Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution

Kobi Inkumsah
Department of Computer Science
North Carolina State University
kkinkums@ncsu.edu

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

***Abstract*—Achieving high structural coverage such as branch coverage in object-oriented programs is an important and yet challenging goal due to two main challenges. First, some branches involve complex program logics and generating tests to cover them requires deep knowledge of the program structure and semantics. Second, covering some branches requires special method sequences to lead the receiver object or non-primitive arguments to specific desirable states. Previous work has developed the symbolic execution technique and the evolutionary testing technique to address these two challenges, respectively. However, neither technique was designed to address both challenges at the same time. To address the respective weaknesses of these two previous techniques, we propose a novel framework called Evacon that integrates evolutionary testing (used to search for desirable method sequences) and symbolic execution (used to generate desirable method arguments). We have implemented our framework and applied it to test 13 classes previously used in evaluating white-box test generation tools. The experimental results show that the tests generated using our framework can achieve higher branch coverage than the ones generated by evolutionary testing, symbolic execution, or random testing within the same amount of time.**

I. Introduction

Software unit test coverage and adequacy measurements [1] provide a good basis for assessing software unit quality. In unit testing, achieving high structural coverage of the program unit under test such as a class helps increase confidence in the quality of the unit. Although various unit-test generation tools have been developed to help increase structural coverage such as branch coverage over manual testing, many branches in the program under test are difficult to cover due to two main challenges. First, some branches involve complex program logics and generating tests to cover them requires deep knowledge of the program structure and semantics. Second, in programs especially object-oriented programs, covering some branches requires special method sequences to lead the receiver object or non-primitive arguments to specific desirable states, and generating such method sequences is often challenging because of the huge search space of method sequences: we need not only the right method sequence skeleton[1] but also the right method arguments in the method sequence skeleton.

To address the first main challenge (especially to generate special primitive-type arguments to cover branches that are difficult to cover), recently symbolic execution tools such as JPF [2] and CUTE/jCUTE [3] explore paths in the program under test symbolically and collect symbolic constraints at all branching points of an explored path. The collected constraints are solved if feasible, and a solution is used to generate a test that forces the execution of the program under test along the path. This process is repeated until all feasible paths have been explored or the number of explored feasible paths has reached the user-specified bound. However, these symbolic execution tools do not provide effective support for generating method sequences that produce desirable receiver-object states or non-primitive-argument states.

To address the second main challenge, some bounded-exhaustive testing tools such as JPF [2], Rostra [4], and Symstra [5] generate exhaustive method sequences up to a small bound (with some pruning based on state equivalence [2], [4] or subsumption [2], [5]). However, sometimes covering some branches requires long method sequences whose length is beyond the low bound that can be handled by these tools. Some evolutionary testing tools such as eToc [6] represent initial randomly generated method sequences as a population of individuals and evolve this population by mutating its individuals until a desirable set of method sequences is found. However, because these evolutionary testing tools do not use program structure or semantic knowledge to directly guide test generation, they cannot provide effective support for generating desirable primitive method arguments even if the right method sequence skeleton is generated.

In this paper, we propose a novel framework called Evacon that integrates evolutionary testing [6] and symbolic execution [3] to address the respective weaknesses of these two techniques and to produce tests that achieve higher branch coverage than the tests generated by each technique alone. In particular, we establish a bridge from evolutionary testing to symbolic execution by generalizing concrete tests generated by evolutionary testing to symbolic tests as test drivers to symbolic execution. Therefore, symbolic execution can

[1] A method sequence skeleton is a method sequence whose methods' primitive arguments are unspecified.

help improve the method arguments in method sequences initially generated by evolutionary testing. We also establish a bridge from symbolic execution to evolutionary testing by encoding concrete tests generated by symbolic execution as chromosomes; these chromosomes are population individuals for evolutionary testing to evolve.

This paper[2] makes the following main contributions:

- a novel integration of two existing techniques to address a significant problem in structural testing of object-oriented programs;
- a comprehensive empirical comparison of our integration with state-of-the-art representative testing tools for various test-generation techniques including search-based test generation using genetic algorithms, symbolic execution, and random testing;
- a detailed comparison of the strengths and weaknesses of different testing tools in terms of achieving high structural coverage. In particular, we have introduced the branch ranking metric to help gain insights on which tools can be good at covering those difficult-to-cover branches and which tools can be used in combination to achieve better coverage.

We have implemented our proposed framework and applied it to test 13 classes previously used in evaluating white-box test generation tools. The empirical results show that our framework can achieve higher branch coverage than evolutionary testing, symbolic execution, or random testing.

The rest of the paper is organized as follows. Section II explains our framework through an illustrative example. Section III describes our framework. Section IV discusses evaluation results. Section V presents threats to validity. Section VI discusses issues of our framework and evaluation. Section VII presents related work. Finally, Section VIII concludes.

II. Example

We next illustrate how our framework is used in testing object-oriented programs through a `BankAccount` example as shown in Figure 1. This bank account example, which implements a bank account service, has been adapted for illustration purposes from the `BankAccount` class used in our evaluation. The `BankAccount` class declares several public methods. Among them, the `deposit` method allows money to be deposited in the account. The `withdraw` method allows money to be withdrawn from the account. The `withdraw` method begins by checking whether the withdrawal amount is more than the available balance. If so, an error message is printed and the method exits. The method also checks if the number of previous withdrawals (`numberOfWithdrawals`) is at least `10`. If so, another error message is printed and the method exits; otherwise, the withdrawal amount is dispensed, and both `balance` and `numberOfWithdrawals` are updated.

