. For example, OCAT executes `repOk()` to check state validity and `getKey()` to verify it after Step 5. If it is still not a desirable receiver, throws an exception when executing `getKey()`, or when `repOk()` returns false, OCAT repeats Step 5 for up to a preset number of times. If the mutated object instance is valid, OCAT serializes the object instance as a new instance and uses it as a test input for the method under test. Otherwise, OCAT tries to mutate other object instances.

Our mutation technique is related to the technique of dynamic symbolic execution [23, 28]. Both use constraint solvers to change object instances and cover more branches using the changed object instances. However, as shown in Figure 1, without desirable object instances at the first phase, sometimes it is difficult to formulate all conditions and determine the constraints. There must exist unrevealed conditions, and thus a constraint solver becomes ineffective in such cases. For example, without a desirable `doc` object instance, a program execution throws an exception at Line 87 in Figure 1. Moreover, although conditions are perfectly formulated and constraints are well determined, it is difficult to construct a desirable object instance that satisfies these constraints. Captured desirable object instances help cover unrevealed conditions without perfect formulation of conditions. Our OCAT approach uses desirable object instances first by object capturing and object generation, and then applies object mutation.

Table 2: Open source project under test

Projects Name	Classes	Methods	KLOC
Apache Commons Collections 3.2 (ACC)	273	2522	63
Apache XML Security 1.0 (AXS)	179	1185	40
JSAP 2.1 (JSAP)	77	462	11

Table 3: Statistics of captured object instances: showing time to capture, number, serialized file size of captured object instances, and branch coverage of captured executions.

Projects	Time (sec)	# Objects	Size	Cov.
ACC	311	14999	7.6MB	52.7%
AXS	366	11390	410MB	36.0%
JSAP	1	292	15KB	32.9%

3.4 Implementation

We have implemented our approach to generate JUnit [6] tests using Randoop [26], Java reflection APIs [31], Java bytecode manipulation and analysis framework (ASM [10]), a test coverage measurement tool (Cobertura [3]), an object serializing and de-serializing library (XStream [38]), and an SMT Solver (Yices [12]). Next, we briefly describe the implementation of each phase in our approach.

CAP. Instrumentation techniques are widely used to capture object instances and infer their associated invariants [5, 14]. Similarly, OCAT uses an instrumentation framework called ASM [10] to insert object-capturing code. For storing object instances, OCAT uses the XStream framework [38], which serializes object instances into XML files. XStream can serialize object instances that do not implement the `java.io.Serializable` interface whereas the Java serialization technique cannot serialize objects that do not implement that interface. Figure 7 shows an example of serialized object instances in a form of XML.

GEN. The captured object instances are de-serialized for Randoop [26], a tool that implements FRT. The de-serialized object instances are used as seeds for Randoop to generate more object instances. To measure branch coverage of tests generated by Randoop with seeded captured object instances, we use Cobertura [3]. Cobertura instruments the target Java Bytecode and generates a coverage report after executing tests.

MTT. We apply a SMT solver called Yices [12] with a static analysis technique to mutate captured object instances. OCAT collects constraints of target branches and generates corresponding Yices inputs. Then, OCAT parses the output from Yices to obtain a concrete input value to mutate existing object instances. By using Java reflection APIs [31], OCAT mutates object instances and generates method sequences in a form of JUnit tests.

4. EMPIRICAL EVALUATION

We evaluate our approach by comparing code coverage of OCAT and a state-of-the-art tool, Randoop. We design our evaluation to address the following research questions:

Q1 How much can OCAT improve code coverage through captured object instances?

Q2 How much can mutated object instances further improve code coverage?

4.1 Evaluation Setup

We conduct our evaluation on a machine with Linux, Intel Xeon 3.00GHz, and 16GB memory. We adopt a widely used Java code coverage analysis tool, Cobertura [3], to measure branch coverage.

We use three open source projects, Apache Commons Collections (ACC), Apache XML Security (AXS), and JSAP as test subjects. Table 2 shows the information of the subjects. ACC [1] is an extended collection library that provides new utilities, such as buffer, queue, and map. AXS [2] is a library implementing the XML Digital Signature Specification and XML Encryption Specification. JSAP [4] is a command-line argument parser.

