# An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing

Marcelo d'Amorim[1], Carlos Pacheco[2], Tao Xie[3], Darko Marinov[1], Michael D. Ernst[2]

[1] Department of Computer Science, University of Illinois, Urbana-Champaign, IL, USA

[2] Computer Science and Artificial Intelligence Lab, MIT, Cambridge, MA, USA

[3] Computer Science Department, North Carolina State University, Raleigh, NC, USA

{damorim, marinov}@cs.uiuc.edu, {cpacheco, mernst}@csail.mit.edu, xie@csc.ncsu.edu

## Abstract

*Testing involves two major activities: generating test inputs and determining whether they reveal faults. Automated test generation techniques include random generation and symbolic execution. Automated test classification techniques include ones based on uncaught exceptions and violations of operational models inferred from manually provided tests. Previous research on unit testing for object-oriented programs developed three pairs of these techniques: model-based random testing, exception-based random testing, and exception-based symbolic testing. We develop a novel pair, model-based symbolic testing. We also empirically compare all four pairs of these generation and classification techniques. The results show that the pairs are complementary (i.e., reveal faults differently), with their respective strengths and weaknesses.*

## 1. Introduction

Unit testing checks the correctness of program units (components) in isolation. It is an important part of software development; if the units are incorrect, it is hard to build a correct system from them. Unit testing is becoming a common and substantial part of the software development practice: at Microsoft, for example, 79% of developers use unit tests [24], and the code for unit tests is often larger than the project code under test [23].

Creation of a test suite requires test input generation, which generates unit test inputs, and test classification, which determines whether a test passed or failed. (This paper often uses the term "test" for "test input", and it uses "classification" for determining the correctness of an execution, which is sometimes called the oracle problem.) Programmers can test manually, using their intuition or experience to make up test inputs and using either informal reasoning or experimentation to determine the proper output for each input. One alternative is to use formal specifications, which can aid both test generation and classification [4]. Such specifications are time-consuming and difficult to produce manually and often do not exist in practice. This research focuses on testing techniques that do not require a priori specifications.

In object-oriented languages, each unit test is a sequence of method calls. In this context, techniques for automated test generation include random generation (RanGen) [6, 18] and symbolic execution (SymGen) [17, 26–28, 33]. RanGen creates random sequences of method calls with random values for method arguments. SymGen executes method sequences with symbolic arguments, builds constraints on these arguments, and solves these constraints to produce actual tests with concrete values for method arguments.

Techniques for automated test classification include those based on uncaught exceptions (UCExp) [6, 17, 26, 33] and operational models (OpModel) [18, 34]. UCExp classifies a test as potentially faulty if it throws an uncaught exception. OpModel first infers an operational model [15] from the tests that programmers manually write. (Many other techniques that automate test generation neglect the manually written tests.) The operational model includes properties such as object invariants and method pre- and post-conditions. As in other model-based techniques, executions that violate the properties are classified as potentially faulty. Since OpModel also catches uncaught exceptions, we can view UCExp as OpModel with a trivial model where all properties are set to `true`.

Previous research proposed three pairs of the RanGen or SymGen generation techniques and the UCExp or OpModel classification techniques for object-oriented programs: exception-based random testing [6], model-based random testing [18], and exception-based symbolic testing [17, 26, 33]. We propose a novel pair, model-based symbolic testing, and compare these four pairs.

This paper makes the following contributions.

**New test generation approach.** We propose model-based symbolic testing. Our approach uses symbolic execution to automate both generation *and* classification of object-oriented unit tests. As in OpModel [15,18,34], our approach first infers an operational model for a set of classes under test. It then symbolically executes both the methods from these classes *and* the operational model to generate method sequences with symbolic arguments and to classify the sequences that violate the model for some constraints on the method arguments. It finally solves the constraints on these symbolic arguments and outputs a small number of (concrete) test inputs that are likely to reveal faults.

**Implementation.** We have implemented our approach in a tool called Symclat. Symclat provides automatic symbolic execution for Java, whereas several previous studies [17,26,33,34] required manual instrumentation for symbolic execution.

**Empirical study.** We present a study that empirically compares the four pairs of RanGen/SymGen generation and UCExp/OpModel classification techniques. Our study investigates the effectiveness of the four pairs in revealing faults. The main question is whether some pair subsumes another. More specifically, we compare two tools that implement automated test generation—Eclat [18] based on RanGen and our tool Symclat based on SymGen—and that both provide test classification based on OpModel and UCExp. The study uses 61 subjects that are a subset of those used in a previous study on Eclat [18].

The results show that the pairs and techniques are complementary. The two pairs based on RanGen can reveal all (but one) faults that the two pairs based on SymGen reveal. Also, RanGen can reveal faults that SymGen does not reveal. However, RanGen depends on the random seed and does not reveal all faults for all seeds. SymGen does not reveal some faults that RanGen can reveal because SymGen does not explore certain programming constructs (partly because Symclat is a prototype but more importantly because of the underlying theoretical limits of theorem provers and constraint solvers used in symbolic execution). UCExp and OpModel are complementary in revealing faults. OpModel finds some faults that violate a post-condition but do not throw an exception, so UCExp does not find them. However, OpModel misses some faults than UCExp finds as OpModel labels actually fault-revealing tests as illegal because some inferred pre-conditions are too strong.

**Suggested improvements.** Based on the above results, we give several suggestions for improving the existing techniques and tools for test generation and classification. We also provide several guidelines for using the existing tools. In summary, since the generation and classification techniques are complementary in revealing faults, we propose that all techniques be applied on the code under test.

## 2. Framework

We first describe a common framework for different techniques for test generation and classification. We then present two tools, Eclat and Symclat, that instantiate the framework.