Figure 2 shows a sample test generated by an evolutionary testing tool for `BankAccount`. The test invokes the `withdraw` method with an argument value (`20.00`) that is greater than the argument value (`1.00`) of the earlier `deposit` method. This test cannot cover the false branch of the first conditional within the method body of `withdraw`. Note that additional method invocations of `withdraw` forming a longer method sequence cannot succeed in exercising the false branch of the first conditional unless a `withdraw` method argument is less than the argument value of the earlier `deposit` method. However, an evolutionary testing tool such as eToc [6] relies on random testing for generating primitive argument values for `withdraw`, and it is not effective in generating desirable argument values.

To address the weakness of evolutionary testing in generating desirable primitive argument values, we integrate evolutionary testing and symbolic execution. In particular, we

[2]An earlier version of this work is described in a short paper presented at ASE 2007 [7].

```
public class BankAccount {
     private double balance;
     private int numberOfWithdrawals;
     public void deposit(double amount) {
       if (amount > 0.00)
         balance = balance + amount;
     }
     public void withdraw(double amount) {
       if (amount > balance) {
         printError();
         return;
       }
       if (numberOfWithdrawals >= 10) {
         printError();
         return;
       }
       dispense(amount);
       balance = balance - amount;
       numberOfWithdrawals++;
     }
}
```

Fig. 1. A bank account example

```
public void testGenByEtoc() {
     BankAccount acc = new BankAccount();
     acc.deposit(1.00);
     acc.withdraw(20.00);
}
```

Fig. 2. A test generated by an evolutionary testing tool

```
public void testGenByEtocAugmentedByjCUTE() {
     BankAccount acc = new BankAccount();
     acc.deposit(10.00);
     acc.withdraw(1.00);
}
```

Fig. 3. A test generated by integrating evolutionary testing and symbolic execution

```
public void testGenByjCUTEAugmentedByeToc() {
     BankAccount acc = new BankAccount();
     acc.deposit(10.00);
     acc.withdraw(1.00);
     ...//repeated acc.withdraw(1.00) 9 times
     acc.withdraw(1.00);
}
```

Fig. 4. A test generated by integrating symbolic execution and evolutionary testing

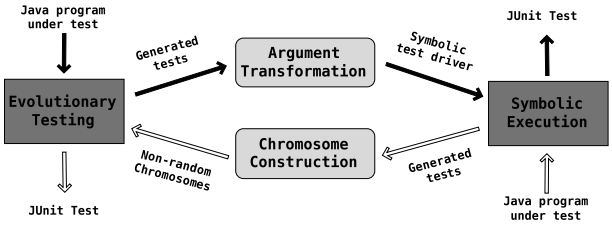

Fig. 5. Framework overview

generalize the concrete primitive values (`1.00` and `20.00` in Figure 2) in the sequence to be symbolic. Then given the sequence with symbolic values, a symbolic execution tool can generate concrete primitive values to cover feasible paths in the methods. One of the generated tests is shown in Figure 3. This test includes a desirable method argument value (`1.00`) of `withdraw` for covering its first conditional's false branch and the value is less than the argument value (`10.00`) of the earlier `deposit` method.

Given the sequence with symbolic values derived from Figure 2, a symbolic execution tool still can never generate method arguments to cover the true branch of the second `if` statement of `withdraw`; its coverage requires at least 11 successful withdrawals denoted by 11 invocations of `withdraw`. An existing symbolic execution tool such as jCUTE [3] does not provide mechanisms in searching for desirable method sequences. To address the weakness of symbolic execution in generating desirable method sequences, we integrate symbolic execution and evolutionary testing by evolving the method sequences generated by symbolic execution to a desirable one as shown in Figure 4 for covering the true branch of the second `if` statement of `withdraw`.

III. Framework

Our framework integrates evolutionary testing and symbolic execution to generate tests that can achieve high code coverage. Figure 5 shows the overview of our framework, including four components: evolutionary testing, symbolic execution, argument transformation (for bridging from evolutionary testing to symbolic execution), and chromosome construction (for bridging from symbolic execution to evolutionary testing).

A. Evolutionary Testing

Evolutionary testing techniques [6], [8]–[10] implement genetic algorithms mimicking natural evolution. In particular, Tonella [6] proposed an evolutionary testing technique to test object-oriented programs such as Java classes. According to the proposed evolutionary testing scheme, method sequences represent actions that can be encoded as chromosomes of individuals in a population. A population represents a potential solution to a testing goal, and this solution can be optimized through genetic re-combination and mutation. Furthermore, optimizing potential solutions requires the use of a formula of *fitness* to filter out *less suitable* individuals with regards to the testing goal while preserving *more suitable* ones. Re-combining and mutating the *more suitable* individuals then become the basis for generating a new population, which is hoped to be at least as *fit* as the predecessors [9].

For the program under test, each chromosome encodes object creation, a sequence of method calls to prepare the receiver object, and finally a call to the method under test. Below is an example of a method sequence and the resulting chromosome, which encodes the method sequence.

```
BankAccount acc = new BankAccount()
acc.deposit(1.00)
acc.withdraw(20.00)

$b0=BankAccount():$b0.deposit(double):
$b0.withdraw(double)@1.00,20.00
```

The receiver object of the sequence is identified using a unique alphanumeric string prefixed by the `$` symbol such as `$b0`. A chromosome has two parts separated by the `@` symbol such as `$b0.withdraw@40.00`. The first part encodes the actions of the method sequence using method argument types rather than the method arguments themselves, while the second part supplies the actual method arguments.

We have developed an algorithm shown in Figure 6 for test generation based on branch coverage. Our algorithm is inspired by and adapted from the approach of Tonella [6]. The algorithm performs four main steps. The first step is Population Initialization (Lines 2 to 5), followed by Fitness Calculation and Chromosome Selection (Line 14), and finally Re-combination and Mutation (Line 15). Evolution begins with instrumenting the program under test (Line 1) to determine the branch points within the program under test. The branch points are initialized as targets to be covered.