We apply OCAT to instrument each subject system to enable object capturing. Then, we execute system tests included in each subject system to capture object instances. Table 3 shows the statistics of the object capturing phase, including time spent in capturing object instances, the number of captured object instances, and the size of the serialized file. For ACC, OCAT captures and serializes 14,999 object instances in 311 seconds, and the serialized file is 7.6MB. For AXS, OCAT takes 366 seconds to capture 11,390 serialized object instances in 410MB. For JSAP, OCAT captures and serializes 292 object instances in 1 second. The serialized file is 15KB. The branch coverage of captured executions for each subject is 52.7% (ACC), 36.0% (AXS), and 32.9% (JSAP).

The captured object instances and the serialized file size depend on the subject system and the executed tests. For example, the AXS's serialized file is relatively big, since AXS loads and maintains XML file contents and the contents are stored in object instances. However, we find that the serialized files are manageable, even if we execute the entire suite of tests for a relatively large software system (i.e., a system that has over 1,000 methods) such as ACC and AXS. In addition, it is possible to limit the number of captured object instances and size of the serialized files.

4.2 Captured Objects

Q1 How much can OCAT improve code coverage through captured object instances?

In this section, we show coverage improvement by leveraging captured object instances through OCAT to assist Randoop (denoted as CAP + GEN). First, as described in Section 4.1, we capture object instances by executing tests provided with each subject system. The number of captured object instances is shown in Table 3. Then, we feed the captured object instances as test inputs for Randoop and measured the branch coverage. Similarly, we run Randoop alone on subject systems and measure the branch coverage. We run each tool until the tool's coverage is saturated, which means either one of the the tools' coverage levels off without much further increase, or one of tools runs out of memory and cannot continue to run. Since the number and quality of captured object instances depend upon the tests initially provided with the subject system, we also measure the coverage of such tests (denoted as a captured executions).

Figure 8 shows the branch coverage of three subjects with two approaches, Randoop and OCAT (CAP + GEN), based on the number of generated tests. As a baseline, the coverage of the captured executions is shown. The x-axis indicates the number of tests generated by each approach. As the number of tests grows, generally the branch coverage also increases.

As Figure 8 indicates, there is a substantial coverage difference between OCAT and Randoop. For ACC, after 46,000 tests are gen-

