# Evacon: A Framework for Integrating Evolutionary and Concolic Testing for Object-Oriented Programs

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 concolic testing technique (a combination of concrete and symbolic testing techniques) 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 concolic testing (used to generate desirable method arguments). We have implemented our framework and applied it on six classes taken from the Java standard library and basic data structures. The experimental results show that the tests generated using our framework can achieve higher branch coverage than evolutionary testing or concolic testing alone.

**Categories and Subject Descriptors:** D.2.5 [Testing and Debugging]: Testing tools

**General Terms:** Reliability.

**Keywords:** Test generation, Structural coverage.

## 1. INTRODUCTION

Software unit test coverage and adequacy measurements 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 concolic testing tools such as CUTE/jCUTE [1] execute the program under test using concrete inputs and collect symbolic constraints at all branching points during concrete execution. The collected constraints are solved if feasible, and the solutions are used to generate a new set of test inputs that force the next execution of the program under test along an alternate 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. During each execution, test inputs that lead to an unexplored path are saved and used to generate tests that achieve high code coverage such as branch coverage. However, these concolic testing 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 [4], Rostra [5], and Symstra [6] generate exhaustive method sequences up to a small bound (with some pruning based on state equivalence [4, 5] or subsumption [4, 6]). 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 [3] 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 [3] and concolic testing [1] 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 concolic testing by generalizing concrete tests generated by evolutionary testing to symbolic tests as

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

```
public class BankAccount {
      ...
     public void deposit(int amount){...}
     public void withdraw(int amount) {
         if (amount > balance) {
            printError();
            return;
         }
         dispense(amount);
         balance = balance - amount;
         points++;
         if (points == 10)
           alertCustomer();
     }
}
```

**Figure 1: A bank account example**

```
public void testGenByEtoc() {
     BankAccount acc = new BankAccount();
     acc.deposit(1);
     acc.withdraw(20);
}
```

**Figure 2: A test generated by an evolutionary testing tool**

```
public void testGenByEtocAugmentedByjCUTE() {
     BankAccount acc = new BankAccount();
     acc.deposit(10);
     acc.withdraw(1);
}
```

**Figure 3: A test generated by our integrated evolutionary-concolic testing tool**

```
public void testGenByjCUTEAugmentedByeToc() {
     BankAccount acc = new BankAccount();
     acc.deposit(10);
     acc.withdraw(1);
     ...//repeated acc.withdraw(1) 8 times
     acc.withdraw(1);
}
```

**Figure 4: A test generated by our integrated concolic-evolutionary testing tool**

test drivers to concolic testing. Therefore, concolic testing can help improve the method arguments in method sequences initially generated by evolutionary testing. We also establish a bridge from concolic testing to evolutionary testing by encoding concrete tests generated by concolic testing as chromosomes; these chromosomes are the generation of population individuals for evolutionary testing to evolve. We have implemented our proposed framework and applied it to test six classes taken from the Java standard library and basic data structures. The experimental results show that our framework can achieve higher branch coverage than evolutionary testing or concolic testing alone.

## 2. EXAMPLE

We next illustrate how our framework is used in testing object-oriented programs through a `BankAccount` example as shown in Figure 1. This `BankAccount` example is a Java class implementation of a bank account service, which declares several public methods. Among them, the `deposit` method allows money to be deposited in the account and 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; otherwise, the withdrawal amount is dispensed and the balance is updated. The end of the `withdraw` method body also implements a reward system in which each successful withdrawal earns a point and when the earned points reach `10`, an alert is issued to the customer.