Population Initialization. To integrate evolutionary testing into our framework, we have added Lines 2 through 5 to Tonella's original algorithm [6]. This modification allows a non-random population of chromosomes (method sequences) obtained through chromosome construction (discussed in Section III-D) to be supplied for evolutionary testing. In particular, if the `useRandom` variable is initialized to `False`, a non-random population of chromosomes (Line 5) is used as a starting point for evolutionary testing instead of a random population. A target is selected (Line 8) and the genetic algorithm searches the population of method sequences for a method sequence that covers the selected target. Specifically, each method sequence in the population is executed (Line 10) to see if it covers the target. If a test is found to cover the target, it is saved, the algorithm exits the inner loop (Line 12), a new target is selected, and the remaining method sequences are executed on the new target. This process continues until all targets are covered or the evolution of tests is terminated because the maximum time for testing is reached.

Fitness Calculation. When a selected target cannot be covered by existing method sequences, the *fitness* of each method sequence is calculated (Line 14). The *fitness* of a method sequence measures the ratio of the control and call dependence edges traversed during the execution of the method sequence over the control and call dependence edges leading to the target. Therefore, method sequences with high ratios closer to one come closer to covering the target while method sequences

```
generateTests(programUnderTest:Class, useRandom:False)
1.  branchTargets <- instrumentor(programUnderTest)
2.  if useRandom then
3.    population <- generateRandomPopulation(size)
4.  otherwise
5.    population <- getCustomPopulation(size)
6.  while hasMoreToBeCovered(branchTargets)
7.     and time() < maxTime
8.    t <- getATarget(branchTargets)
9.    while t is not covered
10.     executeTestCases(population)
11.     update(branchTargets)
12.     if t is covered then break
13.     otherwise
14.       subPopulation <- getFitnessOf(t, population)
15.       population <- recombineAndMutate(subPopulation)
16.   end while
17. end while
```

Fig. 6. Genetic algorithm for test generation using branch coverage as the testing goal (adapted from Tonella [6])

with low ratios closer to zero move farther away from covering the target.

Re-combination and Mutation. Method sequences with high *fitness* values are selected for re-combination and mutation (Line 15) to produce offspring, which is the new population of method sequences. We have adapted Tonella's approach for carrying out re-combination and mutation of chromosomes. Re-combination is done on pairs of chromosomes. To re-combine a pair of chromosomes, we divide each chromosome into two parts using a randomly selected midpoint. Parts of the first and second chromosomes are swapped and re-combined to form new chromosomes. The example below shows how two chromosomes are re-combined (the source code representation of each chromosome is shown below the chromosome).

```
Parent chromosomes
 1. $b0=BankAccount():$b0.deposit(double):|
    $b0.deposit(double)@50.00,25.12

 Test code:
    b0=BankAccount();
    b0.deposit(50.00);
    b0.deposit(25.12);

 2. $b0=BankAccount():$b0.withdraw(double):|
    $b0.deposit(double)@3.50,100.00

 Test code:
    b0=BankAccount();
    b0.withdraw(3.50);
    b0.deposit(100.00);

Offspring chromosomes
 1. $b0=BankAccount():$b0.deposit(double):
    $b0.deposit(double)@50.00,100.00

 Test code:
    b0=BankAccount();
    b0.deposit(50.00)
    b0.deposit(100.00)

 2. $b0=BankAccount():$b0.withdraw(double):
    $b0.deposit(double)@3.50,25.12

 Test code:
    b0=BankAccount();
    b0.withdraw(3.50);
    b0.deposit(25.12);
```

First, the parent chromosomes are divided using the midpoint denoted by the | symbol. A new offspring chromosome is formed by combining one part of the first parent chromosome (the part to the left of the midpoint) with one part of the second parent chromosome (the part to the right of the midpoint). Similarly, another offspring chromosome is derived by combining the the left part of the second parent and the right part of the first parent. After re-combination, mutation operators insert or delete methods or method arguments within the new chromosomes. The mutation operators also insert missing constructors for new objects added through re-combination. After evolution, a set of method sequences are selected. Within our framework, evolutionary testing serves to construct suitable method sequences whose method arguments are to be improved through symbolic execution which we describe next.

B. Symbolic Execution

For Java programs, Sen and Agha [3] developed jCUTE, a symbolic execution tool for combining concrete and symbolic execution. We have adapted into our framework symbolic execution implemented by jCUTE. The symbolic execution technique implemented by jCUTE carries out two steps executed inside a loop. The steps involve concrete execution and constraint collection, followed by constraint solving and new input generation.

Concrete Execution and Constraint Collection. For the `withdraw` method in Figure 1, jCUTE randomly generates a concrete input for the variable `amount` (e.g., `3.00`), while assigning to `amount` a symbolic variable (e.g., a_0). If we assume that `balance` is less than `amount` (e.g., `balance` is `2.00`), when `withdraw` is invoked using amount of `3.00`, the execution takes the `true` branch of the first `if` statement. During this execution, jCUTE collects the path constraint ($a_0 > 2.00$) from the predicate of the first branch of `withdraw`. Furthermore, because `amount` is greater than `balance`, an error is printed and the method exits. However, since not all feasible paths have been explored, jCUTE does constraint solving and new input generation as described next.