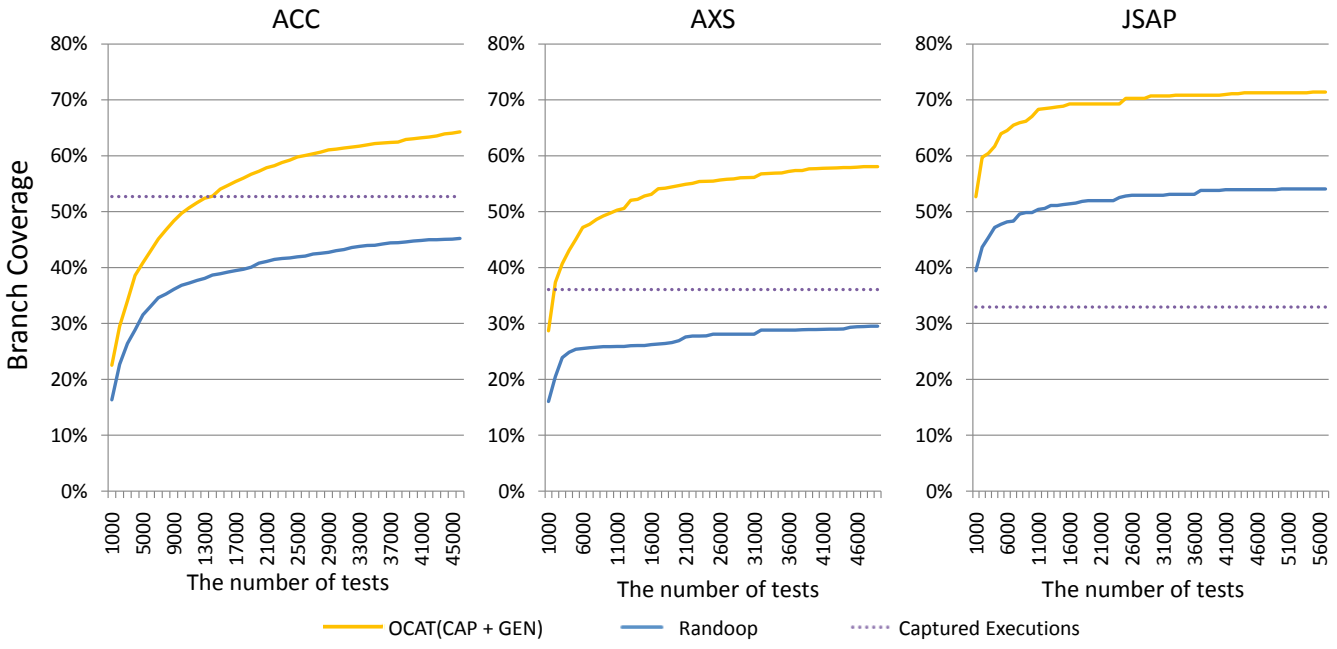

Figure 8: Branch coverage achieved by OCAT, Randoop, and Captured Executions

erated, OCAT achieves 64.2% branch coverage, 19.0% improvement from Randoop's achieved coverage, 45.2%. Similarly, for AXS, OCAT improves 28.5% branch coverage in comparison to Randoop. For JSAP, the coverage improvement is 17.3% over Randoop.

These encouraging results suggest that captured object instances plays an important role to improve code coverage by assisting an existing testing technique. We observe that the captured object instances are influenced by the original set of tests provided with the subject system. However, the OCAT's achieved coverage is much higher than the coverage achieve by simply executing these original tests shown in the last column of Table 3 and captured object instances can lead a random generation technique to cover more branches.

4.3 Mutated Objects

Q2 How much can mutated object instances further improve code coverage?

Next, we evaluate the MTT phase by using static analysis and a SMT solver. Although the CAP and GEN phases increase the code coverage substantially, still there are not-covered branches. For example, for JSAP, 28.6% of the branches are not covered after the CAP and GEN phases.

We further apply the MTT phase described in Section 3.3 and evaluate whether MTT improves the coverage further, although our current mechanism of mutating object instances can handle only limited set of situations. Note that we do not modify private fields of object instances in our evaluation.

The coverage distribution bars in Figure 9 present branch coverage improvement contributed by Randoop, OCAT (CAP + GEN), and OCAT (CAP + GEN + MTT). For ACC, 45.2% are covered by Randoop, and OCAT (CAP + GEN) further improves this coverage percentage by 19.0%. Finally, OCAT (CAP + GEN + MTT) increases an additional 3.9%. Similarly, after OCAT (CAP + GEN) improves the coverage by 28.5% for AXS and 17.3% for JSAP, OCAT (CAP + GEN + MTT) further improves the coverage by 4.2% more for AXS and 3.6% for JSAP.

Such an improvement may seem marginal. However, since code coverage becomes saturated after more than 46,000 tests, further improvement would be difficult to obtain in general. The MTT phase can still improve 3.9% of branch coverage on average, which is not trivial.

Overall, by combining random and systematic approaches, OCAT (CAP + GEN + MTT) improves branch coverage on average by 25.5% (with a maximum increase of 32.7% for AXS) in comparison to the coverage by Randoop alone.

Since we rely on a constraint solver to determine solutions for branch conditions, our object-mutation phase is limited by the ability to solve constraints. Currently, we can address only simple constraints such as Boolean, linear arithmetic, and integer expressions. By exploiting better constraint solvers, we may cover more branches. For example, we can solve string constraints using Hampi [22].

5. DISCUSSIONS

In this section, we discuss limitations of OCAT and threats to validity of our results.