The input to the framework is a set of classes and a model of their correct behavior (consisting of method pre- and post-conditions and object invariants). The user does not need to provide the model directly; instead, the user can provide a (passing) test suite for the classes under test, and the tools then infer the model [8]. The framework uses the model to classify the generated test inputs. The output of the framework is a set of test inputs that are potentially fault-revealing for the code under test. The framework has three components: generation, classification, and reduction.

**Test generation.** A test is a sequence of method calls (and arguments to the calls) that exercise the classes under test. This component generates tests by randomly or symbolically exploring the state space:

- **RanGen:** Generates test inputs by constructing method sequences and method arguments in a random fashion.
- **SymGen:** Generates test inputs by exhaustively exploring method sequences with symbolic variables as method arguments; translates symbolic arguments into concrete values by solving the constraints accumulated during the symbolic execution.

**Test classification.** This component executes each test (concretely or symbolically), and classifies the execution as *normal* (satisfies all pre- and post-conditions), *illegal* (violates some pre-condition) or *fault-revealing* (satisfies all pre-conditions but violates some post-condition or throws an uncaught exception). This component uses a model inferred from the input test suite or a trivial model oblivious to the existing test suite:

- **OpModel:** Labels as illegal or fault-revealing a test whose execution violates the inferred model; labels as normal a test whose execution does not violate the model. It is important to note that SymGen executes the operational model symbolically while RanGen executes it concretely.
- **UCExp:** Labels as fault-revealing a test whose execution throws an uncaught exception

**Test reduction.** This component selects a subset of the inputs labeled fault-revealing [18]; its goal is to select only one of possibly many fault-revealing test inputs that uncover the same fault. It considers two test inputs equivalent if they lead to the same violation pattern, i.e., violate the same set of model properties or throw the same exception from the same program point. Reduction outputs one representative from each equivalence set.

**Generation+classification pairs.** Three of the four pairs of random/symbolic generation and model-based/exception-based classification (RanGen+OpModel, RanGen+UCExp, SymGen+UCExp) have been previously studied [6, 18, 32–34]. This paper proposes a new pair, SymGen+OpModel. Our comparison of these four pairs uses two tools, Eclat [18] (which implements two pairs based on RanGen) and Symclat (which implements two pairs based on SymGen). We compare the four generation+classification pairs with and without reduction.

## 2.1. Eclat: Random Exploration

We next briefly describe Eclat [18], a tool for random test generation that we use to evaluate the pairs RanGen+OpModel and RanGen+UCExp. Eclat takes as input a set of classes to test and an existing test suite (that the code passes). Eclat outputs a set of tests likely to reveal faults. Eclat uses Daikon [8] to dynamically infer an operational model consisting of a set of likely program invariants. Daikon obtains the model from the execution trace of the existing test suite. Eclat instruments the classes with the inferred model to detect invariant violations by running the generated tests.

Eclat generates random test inputs, executes each of them, and detects runtime violations of the model. If the execution has no violations of invariants, Eclat classifies the input as normal. If the execution has a pre-condition violation, Eclat classifies the input as illegal and discards it. Finally, if the execution has no pre-condition violations but a post-condition violation (or the execution throws an uncaught exception), Eclat classifies the input as fault-revealing. Eclat gives all fault-revealing inputs to the reducer that selects a subset to report to the user.

## 2.2. Symclat: Symbolic Exploration

We next describe Symclat, a symbolic exploration engine that we implemented to evaluate test generation based on symbolic execution in SymGen+OpModel and SymGen+UCExp. Symclat takes as input a set of classes and an existing test suite. Symclat outputs a set of tests likely to reveal faults. Like Eclat, Symclat uses Daikon to infer an operational model based on the existing test suite. Unlike Eclat, which executes the model concretely, Symclat executes the model symbolically.

Symclat builds on our previous work on symbolic exploration of Java programs [33]. The key extension that Symclat provides is symbolic execution of model properties. Symclat also provides a significant advance in terms of implementation: Symclat has a fully automatic symbolic execution, whereas several previous projects [17, 26, 33, 34] required manual instrumentation. We next describe the important parts of Symclat.

**Path exploration.** A symbolic execution engine needs to execute each code path symbolically, and it also needs to explore (all) different paths. Execution of one path operates on a symbolic state that consists of a symbolic heap and a path condition [33]. The path condition accumulates constraints from the conditional branches encountered along the path. At each conditional branch, symbolic execution may need to explore both outcomes.

Our Symclat implementation builds on Java PathFinder (JPF) [25], an explicit-state model checker for Java. JPF works by interpreting bytecode instructions. We modified the interpretation of all bytecode instructions in JPF such that they operate on the symbolic state. We use the existing JPF features for exploring Java bytecodes, storing and restoring states, and backtracking. Combining the existing features for exploration with our symbolic execution enables symbolic exploration of method sequences.

**State representation, infeasibility, subsumption.** A symbolic state consists of a symbolic heap and a path condition. Without loss of generality [33], Symclat assumes that it should generate method sequences for only one object under test. The symbolic heap is rooted in a reference to such an object and may contain symbolic variables and expressions. A path condition consists of a conjunction of constraints and denotes the branching decisions made from the beginning of the execution to the current program point. Symbolic execution may generate infeasible paths due to unsatisfiable constraints. Symclat uses the CVC Lite [2] theorem prover to determine feasibility of path conditions and to avoid exploration of infeasible paths. Furthermore, Symclat avoids exploration of equivalent states using state subsumption introduced in our previous work [33].

**Exploration of method sequences.** Symclat uses drivers to exhaustively explore sequences of methods up to a given bound for the sequence length. Each driver specifies the methods that Symclat should explore and their arguments. In the current Symclat implementation, the user manually writes all drivers. It would be possible to automatically produce some drivers, e.g., drivers that include all public methods in the classes under test (conceptually, Eclat already uses such drivers automatically [18]). But it is often necessary to manually constrain the drivers to reduce the exploration space. The current Symclat implementation allows only primitive method arguments, but we describe below how the user can provide wrappers to handle non-primitive arguments. The arguments are symbolic variables from which symbolic execution builds symbolic expressions and path conditions. For more details on exploration, see our previous work [33].