Constraint Solving and New Input Generation. jCUTE proceeds with symbolic execution on the `withdraw` method by negating the last constraint collected to obtain a new constraint ($a_0 \leq 2.00$). The new constraint is then solved to obtain a concrete input for a_0 such that $a_0 \leq 2.00$. The `withdraw` method is invoked again with an argument value (e.g., 1.00) and the `false` branch of the first `if` statement is taken. jCUTE collects another constraint from the predicate of the second `if` statement of `withdraw`, which, when conjuncted with the previous constraint, yields ($a_0 \leq 2.00$ && `numberOfWithdrawals` ≥ 10). Since the second invocation of the `withdraw` method by jCUTE is the only successful one so far, `numberOfWithdrawals = 1` and the `false` branch of the second `if` statement is taken. The method exits after invoking the `dispense` method, updating the `balance`, and increasing the private field `numberOfWithdrawals` by one. To force the next execution path along the `true` branch of

the second `if` statement, jCUTE attempts to solve the last constraint that it collected (`numberOfWithdrawals` $\geq$ `10`). However, jCUTE cannot cover the `true` branch of the second `if` statement of the `withdraw` method due to two factors.

First, a longer sequence of method invocations involving the `withdraw` method (10 more successful invocations) is required, causing the private field `numberOfWithdrawals` to reach `10`. Although jCUTE can be configured to generate longer method sequences, its symbolic execution technique can explore method sequences up to only a small bound.

Second, jCUTE treats a non-primitive symbolic input as a memory graph and collects constraints on the memory graph during symbolic execution. After collecting path conditions including the constraints on the memory graph, jCUTE invokes a default constructor for the non-primitive argument and then directly assigns values to appropriate public fields of the argument in order to satisfy the constraints. If a field involved in the constraints is not public and its value is not the default value assigned by the non-primitive argument's default constructor, the constraints cannot be satisfied and jCUTE cannot generate a test to follow the corresponding path.

Symbolic execution using jCUTE requires a symbolic test driver, which is generated within our framework for the program under test through the use of argument transformation described in the next section. Within our framework, symbolic execution serves to construct suitable method argument values whose method sequences are to be improved through evolutionary testing. Consequently, the final produced tests characterize method arguments obtained through symbolic execution and method sequences obtained through evolutionary testing.

C. Argument Transformation

The argument-transformation component transforms primitive method arguments of method sequences (produced by evolutionary testing) into symbolic arguments [11]. This transformation allows jCUTE's symbolic execution technique to do concrete and symbolic execution on the primitive arguments. Our argument transformation process involves parsing method sequences generated by evolutionary testing to identify method invocations. For each method invocation requiring a method argument, we replace instances of concrete method arguments with equivalent symbolic arguments used to drive symbolic execution. In general, argument transformation can be used to transform any JUnit [12] method sequence into a symbolic test driver compatible with jCUTE's symbolic execution technique.

After symbolic execution, we derive the final test suite by aggregating the tests generated by symbolic execution and method sequences generated by evolutionary testing. In doing so, we preserve the level of coverage achieved by the method sequences obtained from evolutionary testing while augmenting this coverage by generating additional argument values that can achieve new coverage. Below is the resulting symbolic test after argument transformation is applied on the test in Figure 2. A double value is transformed to a symbolic double input represented as `cute.Cute.input.Double()`, an API method provided by jCUTE.

```
public void testGenByEvTest() {
   BankAccount acc = new BankAccount();
   acc.deposit(cute.Cute.input.Double());
   acc.withdraw(cute.Cute.input.Double());
}
```

The argument transformation component is used when test generation starts with evolutionary testing followed by symbolic execution. Given the preceding symbolic test, symbolic execution can help generate desirable method arguments for achieving new branch coverage; one test generated with symbolic execution is shown in Figure 3.

D. Chromosome Construction

The chromosome-construction component constructs chromosomes out of method sequences generated using symbolic execution. By using chromosome construction, method sequences from symbolic execution are made available to evolutionary testing through chromosome encoding. Chromosome construction involves two steps.

First, we extract the method sequences from symbolic tests with a dynamic analysis mechanism. Our mechanism involves compiling the symbolic test using ajc [13], an AspectJ [14] compiler. We then execute the ajc-compiled test using JUnit to dynamically collect exercised method sequences. The ajc compiler is capable of weaving AspectJ pointcuts into Java bytecode; an AspectJ pointcut specifies conditions that should be satisfied during program execution and corresponding actions to be performed when the conditions are met. In particular, our AspectJ pointcuts instruct to output all method sequences invoked by the tests.

In the second step, the entire method sequence is transformed to a chromosome. Each method call is encoded and all the encoded method calls are joined together. Below is the encoding for the test in Figure 3.

```
$b0,BankAccount,[]:
$b0,BankAccount,deposit,[double]:10.00
$b0,BankAccount,withdraw,[double]:1.00
```

Each encoding has four parts except for constructor invocations, which have three parts. The first part, which serves as a variable identifier for the receiver object, is a unique alphanumeric value prefixed by the `$` symbol. The identifier is assigned by the chromosome constructor. The second part is the name of the class to which the method being invoked belongs (this part is omitted for constructor calls). The third part is the name of the method being invoked. Finally, the fourth part lists the method arguments' data types and corresponding values. Below is the chromosome produced for the test in Figure 3 derived after encoding method calls and joining them together.

```
$b0=BankAccount():$b0.deposit(double):
$b0.withdraw(double)@10.00,1.00
```

To produce the above chromosome, the chromosome constructor maintains the association between the chromosome identifier and its associated method calls, as well as the associated method argument types and method argument values in their correct order. The final outcome of chromosome

TABLE I
EXPERIMENTAL SUBJECTS

Class	#public methods	#branches	LOC
BankAccount	6	6	60
BinarySearchTree	16	67	260
BinomialHeap	10	94	215
BitSet	25	130	638
DisjSet	6	44	140
FibonacciHeap	9	92	207
HashMap	10	89	374
LinkedList	29	105	738
ShoppingCart	6	13	117
Stack	5	16	160
StringTokenizer	5	47	222
TreeMap	47	252	1626
TreeSet	13	20	301

construction is a list of non-random chromosomes to be used in evolutionary testing. The chromosome-construction component is used when test generation starts with symbolic execution followed by evolutionary testing.