Figure 2 shows a sample test generated by an evolutionary testing tool for `BankAccount`. The test invokes the `withdraw` method

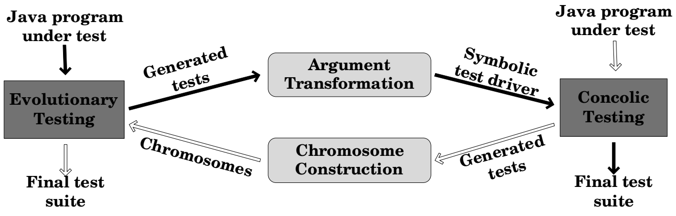

**Figure 5: Framework overview**

with an argument value (`20`) that is greater than the argument value (`1`) of the earlier `deposit` method. This test cannot cover the false branch of the first conditional within the method body. Note that additional invocations of the `withdraw` method and as such a longer method sequence cannot succeed in exercising the false branch unless a `withdraw` method argument is less than the argument value of the earlier `deposit` method. However, an evolutionary testing tool such as eToc [3] 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 and concolic testing. In particular, we generalize the concrete primitive values (`1` and `20` in Figure 2) in the sequence to be symbolic. Then given the sequence with symbolic values, the concolic testing 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`) of `withdraw` for covering its first conditional's false branch and the value is less than the argument value (`10`) of the earlier `deposit` method.

Given the sequence with symbolic values derived from Figure 2, a concolic testing tool still can never generate method arguments to cover the true branch of the second branch of `withdraw`; its coverage requires at least ten successful withdrawals denoted by ten invocations of `withdraw`. An existing concolic testing tool such as jCUTE [1] does not provide mechanisms in searching for desirable method sequences. To address the weakness of concolic testing in generating desirable method sequences, we integrate concolic and evolutionary testing by evolving the method sequences generated by concolic testing to a desirable one as shown in Figure 4 for covering the true branch of the second conditional of `withdraw`.

## 3. FRAMEWORK

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

### 3.1 Evolutionary Testing

Evolutionary testing techniques implement genetic algorithms mimicking natural evolution. We have integrated into our framework the technique for evolutionary testing proposed by Tonella [3]. The technique encodes a test (method sequence) as a chromosome of an individual of a population. 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. Evolution of tests begins with instrumenting the program under test to determine the branch points within the program under test. The branch points are initialized as the targets to be covered. A target is selected and a genetic algorithm searches a randomly generated population of tests for a test that covers the selected tar-

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

**Figure 6: The resulting symbolic test after argument transformation is applied on the test in Figure 2**

get. Specifically, each test is executed to see if it covers the target. If a test is found to cover the target, it is saved, a new target is selected, and the remaining tests are executed on the new target. This process continues until all targets are covered or the evolution of tests is terminated. When existing tests cannot cover a target, the fitness value of each test is calculated. The fitness value of a test is defined as a probability measure proportional to the ratio of the control and call dependence edges traversed during a test execution over the control and call dependence edges of the target. Subsequently, tests with the highest probabilities are selected for crossover and mutation to produce the next generation of tests. The crossover of chromosomes is to recombine those chromosomes based on some predefined guidelines. The mutation of chromosomes is to mutate methods and method arguments, or insert or delete methods within a chromosome. After evolution, a set of test cases are selected. Within our framework, evolutionary testing serves to construct suitable method sequences whose method arguments are to be improved through concolic testing.

## 3.2 Concolic Testing

The concolic testing technique [1] is a combination of concrete execution and symbolic execution. During concrete execution, all symbolic constraints along the path of execution are collected and conjuncted together. The constraints are solved, if feasible, to generate new test inputs that forces the next execution of the program under test along an alternate path. In particular, Sen and Agha [1] developed jCUTE for combining concrete and symbolic execution of Java programs. They use concrete input graphs for concrete inputs and symbolic states for symbolic variables, and execute the program under test simultaneously on both concrete and symbolic inputs with the symbolic execution guided by the concrete execution. During simultaneous execution on both input types, the technique collects constraints on the symbolic variables and solves them to produce inputs that force the next execution along a different path. The process is then repeated on the new set of inputs. Within our framework, concolic testing serves to construct suitable method argument values whose method sequences are to be improved through evolutionary testing.