**Model execution.** Symclat uses Daikon [8] to generate models for the classes under test. Symclat instruments these classes to check model properties at method entry and exit points. (Symclat checks object invariants and method preconditions at entry and object invariants and method postconditions at exit.) The tool discards the properties that it cannot handle due to limitations of theorem provers, for instance non-linear integer constraints. We point out that Symclat does *not* translate Daikon properties into Java code as Eclat does [18]. Such translation would enable Symclat to directly apply symbolic execution of Java code on the symbolic execution of properties. However, such translation would result in unnecessarily many execution paths for properties. Instead, Symclat uses a special (declarative) representation for Daikon properties and interprets every property in its entirety as a single execution path.

At method entry, Symclat conjoins the pre-condition to the current path condition and checks satisfiability. If it is satisfied, Symclat proceeds with the execution. If not, Symclat backtracks as generating tests along this execution path would produce illegal tests. At method exit, Symclat checks if it is possible to satisfy the *negation* of the post-condition in the context of the current path condition. If so, Symclat has found a potentially fault-revealing path, namely an execution that satisfies all pre-conditions but results in a violated post-condition. Symclat uses the POOC [20] constraint solver to generate concrete values for the symbolic variables in the path condition (conjoined with the negation of post-condition). Those symbolic variables represent arguments of methods in the test sequence. When the variables get concrete values, Symclat outputs a test that consists of method sequences with concrete arguments.

**Non-primitive arguments and wrappers.** Symclat can directly explore only methods with primitive arguments. When non-primitive arguments appear in the methods of the classes under test, the user can provide wrapper classes to Symclat. For example, consider testing the method `equals(Object)` declared in a class `C`. This method takes two arguments, so the wrapper defines two fields, one for the receiver and the other for the actual argument. The wrapper also declares operations to construct and mutate the receiver, and to copy the receiver to the argument. If the class `C` has a method `m(int)` for mutation, the wrapper looks like the following:

```
class CequalsWrapper {
  C receiver;
  Object argument;
  public void equals() { receiver.equals(argument) ; }
  public void alias() { argument = receiver ; }
  public void cons() { receiver = new C() ; }
  public void m(int i) { receiver.m(i) ; }
  ... }
```

Symclat can then use drivers for the wrappers to generate tests that operate on the wrappers instead of on the actual classes under test. Sequences of method calls (*without* non-primitive arguments) on the wrappers can then be translated into sequences of method calls (*with* non-primitive arguments) on the classes under test.

**Limitations.** In our implementation, symbolic variables can only have integer types, symbolic expressions cannot index arrays, and the operators % and / are not supported due to limitations on the underlying theorem prover and constraint solver, CVC Lite [2] and POOC [20], respectively. When these operators appear in path conditions, Symclat backtracks the execution. When symbolic expressions appear in a loop conditional, Symclat unrolls the loop a limited number of times. Thus, Symclat cannot catch infinite recursion that results in stack overflow exceptions. In addition, Symclat uses arbitrary-precision integer arithmetic provided by the theorem prover and constraint solver. As a result, Symclat does not catch errors due to integer arithmetic overflows.

Several of these limitations could be handled in improved versions of the symbolic engine. For instance, tests could be reported when the exploration reaches the bound limits set for the branching tree, making it possible to report infinite loops and stack overflows at the expense of decreasing precision. Integers could be encoded in the finite domain with bitvectors to report on arithmetic overflows and to decide on expressions with % and /, and arrays could have symbolic representations in order to allow symbolic dereferences.

## 3. Experimental Study

Our study compares four pairs of techniques discussed in Section 2 with and without reduction of test suites. We use Eclat and Symclat for test generation with RanGen and SymGen, respectively. Both Eclat and Symclat can use either UCExp or OpModel for classification. We first describe the experimental setup, then compare the pairs, and finally summarize the results of comparison.

### 3.1. Experimental Setup

We describe how we ran the tools, what subjects we used in the experiments, and how we labeled the tests produced by the tools. We ran Eclat in its default configuration: bottom-up sequence generation, four rounds of pool iteration, and maximum of 100 inputs per round per subject [18]. As the results of Eclat depend on the initial random seeds, we ran Eclat for ten different seeds. We conducted all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.15 with 2 GB memory. We set a time bound of two minutes for running test generation in Symclat for each subject. In this setup, Eclat (for each seed) and Symclat take roughly the same time, approximately five hours to generate the tests for both OpModel and UCExp, with and without reduction.

| subject | NCNB LOC | #methods |
|---|---|---|
| UBStack 8 | 88 | 11 |
| UBStack 12 | 88 | 11 |
| UtilMDE | 1832 | 69 |
| BinarySearchTree | 186 | 9 |
| StackAr | 90 | 8 |
| StackLi | 88 | 9 |
| IntegerSetAsHashSet | 28 | 4 |
| Meter | 21 | 3 |
| DLList | 286 | 12 |
| E_OneWayList | 171 | 10 |
| E_SLList | 175 | 11 |
| OneWayList | 88 | 12 |
| OneWayNode | 65 | 10 |
| SLList | 92 | 12 |
| TwoWayList | 175 | 9 |
| RatPoly (46 versions) | 582.51 | 17.20 |

**Figure 1. Size of the subjects.**

**Subjects.** Figure 1 lists the subject programs used in our experiments. We show the number of non-comment-non-blank (NCNB) lines of code and the total number of methods for each subject. UBStack is the implementation of the unique bounded stack used in previous studies on testing [6, 18, 22, 34]. This code comes with two test suites, consisting of 8 and 12 test cases. UtilMDE is a supporting class from Daikon. BinarySearchTree, StackAr, and StackLi are taken from a textbook [30] that provides a set of example uses for the classes. The next nine subjects are example classes from the distribution of Java Modeling Language (JML) [16], together with formal specifications (written in JML). RatPoly refers to 46 student solutions to an assignment in the MIT class 6.170. The assignment asked the students to implement the core operations for rational polynomials. The course staff provided some supporting classes and a test suite to the students. The numbers in the table are averages over 46 different implementations.