Evolutionary testing tries to find suitable combinations of method sequences, starting from the method sequences and method arguments generated by symbolic execution. For example, given the preceding chromosome, evolutionary testing can help generate desirable method sequences for achieving new branch coverage, such as the test in Figure 4.

IV. EVALUATION

We have implemented the Evacon framework in a tool for testing Java programs. In our evaluation, we investigate the following research questions:

- Is our proposed framework effective in generating tests that achieve higher branch coverage than existing representative test generation tools? This research question helps to demonstrate the utility of our proposed framework.
- What is the length of method sequences that achieve new branch coverage? This research question helps demonstrate that for certain types of branches, longer method sequences are indeed required for covering them.
- Are there test generation tools that provide unique coverage of some branches that cannot be covered by other tools? This research question helps to show which tools are required for achieving optimal branch coverage.

We compared Evacon's test effectiveness (in terms of branch coverage) with four publicly available test generation tools, representative of existing major test generation techniques. We selected eToc [6], which is an evolutionary testing tool for object-oriented programs as a representative of search-based test generation techniques using genetic algorithms. We selected jCUTE [3], which tests Java classes using the dynamic symbolic execution technique [15] as a representative of test generation tools that use symbolic execution. We also selected AgitarLabs's JUnit Factory [16] as a representative of industrial test generation tools. The JUnit Factory tool is an experimental test-generation service provided by AgitarLabs online at the JUnit Factory website [16]. Finally, we selected Randoop [17] as a representative of test generation tools in random testing. Randoop randomly generates tests for Java classes using execution feedback. Another well-known tool in symbolic execution is JPF [2]. We do not use JPF in our evaluation because it currently lacks support for test code generation. The tools used in our evaluation generate test code in the JUnit [12] format, which makes it possible to use a third-party code coverage tool to measure branch coverage in our tool comparison.

We conducted the experiments on a Pentium PC with a 1.86GHz processor and 1Gb memory. We have adapted Hansel [18] to record branch coverage for generated tests. Table I shows the 13 classes used in the experiments. The 13 classes are experimental subjects that have been previously used in evaluating white-box test generation tools [2], [4]–[6]. The `BankAccount` program, which implements a bank account service, is similar to our running example. The `ShoppingCart` program is an implementation of an online shopping cart service. The classes `BitSet`, `HashMap`, `LinkedList`, `Stack`, `StringTokenizer`, `TreeMap`, and `TreeSet` have been taken from the Java Standard library. The remaining classes are popular data structures. The classes range in size between 60 lines of code (LOC) and 1626 LOC. The number of public methods vary between 6 and 47. The number of branches within the classes vary between 6 and 252.

On the experimental subjects, we applied two types of Evacon integrations: bridging eToc to jCUTE with the argument transformation component (denoted as *Evacon-A*) and bridging jCUTE to eToc with the chromosome construction component (denoted as *Evacon-B*).

To provide a fair comparison across the six tools, we measure the branch coverage achieved by the tests generated by each of these six tools within the same period of running time, denoted as common runtime, except for JUnit Factory. It was not possible to impose a time limit on testing done by JUnit Factory. To use the online service requires uploading the program under test to a test server, which tests the program and makes the results available for download. The test server, which carries out testing of the program under test, cannot be stopped when the time limit for testing is reached. For the remaining tools, we use Evacon-A's runtime as the common runtime, being eToc's default runtime (60 seconds) plus jCUTE's runtime during Evacon-A's integration process.

We configure the other five tools (except for JUnit Factory) with the same runtime as below. For Evacon-B, we first construct a symbolic test driver for jCUTE to try bounded exhaustive method sequences up to the length of half the number of public methods. We then run eToc up to the common runtime. For eToc alone, we run it up to the common runtime. For jCUTE alone, we incrementally increase the bound of the bounded exhaustive method sequences till a bound that can cause the runtime to exceed the common runtime, and stop jCUTE when reaching the common runtime. For JUnit Factory, we upload each program under test to the JUnit Factory online test server and wait for the results to be

TABLE II
BRANCH COVERAGE ACHIEVED BY THE SIX TOOLS ON THE EXPERIMENTAL SUBJECTS

Class	Time (secs)	Evacon-A branch cov(%) (eToc⇒jCUTE)	Evacon-B branch cov(%) (jCUTE⇒eToc)	eToc branch cov(%)	jCUTE branch cov(%)	JUnit Fact branch cov(%)	Randoop branch cov(%)	All tools branch cov(%)
BankAccount	28	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	100.0
BinarySearchTree	238	94.0	95.5	92.5	88.6	88.1	**98.5**	**100.0**
BinomialHeap	228	94.7	92.6	90.4	89.9	88.3	**95.7**	**98.9**
BitSet	559	96.2	90.8	89.2	46.1	92.3	**96.9**	**100.0**
Disjset	346	**95.5**	93.8	90.9	55.8	86.4	90.9	**100.0**
FibonacciHeap	229	**97.8**	96.7	95.7	84.3	73.9	81.5	**100.0**
HashMap	374	**95.5**	91.0	80.9	58.8	80.9	60.7	**100.0**
LinkedList	687	**81.0**	77.1	79.0	64.0	67.6	1.0	**100.0**
ShoppingCart	145	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	100.0
Stack	28	**100.0**	93.8	93.8	95.8	93.8	**100.0**	100.0
StringTokenizer	291	**100.0**	97.9	89.4	80.4	**100.0**	91.5	100.0
TreeMap	276	**59.9**	57.9	48.8	53.4	23.4	15.5	**74.6**
TreeSet	135	**95.0**	85.0	65.0	86.1	25.0	80.0	95.0
Average	274	**93.05**	90.16	85.82	77.17	78.44	77.86	**97.58**

available for download. For Randoop, we run the command line utility in regression mode using the `timelimit` option, which we set to the common runtime.