## 3.3 Argument Transformation

The argument transformation component transforms the primitive method arguments of method sequences (produced by evolutionary testing) into symbolic arguments [2]. This transformation allows concolic testing to do concrete and symbolic execution on the primitive arguments. After concolic testing, we derive the final test suite by aggregating the test inputs generated by concolic testing and method sequences generated by evolutionary testing. Figure 6 shows the resulting symbolic test after argument transformation is applied on the test in Figure 2 (generated by evolutionary testing). An integer value is transformed to a symbolic integer input represented as `cute.Cute.input.Integer()`. The argument transformation component is used when test generation starts with evolutionary testing followed by concolic testing. Given the symbolic test in Figure 6, concolic testing can help generate desirable method arguments for achieving new branch coverage; one test generated with concolic testing is shown in Figure 3.

```
$x71,BankAccount,[]:
$x71,BankAccount,deposit,[int]:10
$x71,BankAccount,withdraw,[int]:1
```

**Figure 7: The resulting chromosome after chromosome construction is applied on the test in Figure 3**

| Class | #public methods | #branches | LOC |
|---|---|---|---|
| BinomialHeap | 10 | 63 | 215 |
| BitSet | 25 | 114 | 638 |
| FibonacciHeap | 9 | 73 | 207 |
| HashMap | 10 | 65 | 374 |
| LinkedList | 29 | 106 | 738 |
| TreeMap | 47 | 240 | 1626 |

**Table 1: Experimental subjects**

## 3.4 Chromosome Construction

The chromosome construction component constructs chromosomes out of method sequences collected from tests generated by concolic testing. Therefore, the tests generated by concolic testing are made available to evolutionary testing through the chromosome encoding. Figure 7 shows the resulting chromosome after chromosome construction is applied on the test in Figure 3 (generated by concolic testing). Each chromosome is a string encoding of the actions performed by a test. These actions are constructor invocations for object creation and one or more method invocations on the object. A strand in the chromosome has four parts except for constructor invocations, which have three parts. The first part identifies the chromosome using a unique alphanumeric value prefixed by the $ symbol. 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. Evolutionary testing tries to find suitable combinations of method sequences, starting from the method sequences generated by concolic testing. For example, given the chromosome in Figure 7, evolutionary testing can help generate desirable method sequences for achieving new branch coverage; one test generated with evolutionary testing is shown in Figure 4.

## 4. EVALUATION

We have implemented our Evacon framework by adapting eToc [3] and jCUTE [1]. We compared Evacon's test effectiveness (in terms of branch coverage) with eToc [3] or jCUTE [1] alone. We conducted the experiments on a Pentium PC with a 1.86GHz processor and 1Gb memory. Table 1 shows the six classes (taken from the Java standard library and basic data structures) used in the experiments; these classes were previously used in evaluating white-box test generation tools [3–6]. The classes range in size between 207 lines of code (LOC) and 1626 LOC. The number of public methods vary between 9 and 47. The number of branches within the classes vary between 63 and 240.

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*). As comparison bases, we also applied eToc and jCUTE, respectively, on the experimental subjects. To provide a fair comparison across the four tools, we measure the branch coverage achieved by the tests generated by these four tools within the same period of runtime, denoted as common runtime. 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

| Class | Time (mins) | Evacon-A branch cov (eToc⇒jCUTE) | Evacon-B branch cov (jCUTE⇒eToc) | eToc branch cov | jCUTE branch cov | New branch cov by Evacon over eToc/jCUTE |
|---|---|---|---|---|---|---|
| BinomialHeap | 5 | **100.0**% | **100.0**% | 96.0% | 92.0% | 4.0% |
| BitSet | 60 | **93.0**% | 91.0% | 88.0% | 65.0% | 5.0% |
| FibonacciHeap | 10 | 98.0% | **100.0**% | 98.0% | 66.0% | 2.0% |
| LinkedList | 23 | 68.0% | **69.0**% | 62.0% | 65.0% | 4.0% |
| HashMap | 9 | **67.0**% | 64.0% | 62.0% | **67.0**% | 0.0% |
| TreeMap | 56 | 67.0% | **75.0**% | 62.0% | 69.0% | 6.0% |