Our subjects are a subset of those used in a previous study on Eclat [18]. We selected all subjects that Symclat can currently explore; the study on Eclat used 631 implementations, but approximately 90% of them contain programming constructs that Symclat cannot currently explore. Moreover, RanGen can explore all methods from the subjects, while SymGen may not be able to execute all methods even for the selected subjects. In UtilMDE, for example, the current Symclat implementation can explore only the two (overloaded) `create_combinations` methods as the other 67 methods heavily use `String` objects, `float` numbers, or other constructs not currently supported.

**Test labeling.** We next describe how we label the tests generated by different techniques. The goal is to determine the tests that reveal *actual* faults. We use formal JML specifications (used in the Eclat study) for each subject and detect tests that violate the specifications. However, we separate arithmetic (integer) overflows that several tests for RatPoly implementations produce. This potential type of fault is inherent in all RatPoly subjects because the problem set asked the students to use a staff-provided class for rational numbers based on the fixed-precision numbers (Java 32-bit `int` numbers). Since the staff-provided class can overflow, the student-written code can produce results that differ from those based on an arbitrary-precision numbers, but these results should not be considered faults. We instrumented the staff-provided class to detect overflows during test execution, and we discard from the generated tests those that result in overflows.

We label each test as follows. A test that produces a JML violation or throws an uncaught exception *before* an overflow is labeled as a *failing test*. Such test reveals a mismatch between the code and the specification, and in our study, every such test indeed revealed an actual fault in either the code or the specification.

**Distinct faults.** We also determine for each test suite the number of distinct specification violations. The rationale is that the number of distinct faults revealed by a test suite may be more valuable than the number of fault-revealing tests. For example, a test suite with ten fault-revealing tests that all reveal the same fault may be less useful than a test suite with ten tests of which only two reveal faults but distinct faults. Each test that violates a specification either throws an uncaught exception (violating the implicit specification that the code should not throw uncaught exceptions) or violates some explicit part of the specification (object invariant, internal method pre-condition, or method post-condition).

We label two tests as having the same violation if they either violate the same part of the specification *at the same program point* or throw the same exception *at the same program point*. This is similar to the reduction described in Section 2 but uses JML specifications instead of the inferred operational models. We found this approach to be precise for our experiments: two failing tests are labeled with distinct specification violations if and only if these tests reveal distinct actual faults in the subjects.

## 3.2. Comparison of Techniques

Figure 2 shows the results of running the subjects in all four pairs with reduction of generated test suites. The columns show the number of non-arithmetic-overflow tests generated (TG); the number of failing tests (FT) that either violate JML specifications or raise uncaught exceptions; and the precision (Pr), which shows the ratio of fault-revealing tests, FT/TG. The number of false positives is the complement of the precision, TG-FT. The FT column also shows the number of distinct faults in the parentheses. Since Eclat uses random generation, its results depend on the ran-

| subject | RanGen+OpModel | | | RanGen+UCExp | | | SymGen+OpModel | | | SymGen+UCExp | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #TG | #FT | Pr. | #TG | #FT | Pr. | #TG | #FT | Pr. | #TG | #FT | Pr. |
| UBStack (8) | 3.0 | 2.0(2.0) | 0.67 | 2.0 | 2.0(2.0) | 1.00 | 3 | 2(2) | 0.67 | 2 | 2(2) | 1.00 |
| UBStack (12) | 1.0 | 1.0(1.0) | 1.00 | 2.0 | 2.0(2.0) | 1.00 | 1 | 1(1) | 1.00 | 2 | 2(2) | 1.00 |
| UtilMDE | 10.8 | 1.7(1.7) | 0.15 | 11.3 | 1.6(1.6) | 0.14 | 2 | 0(0) | 0.00 | 10 | 0(0) | 0.00 |
| DLList | 5.0 | 0.9(0.9) | 0.18 | 5.0 | 1.0(1.0) | 0.20 | 4 | 0(0) | 0.00 | 4 | 0(0) | 0.00 |
| E_OneWayList | 0.0 | 0.0(0.0) | - | 15.0 | 0.4(0.4) | 0.03 | 4 | 0(0) | 0.00 | 4 | 0(0) | 0.00 |
| OneWayList | 6.0 | 0.9(0.9) | 0.15 | 14.8 | 1.0(1.0) | 0.07 | 6 | 0(0) | 0.00 | 6 | 0(0) | 0.00 |
| SLList | 3.0 | 0.8(0.8) | 0.26 | 7.0 | 0.9(0.9) | 0.13 | 6 | 0(0) | 0.00 | 6 | 0(0) | 0.00 |
| TwoWayList | 0.0 | 0.0(0.0) | - | 22.3 | 3.0(1.0) | 0.14 | 2 | 0(0) | 0.00 | 2 | 0(0) | 0.00 |
| s5 | 0.1 | 0.0(0.0) | 0.00 | 1.5 | 0.3(0.3) | 0.15 | 1 | 0(0) | 0.00 | 7 | 1(1) | 0.14 |
| s7 | 20.0 | 0.8(0.5) | 0.04 | 17.1 | 0.2(0.2) | 0.01 | 4 | 1(1) | 0.25 | 9 | 1(1) | 0.11 |
| s13 | 0.9 | 0.4(0.4) | 0.43 | 1.5 | 0.3(0.3) | 0.15 | 2 | 0(0) | 0.00 | 7 | 1(1) | 0.14 |
| s14 | 2.3 | 1.0(0.6) | 0.38 | 3.1 | 1.3(1.0) | 0.42 | 3 | 0(0) | 0.00 | 7 | 0(0) | 0.00 |
| s21 | 0.0 | 0.0(0.0) | - | 14.0 | 5.4(1.1) | 0.39 | 0 | 0(0) | - | 12 | 1(1) | 0.08 |
| s24 | 0.7 | 0.6(0.5) | 0.92 | 1.4 | 0.3(0.3) | 0.15 | 2 | 1(1) | 0.50 | 7 | 1(1) | 0.14 |
| s33 | 2.0 | 2.0(1.0) | 1.00 | 3.0 | 1.9(1.0) | 0.64 | 3 | 1(1) | 0.33 | 8 | 1(1) | 0.12 |
| s39 | 1.1 | 1.0(1.0) | 0.95 | 1.9 | 0.9(0.9) | 0.45 | 1 | 0(0) | 0.00 | 6 | 0(0) | 0.00 |
| s46 | 0.3 | 0.3(0.3) | 1.00 | 1.5 | 0.3(0.3) | 0.15 | 2 | 1(1) | 0.50 | 7 | 1(1) | 0.14 |
| Total (17 subjects) | 56.2 | 13.4(11.6) | 0.51 | 124.4 | 22.8(15.3) | 0.31 | 46 | 7(7) | 0.20 | 106 | 11(11) | 0.17 |
| 4 other subjects | 0 | 0(0) | - | 0 | 0(0) | - | 0 | 0(0) | - | 0 | 0(0) | - |
| 40 other subjects | 57 | 0(0) | 0 | 86 | 0(0) | 0 | 53 | 0(0) | 0 | 234 | 0(0) | 0 |