A. Branch Coverage

Table II shows the first experimental results. Column 2 shows the common runtime measured in seconds. Columns 3-8 show the branch coverage achieved by the six tools: Evacon-A, Evacon-B, eToc, jCUTE, JUnit Factory, and Randoop, respectively. Column 9 shows the branch coverage achieved by aggregating all the tests generated using the six tools for each class.

We highlight in bold font and underline the table entries where the highest branch coverage is achieved among the six tools. We also highlight (in bold font) entries in the last column where the highest branch coverage for each program achieved by the six tools individually is less than the aggregated branch coverage by all tools. We observe that Evacon-A achieves the highest branch coverage in 10 of the 13 classes but for 3 of the 13 classes (`BinarySearchTree`, `BinomialHeap`, and `BitSet`), Randoop achieves the highest branch coverage. We suspect that this result is due to Randoop's ability to use execution feedback to avoid generating exception-throwing tests. We observed that while the other tools generated tests that caused the program under test to throw exceptions, preventing branch coverage of some key parts of the program under test, Randoop generated tests that tried to avoid these previously encountered exceptions.

In contrast, we also observed that for the `LinkedList` program, Randoop's branch coverage was the least among all the tools. In particular, we observed that although Randoop generated a large number of tests for `LinkedList` class, only a small fraction of the generated tests involved public methods of the `LinkedList` class. A large number of the tests involved setup and helper classes of public methods of `LinkedList`. Randoop also tied with Evacon-A in achieving full branch coverage for `Stack`. Both Evacon-A and JUnit Factory achieved full branch coverage for the `StringTokenizer` class. All tools achieved full branch coverage for the `ShoppingCart` and `BankAccount` classes. The tools eToc, jCUTE, and JUnit Factory did not perform better in terms of branch coverage compared with Evacon-A.

Furthermore, we observed that although Evacon-A and Evacon-B obtained the same level of coverage for the classes `BankAccount` and `ShoppingCart`, Evacon-A performed better than Evacon-B in terms of branch coverage for all other classes except for `BinarySearchTree`. We suspect that this result is because evolutionary testing in Evacon-B is not able to achieve new coverage using the method arguments obtained from symbolic execution even after evolving the method sequences. In contrast, in Evacon-A, a new set of method arguments are derived using symbolic execution. After aggregating the tests generated by all six tools, we observed that the branch coverage achieved by the six tools together surpassed the best branch coverage achieved by Randoop among all tools for three classes (`BinarySearchTree`, `BinomialHeap`, and `BitSet`) and the best branch coverage achieved by Evacon-A among all tools for five classes (`DisjSet`, `FibonacciHeap`, `HashMap`, `LinkedList`, and `TreeMap`). However, for five classes (`BankAccount`, `ShoppingCart`, `Stack`, `StringTokenizer`, and `TreeSet`), Evacon-A achieves the same level of branch coverage as all the tools combined.

The last row of Table II shows the average branch coverage achieved by each tool. To calculate the average branch coverage, we sum the branch coverage values for each column (from Columns 3 to 9) and divide each column by 13 (the number of programs under test). Evacon-A achieved the highest average branch coverage (93.05%) compared with all other tools followed by Evacon-B, which achieved (90.16%) average branch coverage, although the average branch coverage of either Evacon-A or Evacon-B is lower than the aggregated average branch coverage achieved by all the tools combined. This result suggests that for some classes, using a single tool may not be sufficient in achieving optimal coverage and using multiple tools in combination may be beneficial.

Table III shows the length of the longest method sequences that achieve new branch coverage for subjects where Evacon-A

TABLE III
THE LENGTH OF THE LONGEST METHOD SEQUENCE GENERATED BY EVACON-A OR RANDOOP THAT ACHIEVES NEW BRANCH COVERAGE

Class	Evacon-A	Randoop
BinarySearchTree	-	17
BinomialHeap	-	18
BitSet	-	23
DisjSet	7	-
FibonacciHeap	14	-
HashMap	13	-
LinkedList	16	-
Stack	13	-
StringTokenizer	9	-
TreeMap	23	-
TreeSet	13	-

or Randoop achieves higher branch coverage than the remaining tools. We have used the "-" symbol for the table entries under Evacon-A in which Randoop achieves higher branch coverage or table entries under Randoop in which Evacon-A achieves higher branch coverage. For all the classes under test except for `DisjSet` and `StringTokenizer`, method sequences of length at least 13 were required to achieve new branch coverage. In other words, existing tools [2]–[5] with bounded exhaustive method sequences may need to be able to handle a relatively large bound in order to achieve new branch coverage for many of the experimental subjects.

B. Branch Coverage Subsumption

We also observed from our experiments that the branch coverage achieved by some tools subsumed[3] the branch coverage achieved by other tools for some of the programs under test. Although the branch coverage of no tool subsumed the branch coverage of all other tools in all the programs under test, the branch coverage of Evacon-A subsumed the branch coverage of Evacon-B (in 12 of the 13 programs under test), eToc (in 7 of the 13 programs), jCUTE (in 3 of the 13 programs), JUnit Factory (in 1 of the 13 programs), and Randoop (in 4 of the 13 programs). The branch coverage of Randoop subsumed the branch coverage of Evacon-A (in 1 of the 13 programs). Randoop was the only tool whose branch coverage subsumed Evacon-A's branch coverage. We suspect that this result is due to the large number of tests (over 1800 tests on the average) generated by Randoop.