5.1 Object-Capturing Process

OCAT's achieved coverage depends upon both quality and quantity of captured object instances. However, capturing object instances is a relatively easier process than writing unit test cases. For example, object instances can be captured from:

- executions of system tests
- typical system executions by individual developers or users

These executions can be completed independently and captured object instances are reused for test generation. Even captured ob-

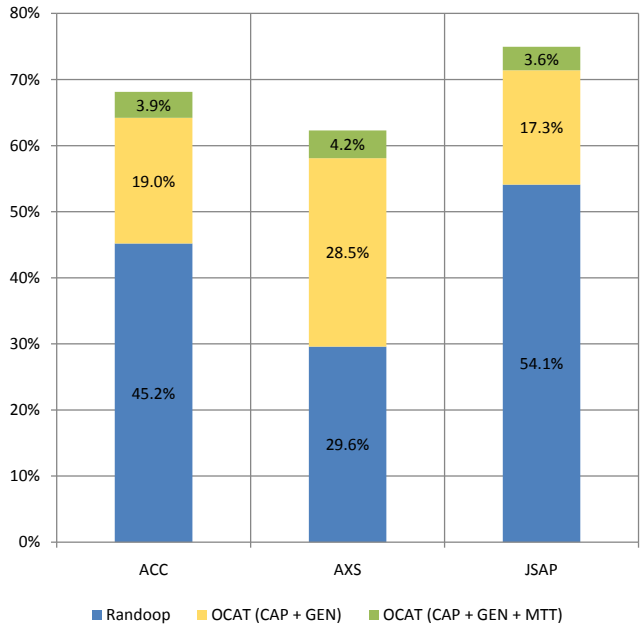

Figure 9: Branch coverage distributions of each approach with captured and mutated object instances

ject instances from a system can be reused for different system testing tasks as long as they share some object instances. For example, object instances in the JDK package (i.e., `java.util`) are commonly used for many Java applications. Note that a system under monitoring (a system with executions used by OCAT for capturing object instances) is not necessarily the system where the classes under test are integrated. A system under monitoring can be another system that uses (either consumes or produces) object instances falling into the same type of object instances shared between a system under test and a system under monitoring. That is to say, OCAT can be applied even before classes under test are integrated. Note that existing manually written unit tests or automatically generated unit tests for the classes under test can also be used to produce normal program executions. In this case, a system under monitoring is just the classes under test.

5.2 Software Evolution

Captured object instances may be obsolete and not be valid anymore during software evolution since an object instance from running a new system version may have different fields and methods from a same-type object from running a previous version. Nonetheless, if we can capture both the (dynamically collected) method sequences and the states of object instances of concern, it is possible to locally reproduce the object instances by executing the dynamically collected method sequences and update the obsolete object instances based on the captured state information. Such an extended capability for OCAT remains as our future work.

5.3 Branches Still Not Covered

Our evaluation result shows that OCAT increases 25.5% of branch coverage on average by using both generated and mutated object instances. However, there are still more than 20% of branches not covered. As shown in Table 1, some not-covered branches (due to string manipulation, container object access, and exception branches) are difficult to cover. We plan to further improve branch coverage via the following directions.

- Cross-system object capturing. We do not have to limit the sources for capturing object instances to be only subject systems. Suppose that we are testing ACC and it requires an object instance, `FOO`. It is possible to use other systems that use `FOO`, and capture the object instances from these systems. Capturing object instances from one system and use them for testing other systems may help further improve branch coverage.
- Static analysis. Currently we use a simple static analysis technique to mutate object instances. More complex analysis techniques such as inter-procedural alias analysis [17] can be applied to mutate object instances to cover more branches.
- Iterative generation and mutation phases. Two phases, object generation and object mutation, can be iteratively applied to generate a larger number of object instances over iteration. The quality of the generated object instances may decrease when the number of iterations increases. However, the iterations could likely improve branch coverage gradually over time since the two phases have complementary strengths to achieve structural coverage, and the new object instances generated from the object mutation phase could be exploited by the object generation phase.

5.4 Validity of Generated Object Instances

In the object generation phase, we assume that generated method sequences indirectly create valid object instances, if invocations of the method sequences do not throw exceptions. Indeed, the chance of creating invalid object instances may be low in this case. However, this assumption may not be true in general. In the object mutation phase, OCAT does not change private fields of object instances as a default option. Even if OCAT does not change private fields and change only public fields, it may still be able to create invalid object instances, which violate class invariants (either explicitly specified by developers or not specified at all). To avoid invalid object instances, developer could write the implementation of a special class-invariant-checking method, called `repOk()`, which checks the validity of mutated object instances. However, this method can be difficult or time-consuming to implement. To explore this issue, in future work, we plan to empirically investigate how high percentage of invalid object instances could be generated by our object generation and object mutation phases, respectively, when a `repOk()` method is not provided.