**Table 2: Branch coverage achieved by the four tools on the experimental subjects**

| Framework type | BinomialHeap | BitSet | FibonacciHeap | LinkedList | TreeMap |
|---|---|---|---|---|---|
| Evacon-A | 6 | 12 | - | 12 | - |
| Evacon-B | 8 | 13 | 10 | 14 | 20 |

**Table 3: The length of the longest method sequence generated by Evacon-A or Evacon-B that achieve new branch coverage**

configure the other three tools 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. Note that the branches not covered by jCUTE or eToc are usually difficult to cover and therefore covering even one of these difficult-to-cover residual branches is challenging enough. Evacon's ability of covering some of these residual branches would strongly reflect its effectiveness in test generation.

Table 2 shows the experimental results. Column 2 shows the common runtime. Columns 3-6 shows the branch coverage achieved by the four tools: Evacon-A, Evacon-B, eToc, and jCUTE, respectively. Column 7 shows the new branch coverage achieved by Evacon-A or Evacon-B over eToc or jCUTE. We highlight in bold font the table entries where the highest branch coverage is achieved. We observe that Evacon-A achieves highest branch coverage in three of the six classes (`BinomialHeap`, `BitSet`, and `HashMap`). For two of the three classes (`BinomialHeap` and `HashMap`), Evacon-A ties with Evacon-B and jCUTE, respectively, for achieving the highest branch coverage. We observe that Evacon-B achieves the highest branch coverage in four of the six classes (`BinomialHeap`, `FibonacciHeap`, `LinkedList`, and `TreeMap`). Evacon-B ties with Evacon-A for `BinomialHeap`. Both Evacon-A and Evacon-B together achieve higher branch coverage than eToc and jCUTE alone for all six classes except for `HashMap`, where Evacon-A ties with jCUTE. In summary, the experimental results demonstrate the benefits of the Evacon integration techniques over eToc and jCUTE alone and the two types of Evacon integrations are complementary, being effective in different subjects.

Table 3 shows the length of the longest method sequences that achieve new branch coverage for subjects where Evacon-A or Evacon-B achieves higher branch coverage than eToc and jCUTE alone. For these subjects except for `BinomialHeap`, method sequences of length at least ten were required to achieve new branch coverage. In other words, existing tools [1, 4–6] 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.

## 5. 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 concolic testing. The former searches for desirable method sequences with evolutionary algorithms 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 concolic testing by generalizing concrete tests (generated by evolutionary testing) to symbolic tests as test drivers to concolic testing. Our Evacon framework also establishes a bridge from concolic testing to evolutionary testing by encoding concrete tests generated by concolic testing as chromosomes, inputs to evolutionary testing.We have implemented our framework and applied it to six classes taken from the Java standard library and basic data structures. The experimental results show that the tests generated using our framework can achieve higher branch coverage than evolutionary testing or concolic testing alone.

The two types of integrations in Evacon can form a feedback loop between evolutionary testing and concolic testing. The feedback loop can start from either evolutionary testing or concolic testing. Then the iterations can continue until neither evolutionary testing nor concolic testing 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. We also plan to compare our Evacon tool with random testing tools [4] and systematic testing tools such as JPF [4], Rostra [5], and Symstra [6] in terms of their effectiveness of achieving structural coverage.

## Acknowledgments

This work is supported in part by a gift from Microsoft Research.

## 6. REFERENCES

[1] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pages 419–423, 2006.

[2] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.*, 23(4):38–47, 2006.

[3] P. Tonella. Evolutionary testing of classes. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[4] 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*, pages 37–48, 2006.

[5] 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*, pages 196–205, 2004.

[6] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.