To identify the pairwise combination of the tools that achieve an optimal coverage of the programs under test, we investigated the pairwise combinations of the tools that led to the highest subsumption. We observed that Evacon-A and JUnit Factory achieved the highest pairwise subsumption. The branch coverage achieved by both tools together subsumed all the other tools in 5 of the 13 programs under test. This result suggests that for 8 of the 13 programs under test, more than a pairwise combination of the tools may be required to achieve an optimal coverage.

[3]Branch coverage A subsumes branch coverage B if all branches covered in B are also covered in A, but there exist branches covered in A not covered in B.

TABLE IV
BRANCH COVERAGE ACHIEVED BY DIFFERENT TOOLS USING BRANCH RANKING

Branch rank	Evacon-A	Evacon-B	eToc	jCUTE	JUnit Fact	Randoop
1	5/13	0/13	2/13	2/13	2/13	2/13
2	5/17	1/17	2/17	7/17	13/17	6/17
3	13/16	7/16	3/16	4/16	13/16	8/16
4	49/49	39/49	27/49	24/49	33/49	24/49
5	129/129	127/129	120/129	86/129	78/129	105/129

C. Branch Ranking

We have also conducted an initial study to gain insights on which tools can be good at covering difficult-to-cover branches and which tools can be used in combination to achieve better coverage. In our study, we rank all the branches within the 13 classes under test based on the number of tools that can cover them. Table IV shows the branch coverage achieved by the six tools using our branch ranking metric. Column 1 shows the different ranks. A rank-1 branch is covered by only one of the six tools while a rank-2 branch is covered by only two of the six tools. Similarly, rank-3, rank-4, and rank-5 branches are covered by only three, four, and five of the six tools, respectively. Covering a majority of the top-ranked (e.g., rank 1 and 2) branches demonstrates a tool's effectiveness in covering branches that cannot be covered by other tools. To be concise, we have omitted rank-0 branches (branches covered by none of the six tools) and rank-6 branches (branches covered by all six tools) from our comparisons in Table IV.

The entries in each column (from Columns 2 to 7) show the ratio of the number of branches that have been covered by each tool over the number of branches in a particular rank. Out of 975 branches taken from the 13 classes under test, 675 of the branches were covered by all six tools while 76 branches were not covered by any of the six tools. Among the remaining 224 branches, Evacon-A covered 5 of 13 rank-1 branches while each of the remaining tools covered 2 of 13 rank-1 branches except for Evacon-B, which did not cover any rank-1 branches. Evacon-A also covered 5 of 17 rank-2 branches second to JUnit Factory, which covered 13 of 17 rank-2 branches. Among the top-ranked (rank 1 and 2) branches, JUnit Factory covered 15 of 30 branches, five more than Evacon-A, which covered 10 of 30 top-ranked branches. In general, Evacon-A covered more branches than all the other tools except for rank-2 branches in which JUnit Factory covered more. This result suggests that both Evacon-A and JUnit Factory are effective in covering the branches that are not covered by the remaining tools. However, more experiments are needed to gain a better understanding of the benefits of our branch ranking metric.

Overall, the experimental results demonstrate the benefits of the Evacon integration techniques over eToc and jCUTE alone as well as JUnit Factory and Randoop.

V. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs are representative of true

practice. Our subjects are from various sources and they have non-trivial size for unit testing. Our experiment had integrated and compared with two third-party tools (eToc and jCUTE), both are representative test generation tools. Our comparison also includes JUnit Factory and Randoop, which are also representative test generation tools. These threats could be further reduced by experiments on more subjects and third-party tools. The main threats to internal validity include instrumentation effects that can bias our results. Faults in our tool implementation, eToc, or jCUTE might cause such effects. To reduce these threats, we have manually inspected the source code of the generated tests for several program subjects.

VI. Discussion

Our initial study with the branch ranking metric shows that comparing tools based on the details of their covered branches could offer new insight beyond comparing just the percentages of branch coverage, which is an existing common way in comparing the effectiveness of tools. Our branch ranking metric is especially beneficial when selecting multiple tools to use in combination among the tools under comparison. As is shown in our initial study, detailed comparison of the branches being covered by the tools under comparison would provide guidance in selecting and using multiple tools together to achieve optimal branch coverage.

The motivation of the branch ranking metric is also related to residual coverage [19], which focuses on the coverage of entities such as branches that have not been covered yet. We can basically first apply one test generation tool or tool combination to cover those relatively-easy-to-cover branches and then select the best tool or tool combination to achieve the residual branch coverage (covering the not-yet-covered, often difficult-to-cover branches).

Our branch ranking metric is designed to compare the *relative* strength of each tool in a set of tools in terms of achieving branch coverage. For example, consider that we have a tool T that is not satisfactory in achieving branch coverage. If we compare tool T with a set of poorer tools that can help cover even fewer branches, T would be measured to perform well in terms of the branch ranking metric since T may be able to cover many branches (even ones considered to be easy to cover generally by test generation tools) that cannot be covered by all the other poorer tools under comparison. From the relative point of view, indeed T is better than the other poorer tools under comparison but T is not necessarily good in terms of achieving branch coverage. In other words, selecting which tools to compare with has important implication. In our experiments, we compared Evacon with advanced or state-of-the-art test generation tools with respect to the branch ranking metric. In such a setting, the better branch ranking metric achieved by Evacon over these existing tools would strongly indicate the effectiveness of Evacon in achieving branch coverage by itself.