**Figure 2. Comparison of generation+classification pairs with reduction of test suites.**

dom seed, and the columns for RanGen show the *average* results for ten different seeds.

For 4 of the 61 subjects, no pair generated any test. For 40 subjects, the pairs generated some tests but no test resulted in any failure. These 44 subjects appear at the bottom of Figure 2; the fewer tests (false positives) a pair generates for these subjects, the better it is. Figure 2 shows detailed results for the 17 subjects for which at least one pair found a fault.

We next discuss the fault-revealing effectiveness of RanGen and SymGen generation techniques with OpModel and UCExp classification techniques.

**Complementary test generation techniques.** RanGen and SymGen are complementary techniques for test generation. SymGen is deterministic and does not depend on random seeds. However, SymGen often cannot explore code (which contains certain programming constructs); we ran Symclat on only 61 out of 631 subjects from the Eclat study [18], and Symclat cannot even explore all the code from these 10% of subjects. A better implementation of SymGen could explore more subjects, but SymGen cannot explore arbitrary code due to the theoretical limitations of the underlying technology (incompleteness of theorem provers and constraint solvers due to undecidability). RanGen can, in contrast, explore all code and thus reveal some faults that SymGen misses. However, which faults RanGen reveals depends on the random seed used to generate test inputs.

**Sensitivity of RanGen.** RanGen is sensitive to the choice of random seed. For some subjects, RanGen generates a fault-revealing test for every seed (e.g., s33); sometimes, this is because RanGen generates (almost) the same results for all ten seeds (e.g., UBStack 8/12). In other cases, RanGen generates a fault-revealing test only for some of the ten seeds (3–9 in RanGen+OpModel and 2–9 in RanGen+UCExp). Consider, for example, the subject code s7. RanGen+OpModel generates on average 20 tests for this subject, i.e., total of 200 tests for ten seeds. Only 8 of these 200 tests reveal a fault, and for only five of the ten seeds is there at least one test that reveals the fault.

**Missing faults.** RanGen can miss faults for two reasons: (i) it does not generate a fault-revealing method sequence, or (ii) it does generate a potentially fault-revealing sequence but does not generate the appropriate argument values. For example, subjects s5, s13, s24, and s46 have a fault in the method `div` of the rational polynomial class. RanGen does not detect the fault for every seed as it does not select the appropriate values for the method arguments in the tests. As another example, subject s37 has a fault in the method `add`, but revealing this fault requires that the `NaN` polynomial be passed to `add`. RanGen does not often generate a method sequence that builds the `NaN` polynomial.

SymGen misses faults (that RanGen finds) for two main reasons: (i) it does not generate long enough method sequences, or (ii) it is given incomplete drivers and wrappers. For example, RanGen detects a real fault in UtilMDE as `create_combinations` ends up in an infinite recursion when two of its arguments are 0. RanGen detects this scenario because the test execution results in a stack overflow. SymGen misses the fault as the stack overflow requires a long execution sequence. RanGen also reveals a fault in UtilMDE in the method `replaceString`, which gets into an infinite loop when one of the arguments is the empty string. In this case, the execution exceeds the maximum heap size due to the allocation of strings in the loop. (Without heap overflow, RanGen would instead report that the infinite-loop execution timed out.) As discussed, SymGen does not even explore methods with string arguments. For s21 and s33 (without reduction, Figure 3), RanGen reveals a fault that requires a short execution, but SymGen misses this fault as its exploration time expires before reaching this particular execution.

As another example, for 5 of the 12 subjects from the textbook and JML samples (`DLList`, `E_OneWayList`, `OneWayList`, `SLList`, and `TwoWayList`), RanGen generates some tests that violate the formal JML specifications. All these tests give as the input to the `toString` methods a list that contains itself, which leads to infinite recursion. Since the pre-conditions for `toString` do not specify that the list cannot contain itself, these tests violate (total) JML specifications. But, the preferred way to correct these "faults" seems to be changing the specification (e.g., to be partial) rather than the implementation. SymGen misses all these faults since the drivers and wrappers (discussed in Section 2.2) do not allow a list to be added to itself.