5.5 Threats to Validity

We identify the following threats to validity of our evaluation.

Our subject systems might not be representative. We use three open source projects, ACC, AXS, and JSAP, for our evaluation. It is possible that these three subject systems yield better or worse OCAT effectiveness than other systems. This threat could be reduced by more experiments on wider types of subjects in future work.

OCAT results rely on the captured object instances. The quality of our captured object relies on existing system tests or other program executions used for object capturing or existing system tests. Thus, OCAT's achieved coverage improvement depends on the quality and quantity of the system tests initially provided with the subject systems. In the future work, we plan to empirically investigate the impact of either quality or quantity of existing system tests or other program executions on the effectiveness of OCAT.

6. RELATED WORK

As discussed in Section 1, there are two main types of techniques for generating desirable object instances: direct object construction and method-sequence generation.

Two representative techniques for direct object construction are Korat [8] and TestEra [21]. Korat requires users to provide a `repOk()` predicate method, which checks the validity of a given object instance against the required class invariant for the class of the object. TestEra [21] requires users to provide class invariants specified in the Alloy specification language [19]. Both Korat and TestEra use class invariants to efficiently prune both invalid object instances (those violating class invariants) and redundant object instances (those with isomorphic states) when generating a bounded-exhaustive set of object instances (whose size[7] is within a relatively small bound). These techniques also require users to provide a finite domain of values for primitive-type fields in the generated object instances. The object-capturing phase of our OCAT approach can be seen as a type of direct object construction. However, OCAT constructs object instances from normal program executions (from system tests or real use) and these captured object instances are valid by construction, without requiring class invariants or a finite domain of values for primitive-type fields of objects.

Various techniques on method-sequence generation have been proposed for generating object instances used in test generation. Random-testing techniques (such as JCrasher [11] and Randoop [26]) generate random method sequences, sometimes with pruning based on feedback from previously generated sequences [26]. Evolutionary testing techniques (e.g., eToc [35] and Evacon [18]) use genetic algorithms to evolve initial method sequences to ones more likely to cover target branches. Our OCAT approach integrates Randoop, a random-testing technique, for evolving captured object instances, and in principle can integrate an evolutionary technique to evolve captured object instances. The object-capturing and object-mutation phases of OCAT provide added value and benefit beyond the integrated random testing technique in achieving branch coverage, as shown in our empirical evaluation in Section 4.

Bounded-exhaustive testing techniques (such as JPF [36], Rostra [39], and Symstra [40]) for method-sequence generation produce exhaustive method sequences up to a certain length, sometimes with pruning of equivalent concrete object states [36, 39] or subsumed symbolic object states [40]. However, the coverage of various branches requires long method sequences, whose lengths are beyond the small bound that can be handled by these techniques. In contrast, our OCAT approach is able to capture object instances produced with long method sequences (from real system executions) and further evolve these object instances with more method sequences or directly mutate these object instances.

Recent sequence-mining techniques, such as MSeqGen [32], statically collect method sequences (that can produce object instances of a specific) from various applications, and then apply dynamic symbolic execution [33] or random testing [26] on these collected sequences. MSeqGen shares the same spirit with OCAT in exploiting code elsewhere beyond the code implementation under test to improve automated test generation. Our OCAT approach has unique benefits over MSeqGen in the following two main aspects. First, OCAT dynamically captures real program execution environments such as user inputs, global states, and file I/O, whereas MSeqGen statically collects partial method sequences (from code bases), whose later execution often does not reproduce desirable program execution environments. Second, OCAT can capture object instances produced or affected by multi-threading, whereas MSeqGen does not support the collection of method sequences involving multi-threading. On the other hand, MSeqGen has its advantages, complementing OCAT in addressing the issue of generating desirable object instances. For example, since OCAT dynamically captures object instances from program executions, the ability to capture object instances relies on the quality of not only the programs under monitoring but also the system tests or real use that produces the program executions, whereas MSeqGen relies on the quality of only the programs under mining.