With the branch ranking metric, including a poor tool in tool comparison would not affect the comparison of tools with relatively better effectiveness. To show an extreme case, let us include an extremely poor tool T (covering no branch of the programs under test) to our tool comparison shown in Table IV; in the new comparison, we have seven tools. In the new comparison, the resulting new table would have an extra column for the new tool T and the "x/y" entries for this column would have 0 for x, indicating that T can cover none of the rank-1 to rank-5 branches (in fact, T can cover no branches). In addition, the new table would have an extra row on the table bottom for rank-6 branches, which are the branches that can be covered by six tools out of the seven tools. This row would list "675/675" for all the six original tools and "0/675" for tool T. Note that the original entries in Table IV remain the same, i.e., the comparison among the six original tools is not affected. In general, for a tool with relatively poor performance (not to the extreme with 0 branch coverage), adding it to the tool comparison would primarily incur relatively slight changes on lower portion of the existing table entries.

In general, our branch ranking metric can be extended to other types of structural coverage types beyond branch coverage or even general types of coverage criteria beyond structural coverage. In future work, we plan to compare the effectiveness of test generation tools in achieving other types of coverage criteria such as data flow coverage [20] and mutation testing [21].

The two types of integrations in Evacon can form a feedback loop between evolutionary testing and symbolic execution. The feedback loop can start from either evolutionary testing or symbolic execution. Then the iterations can continue until neither evolutionary testing nor symbolic execution can generate tests that achieve new branch coverage. In future work, we plan to empirically investigate the effectiveness of the feedback loop compared to the two existing integration types in Evacon.

VII. Related Work

There is a large body of work in the area of automatic test generation for programs but none of the existing techniques leverage evolutionary testing and symbolic execution to generate tests that include both suitable method arguments and suitable method sequences as we have proposed.

Among the existing test generation tools, DART [15], Randoop [17], and Jartege [22] use randomized processes to cheaply generate test data from a test domain. Random testing is not effective in generating tests to cover structural entities such as branches whose coverage requires special argument values or method sequences. To compensate for this limitation, DART uses dynamic analysis of the program behavior during random testing and generates new test data to direct program execution along alternative program paths. Randoop [17] uses user-specified properties in conjunction with execution feedback to produce both fault-revealing and regression tests. Jartege [22] uses specifications to reduce the number of irrelevant tests produced by its random process. By leveraging both symbolic execution and evolutionary testing, our technique both generates test data to exercise different

branches within the program under test and ensures optimal combination of method sequences.

Other techniques [3], [23], [24] aim at increasing structural coverage such as branch coverage of the program under test. Although our work is similar to these techniques in generating appropriate method arguments, our proposed technique produces appropriate method sequences, which are lacking in these structural coverage techniques. Search-based test generation [6], [10] uses genetic algorithms to find both method arguments and method sequences for the program under test. This testing process is usually referred to in the literature as evolutionary testing. Traditional evolutionary testing techniques suffer from the *path problem* (a phenomenon in which a search process is led away from its target) as described by McMinn et al. [8]. They propose factoring out paths to the search target and searching for test data for each individual path using dedicated genetic algorithms all operating in parallel. As a result, feasible paths contribute more toward the search landscape, promoting test data discovery. In contrast, our technique does not involve multiple genetic algorithms but still addresses the *path problem* by means of leveraging symbolic execution and constraint solving.

VIII. Conclusion

To achieve high structural coverage such as branch coverage of object-oriented programs, we have developed a novel unit-test generation framework called Evacon for integrating evolutionary testing and symbolic execution. The former searches for desirable method sequences with a genetic algorithm and the latter generates desirable method arguments by exploring alternate paths within the methods under test. In particular, our Evacon framework provides a bridge from evolutionary testing to symbolic execution by generalizing concrete tests (generated by evolutionary testing) to symbolic tests as test drivers to symbolic execution. Our Evacon framework also establishes a bridge from symbolic execution to evolutionary testing by encoding concrete tests generated by symbolic execution as chromosomes, inputs to evolutionary testing.

We have implemented our framework and applied it to test 13 classes previously used in evaluating white-box test generation tools. The experimental results show that the tests generated using our framework can achieve higher branch coverage than tests generated by evolutionary testing, symbolic execution, or random testing within the same amount time.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and a gift from Microsoft Research.

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[2] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for Java containers using state matching," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2006, pp. 37–48.

[3] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proc. International Conference on Computer Aided Verification*, 2006, pp. 419–423.

[4] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proc. IEEE International Conference on Automated Software Engineering*, 2004, pp. 196–205.

[5] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[6] P. Tonella, "Evolutionary testing of classes," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[7] K. Inkumsah and T. Xie, "Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs," in *Proc. IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 425–428.

[8] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search-based test data generation," in *Proc. International Symposium on Software Testing and Analysis*, 2006, pp. 13–24.

[9] R. P. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification & Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[10] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," in *Proc. Conference on Genetic and Evolutionary Computation*, 2005, pp. 1053–1060.

[11] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE Software*, vol. 23, no. 4, pp. 38–47, 2006.

[12] JUnit, http://www.junit.org/.

[13] AspectJ Compiler, http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html.

[14] AspectJ, http://www.eclipse.org/aspectj/.

[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[16] Agitar Labs, "JUnit Factory," http://www.junitfactory.com/.

[17] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion*, 2007, pp. 815–816.

[18] "Hansel 2.0," http://hansel.sourceforge.net/.

[19] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," in *Proc. International Conference on Software Engineering*, 1999, pp. 277–284.

[20] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 154–163.

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[22] C. Oriat, "Jartege: A tool for random generation of unit tests for java classes," http://arxiv.org/abs/cs/0412012.

[23] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in *Proc. IEEE International Conference on Automated Software Engineering*, 2000, pp. 219–228.

[24] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transaction Software Engineering Methodology*, vol. 5, no. 1, pp. 63–86, 1996.