**False positives.** All techniques generate some false positives, i.e., tests that do not actually reveal faults. For example, for UBStack 8, both _+OpModel pairs generate three tests, two of which expose two different faults, but one of which is a false positive due to an incorrectly inferred invariant. For UBStack 12, each _+OpModel pair generates only one test that detects an actual fault. This increases the precision of the test suite but also misses the other fault. UBStack 12 differs from UBStack 8 only in the tests from which the operational model is inferred. (Thus, the results for these two subjects differ only for the two pairs based on OpModel.) The operational model for UBStack 12 includes a too strong pre-condition. Namely, the pre-condition states that the argument of `equals` should not be `null`, so the tests that set it to `null` are classified as illegal, although they are legal and fault-revealing.

As another example, the pairs based on RanGen generate some false positives for `DLList`, `E_OneWayList`, `OneWayList`, `SLList`, and `TwoWayList`. This is due to the weak method pre-conditions (in the inferred operational models for OpModel and `true` for UCExp). For instance, an actual specification for `OneWayList` is that a read of the next element of a list should throw an exception if the iterator is past the end of the list. This property is present only in the formal specification, and thus some tests classified with uncaught exceptions lead to this exceptional but correct behavior.

**Complementary test classification techniques.** UCExp and OpModel are complementary techniques for test classification. In the experiments, UCExp revealed more faults than OpModel (total of 15.3+11 to 11.6+7 distinct faults). But each technique revealed some faults that the other missed. The non-trivial models can make OpModel better or worse than UCExp. In general, OpModel is worse when it labels as illegal a test that violates an incorrectly inferred pre-condition although the test is legal and throws an exception, and OpModel is better when it labels as potentially fault-revealing a test that violates a post-condition although the test does not throw an exception.

### 3.3. Reduction

Figure 3 shows the results of running the subjects in all four pairs without reduction of generated test suites. We highlight the cases where test suites without reduction reveal more faults than with reduction. But test suites without reduction often have lower precision than with reduction (although the precision can be also higher, depending on the quality of the inferred invariants). In other words, reduction decreases the fault-revealing effectiveness, while slightly increasing the (average) precision. More importantly from the user's perspective, reduction decreases the absolute number of tests that need to be manually inspected.

We next discuss the cases when reduction removes fault-revealing tests. For four subjects (UtilMDE, s7, s14, and s33) and various generation+classification pairs, reduction removes all tests that reveal one of the faults but still keeps some tests that reveal other faults. For two subjects (s37 and s42), reduction removes all fault-revealing tests. (Thus, these subjects have no separate rows in Figure 2.) Note also that reduction removes all tests for some random seeds in pairs based on RanGen. For example, for `E_OneWayList` and RanGen+UCExp, the fault is revealed for all ten seeds without reduction but for only four seeds with reduction. Reduction removes fault-revealing tests because the inferred operational model for OpModel or `true` for UCExp differ from the actual specification for the code; two tests that do (not) violate the model in an equivalent way can violate the actual specification in different ways.

We finally discuss the effect of reduction on the precision and the size of generated test suites. For all four pairs, reduction improves the average precision for subjects *with* faults. For example, for RanGen+UCExp, the precision goes from 0.20 to 0.31; to find a fault-revealing test the user needs to inspect, on average, about 3 tests (with reduction) instead of 5 tests (without reduction). This improvement is important, but an even bigger improvement is for subjects *without* faults. For them, the precision is 0, and the user needs to inspect all generated tests to determine that none reveals a fault. Reduction decreases the number of tests an order of magnitude, from 723–4,805 (for various pairs without reduction) to 53–234 (for various pairs with reduction).

### 3.4. Improving Test Generation

We give several suggestions for improving the existing techniques and tools for test generation and classification. We also give suggestions for using such tools.

RanGen may be improved by running it multiple times; it finds nearly all faults that SymGen does, for at least one