Capture-and-replay techniques have been primarily used for debugging, such as jRapture [30] and ReCrash [5] in reproducing a given program failure. These techniques have recently been used to generate unit tests for regression testing [13]. Based on the captured interactions, these techniques then generate unit tests for the class under test; in contrast to the system tests, these unit tests are less expensive to run when the class undergoes some changes. The execution of the generated unit tests would replay exactly the same unit behavior exercised in the capturing phase; the code coverage of the unit achieved in the capturing phase is the same as the code coverage of the unit achieved by the generated unit tests. In contrast, OCAT evolves and mutates the captured object instances, achieving higher branch coverage than the capturing phase. In addition, a captured object instance is used as inputs for many other methods (whose argument or receiver type is the same as the type of the captured object instance) well beyond the method whose input point of the object was originally captured.

7. CONCLUSIONS AND FUTURE WORK

In automated unit testing of object-oriented software, one important and yet challenging problem is to generate desirable object instances for receivers or arguments to achieve high code coverage (such as branch coverage) or find bugs. To address this significant problem, we proposed the OCAT approach, which captures object instances from program executions, generates more object instances using captured object instances and method sequences (exploiting a state-of-the-art random-testing tool called Randoop [26]), and mutates the object instances to cover those not-yet-covered branches. We evaluate OCAT on three open source projects, and our empirical results show that OCAT helps Randoop achieve high branch coverage: 68.5%, on average, improved from only 43.0% achieved by Randoop alone.

Besides the future work discussed in Section 5, we plan to explore other types of techniques in evolving captured object instances beyond random testing. Some other techniques under consideration include evolutionary-testing techniques [18, 35] and sequence-mining techniques [32]. Our current implementation for object mutation still has quite limited capabilities. We also plan to explore other types of techniques in mutating object instances. For example, we plan to apply dynamic symbolic execution [15, 29, 33] to derive constraints for mutating captured object instances to attempt to cover more not-covered branches.

8. ACKNOWLEDGMENTS

Hojun Jaygarl was a visiting student at the Hong Kong University of Science and Technology in the summer of 2009 when this work was partially carried out. Tao Xie's work is supported in part by NSF grants CCF-0725190, CCF-0845272, and CCF-0915400.

[7]The size of an object is the total number of object instances used to construct direct or indirect non-primitive fields of the object.

9. REFERENCES

[1] Apache commons. http://commons.apache.org.

[2] Apache XML security. http://santuario.apache.org/.

[3] Cobertura. http://cobertura.sourceforge.net/.

[4] JSAP. http://martiansoftware.com/jsap.

[5] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *Proc. ECOOP*, pages 542–565, 2008.

[6] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[7] N. Bjorner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proc. TACAS*, pages 307–321, 2009.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

[9] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proc. PLDI*, pages 363–374, 2009.

[10] O. Consortium. Asm. http://asm.objectweb.org.

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

[12] B. Dutertre and L. de Moura. System description: Yices 1.0. In *Proc. SMT-COMP*, 2006.

[13] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. Software Eng.*, 35(1):29–45, 2009.

[14] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, aug 2000.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[16] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proc. PLDI*, pages 188–198, 2009.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.

[18] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, September 2008.

[19] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.

[20] H. Jaygarl, S. Kim, and C. K. Chang. Practical extensions of a randomized testing tool. In *Proc. COMPSAC*, pages 148–153. IEEE Computer Society, 2009.

[21] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engg.*, 11(4):403–434, 2004.

[22] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Proc. ISSTA*, pages 105–116, 2009.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[24] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, specification, and object-oriented design*. Addison-Wesley, 2000.

[25] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *Extreme Programming Examined*. Addison-Wesley, 2001.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[27] K. Sen. Effective random testing of concurrent programs. In *Proc. ASE*, pages 323–332, 2007.

[28] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. CAV*, pages 419–423, 2006.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[30] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proc. ISSTA*, pages 158–167, 2000.

[31] Sun Microsystems. Java reflection API, 2001. Online manual.

[32] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. In *Proc. ESEC/FSE*.

[33] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[34] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Proc. ASE*, pages 365–368, 2006.

[35] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[36] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. ISSTA*, pages 97–107, 2004.

[37] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *Proc. ISSTA*, pages 37–48, 2006.

[38] E. Y. C. Wong, A. T. S. Chan, and H.-V. Leong. Efficient management of XML contents over wireless environment by Xstream. In *Proc. SAC*, pages 1122–1127, 2004.

[39] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.

[40] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.