| subject | RanGen+OpModel | | | RanGen+UCExp | | | SymGen+OpModel | | | SymGen+UCExp | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #TG | #FT | Pr. | #TG | #FT | Pr. | #TG | #FT | Pr. | #TG | #FT | Pr. |
| UBStack (8) | 29.7 | 19.6(2.0) | 0.66 | 17.0 | 17.0(2.0) | 1.00 | 118 | 75(2) | 0.64 | 91 | 90(2) | 0.99 |
| UBStack (12) | 10.9 | 10.9(1.0) | 1.00 | 17.0 | 17.0(2.0) | 1.00 | 26 | 25(1) | 0.96 | 91 | 90(2) | 0.99 |
| *UtilMDE* | 166.4 | 14.5(1.8) | 0.09 | 174.9 | 22.7(*2.2*) | 0.13 | 2 | 0(0) | 0.00 | 10 | 0(0) | 0.00 |
| DLList | 332.8 | 18.0(1.0) | 0.06 | 299.8 | 18.3(1.0) | 0.08 | 75 | 0(0) | 0.00 | 75 | 0(0) | 0.00 |
| E_OneWayList | 0.0 | 0.0(0.0) | - | 803.6 | 10.6(1.0) | 0.02 | 235 | 0(0) | 0.00 | 235 | 0(0) | 0.00 |
| OneWayList | 247.5 | 18.9(1.0) | 0.08 | 715.7 | 14.3(0.9) | 0.02 | 1,973 | 0(0) | 0.00 | 1,973 | 0(0) | 0.00 |
| SLList | 244.7 | 34.8(1.0) | 0.14 | 560.6 | 29.6(1.0) | 0.06 | 1,983 | 0(0) | 0.00 | 1,983 | 0(0) | 0.00 |
| TwoWayList | 0.0 | 0.0(0.0) | - | 889.2 | 29.8(1.0) | 0.04 | 61 | 0(0) | 0.00 | 61 | 0(0) | 0.00 |
| s5 | 0.1 | 0.0(0.0) | 0.00 | 26.3 | 0.4(0.3) | 0.02 | 28 | 0(0) | 0.00 | 135 | 2(1) | 0.01 |
| *s7* | 349.2 | 22.4(1.0) | 0.06 | 326.9 | 20.0(1.0) | 0.06 | 134 | 44(1) | 0.33 | 698 | 218(*2*) | 0.31 |
| s13 | 1.6 | 0.7(0.4) | 0.50 | 26.3 | 0.4(0.3) | 0.02 | 16 | 0(0) | 0.00 | 73 | 1(1) | 0.01 |
| *s14* | 8.6 | 3.0(0.7) | 0.34 | 33.0 | 4.0(1.0) | 0.12 | 34 | 8(*2*) | 0.24 | 112 | 25(*1*) | 0.22 |
| s21 | 0.0 | 0.0(0.0) | - | 64.9 | 27.9(1.1) | 0.42 | 0 | 0(0) | - | 317 | 70(1) | 0.22 |
| s24 | 0.8 | 0.7(0.6) | 0.92 | 26.2 | 0.4(0.3) | 0.02 | 11 | 1(1) | 0.09 | 53 | 1(1) | 0.02 |
| *s33* | 26.0 | 25.5(*1.2*) | 0.97 | 57.0 | 30.8(*1.4*) | 0.53 | 32 | 7(1) | 0.22 | 126 | 15(1) | 0.12 |
| *s37* | 0.3 | 0.0(0.0) | 0.00 | 25.8 | 0.0(0.0) | 0.00 | 24 | 0(0) | 0.00 | 111 | 1(*1*) | 0.01 |
| s39 | 12.1 | 11.7(1.0) | 0.97 | 38.3 | 12.4(0.9) | 0.31 | 20 | 0(0) | 0.00 | 54 | 0(0) | 0.00 |
| *s42* | 235.5 | 0.1(*0.1*) | 0.00 | 238.7 | 0.0(0.0) | 0.00 | 25 | 0(0) | 0.00 | 177 | 0(0) | 0.00 |
| s46 | 0.4 | 0.4(0.3) | 1.00 | 26.3 | 0.4(0.3) | 0.02 | 31 | 4(1) | 0.13 | 301 | 2(1) | 0.01 |
| Total (19 subjects) | 1,666.6 | 181.2(13.1) | 0.42 | 4,367.5 | 256.0(17.7) | 0.20 | 4,828 | 164(9) | 0.14 | 6,676 | 515(14) | 0.15 |
| 4 other subjects | 0 | 0(0) | - | 0 | 0(0) | - | 0 | 0(0) | - | 0 | 0(0) | - |
| 38 other subjects | 723 | 0(0) | 0 | 1,546 | 0(0) | 0 | 1,417 | 0(0) | 0 | 4,805 | 0(0) | 0 |

**Figure 3. Comparison of generation+classification pairs without reduction of test suites.**

of the ten seeds. (It does not find the fault in s33.) Running RanGen multiple times increases generation time; in our experimental setup, each run of RanGen takes about as much time as the run of SymGen. But a more important question for the future work is how to combine the test suites generated by multiple runs in order to reduce the user time necessary to inspect the combined test suite.

RanGen may be improved by biasing its selection to boundary and special values that are more likely to reveal faults. For instance, in RatPoly subjects, RanGen should bias selection to 0 for integers and to `NaN` for polynomials and not select the values uniformly from a large pool. The tools should allow the users to specify such values. The tools may also try to determine such values by static or dynamic analysis of code.

RanGen may be improved by selecting values that satisfy certain relationships. For example, to generate an argument value in a method sequence, RanGen can bias its selection to the values already selected in that sequence. This leads to selecting equal values in a sequence. The tools should allow the users to provide heuristics for selection [5].

In addition to biasing selection of inputs to method calls, RanGen may benefit from biasing the selection of method calls. For example, an error that RanGen was unable to find for several seeds requires using two equivalent polynomials as arguments to a method call. Equivalent polynomials can be obtained by repeating a sequence of method calls that creates one polynomial, but purely random generation is unlikely to produce such repetition.

SymGen may be improved by combining it with RanGen. A recent proposal is to do that by executing code with both symbolic values and random values [3, 11, 21]. But there are many other ways, such as randomly choosing method sequences and then symbolically exploring the argument values or exhaustively choosing the sequences but randomly choosing the argument values, as suggested elsewhere [33]. It would be also important to empirically compare these technique with (biased) random generation for a large number of subjects.

Users should use both UCExp and OpModel classifications with the tools. These two classifications complement each other and can be easily used in any tool that provides OpModel. (Recall that UCExp is just a special case with all model properties being `true`.) Additionally, it is helpful when users can provide some actual specifications as they enable more precise classification. In general, tools may combine any number of classifiers in addition to UCExp and OpModel and classify a test as potentially fault-revealing as long as any of the classifiers finds it fault-revealing; such combination increases the number of revealed faults, but it would be necessary to evaluate the size of the generated test suites and their precision.

Users should initially apply reduction as it increases the precision of the generated test suites and decreases their size. But reduction can miss faults, so when more resources are available for testing, users should also inspect the entire generated test suites, without reduction. We leave it as the future work to evaluate what results users would achieve in different inspection scenarios.

### 3.5. Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, faults, manually written test cases, and testing tools are representative of true practice. The subject programs except for UtilMDE are relatively small. These threats could be reduced by more experiments on a larger set of subjects and tools. The threats to internal validity are implementation effects that can bias our results. Faults in our two tools might cause such effects. To reduce these threats, we manually inspected the results for all subject programs for SymGen and for one of the random seeds for RanGen. One threat to construct validity is that our experiments use the violations of manually written JML specifications as symptoms of fault exposure. These JML specifications may be weak and thus miss some faults that are in fact exposed by the generated tests.

## 4. Related Work

There are numerous comparisons of techniques for test generation (or selection), and many comparisons take random testing as the baseline. For example, Duran and Ntafos [7] and Hamlet and Taylor [13] empirically compared random testing and partition testing such as path testing. They observed that random testing could be a cost-effective alternative to path testing. Frankl and Weiss [10] experimentally compared branch testing, dataflow testing, and random testing. They observed that branch or dataflow testing performed somewhat better than random testing on most subjects. Weyuker and Jeng [31] analytically showed that partition testing is more effective than random testing when at least one subdomain of the partition has a high concentration of fault-revealing inputs. Our empirical comparison considers test generation *and* test classification techniques (with and without reduction).

There are several comparative studies of static fault-finding tools. Rutar et al. [19] compared five such tools against five open source projects. Their results show that none of the five tools strictly subsumes another in terms of the fault-finding capability. They proposed a meta-tool for combining and correlating the outputs of these five tools. Wagner et al. [29] did a case study that applied three static fault-finding tools as well as code review and manual testing to several industrial projects. Their study showed that the static tools predominantly find different faults than manual testing but a subset of faults found by reviews. They proposed a combination of these three types of techniques. Zitser at al. [35] applied five static fault-finding tools on three open source projects and found the average rates of false positives produced by these tools to be high. Our comparative study focuses on different (dynamic) test generation and classification techniques, not static fault-finding tools.

Symclat developed in this research extends the work on using inferred operational models to guide test-input generation and execution classification. Harder et al. [15] developed the operational difference technique that automatically generates or augments an existing test suite by adding test cases until the inferred operational model stops changing. Hangal and Lam [14] developed the DIDUCE tool that continuously checks a program's behavior against the incrementally inferred operational models. Jov [34] and Eclat [18] tools automatically use models inferred by Daikon [8]. Agitar Agitator [1], a commercial tool, also infers operational models from test executions but suggests these models to developers to manually and selectively promote them to assertions. In contrast to these previous tools, Symclat uses *symbolic execution* to systematically explore the code under test *and* the operational models.

Various test-generation tools have also been developed for object-oriented programs that are not equipped with specifications. For example, JCrasher [6] generates random method sequences for producing non-primitive method arguments and some default values for primitive arguments. Visser et al. [26] used the Java PathFinder model checker [25] to systematically explore the object-state space of the class under test, concretely or symbolically. Rostra [32] concretely explores the object-state space in a similar way, while Symstra [33] uses symbolic execution and prunes state exploration based on the symbolic-state comparisons. More recently, several testing tools for C programs—DART [11], EGT [3], and CUTE [21]—have been developed to combine random generation and symbolic execution. There are also many approaches to automated test generation not based on random or symbolic execution. As just two examples, Ferguson and Korel proposed the chaining approach [9], and Gupta et al. [12] proposed the use of iterative relaxation method. Our study considers only random and symbolic execution, but it also considers classification. Our tool Symclat improves on the existing tools (based on symbolic execution) by using the inferred operational models for classification.

## 5. Conclusions

We have presented an empirical comparison of automated generation and classification techniques for object-oriented unit testing. Specifically, we have compared pairs of test-generation techniques based on random generation or symbolic execution and test-classification techniques based on uncaught exceptions or operational models. Our study shows that the techniques are complementary in revealing faults. Therefore, the tools and users should apply several techniques on the same code under test. In the future, the comparison should be extended to more generation and classification techniques. Also, it would be useful to investigate how to combine various techniques to increase the effectiveness of revealing faults while not decreasing the precision of generated test suites.

**Acknowledgments.** We thank Willem Visser for helping us modify Java PathFinder, Sarfraz Khurshid for providing us an instrumentation package for symbolic execution, Chao Liu for his advice on comparing different techniques, and the anonymous reviewers for their comments on a previous version of this paper. This work was partially supported by CAPES fellowship under grant #15021917, an NSF graduate fellowship, and NSF grant CCR-0133580.

## References

[1] Agitar Agitator 3.0, 2005. http://www.agitar.com/.

[2] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th CAV*, pages 515–518, July 2004.

[3] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking Software*, pages 2–23, August 2005.

[4] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. 16th ECOOP*, pages 231–255, June 2002.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN ICFP*, pages 268–279, 2000.

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

[7] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[9] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[10] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.*, 19(8):774–787, August 1993.

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

[12] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proc. 6th ACM SIGSOFT FSE*, pages 231–244, 1998.

[13] R. G. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. Software Eng.*, 16(12):1402–1411, 1990.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th ICSE*, pages 291–301, 2002.

[15] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. 25th ICSE*, pages 60–71, 2003.

[16] JML website, 2006. http://www.jmlspecs.org/.

[17] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th TACAS*, pages 553–568, April 2003.

[18] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th ECOOP*, pages 504–527, July 2005.

[19] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. 15th IEEE ISSRE*, pages 245–256, November 2004.

[20] H. Schlenker and G. Ringwelski. POOC: A platform for object-oriented constraint programming. In *Proc. 2002 International Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–170, June 2002.

[21] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 5th ESEC/FSE*, pages 263–272, September 2005.

[22] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.

[23] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. Technical Report MSR-TR-2005-153, Microsoft Research, Redmond, Washington, November 2005.

[24] G. Venolia, R. DeLine, and T. LaToza. Software development at Microsoft observed. Technical Report MSR-TR-2005-140, Microsoft Research, Redmond, WA, Oct. 2005.

[25] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE ASE*, pages 3–12, 2000.

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

[27] W. Visser, C. S. Pasareanu, and R. Pelánek. Test input generation for red-black trees using abstraction. In *Proc. 20th IEEE/ACM ASE*, pages 414–417, November 2005.

[28] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *Proc. 17th IEEE ASE*, pages 149–160, September 2002.

[29] S. Wagner, J. Jurjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, pages 40–55, May 2005.

[30] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1999.

[31] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Software Eng.*, 17(7):703–711, 1991.

[32] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE ASE*, pages 196–205, Sept. 2004.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th TACAS*, pages 365–381, April 2005.

[34] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE ASE*, pages 40–48, 2003.

[35] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. 12th ACM SIGSOFT FSE*, pages 97–106, 2004.