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

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

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

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

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

E-mail: {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. Techniques for automating these activities include automated test generation based on symbolic execution or random generation and automated test classification based on uncaught exceptions or operational models (more precisely, violations of the models inferred from manually provided tests). Previous research on unit testing for object-oriented programs developed three combinations of these techniques: exception-based symbolic testing, model-based random testing, and exception-based random testing. In this research, we have developed a novel combination, model-based symbolic testing. This paper presents an empirical study that compares all four combinations of these generation and classification techniques. The empirical study uses 61 subjects taken from a variety of sources. The results show that the techniques are complementary (i.e., detect different faults) and illustrate 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, a recent survey found that 79% of developers use unit tests [24] and in many projects, the code for unit tests is larger than the code for the project under test [23].

Creation of a test suite requires test generation, which generates unit tests, and test classification, which determines whether a test passed or failed. (This paper uses the term "classification" for determining the correctness of an execution, which is sometimes called the oracle problem.) Programmers often perform some test generation and test classification by hand, but it is tedious and error-prone to systematically perform testing by hand. Programmers use their intuition or experience to make up test inputs, and use either informal reasoning or experimentation to determine the proper output for each input. One alternate technique is to use formal specifications, which can aid both test generation and test 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. We focus on automated testing as it can significantly reduce the cost of software development and maintenance.

Researchers proposed several techniques that automate test generation and classification for unit testing of object-oriented programs [6, 16, 18, 26, 32, 33]. In object-oriented languages, each unit test is a sequence of method calls.

Techniques for automated test generation include random generation (RanGen) [6, 18], symbolic execution (SymGen) [16, 26–28, 32], and a recent mix of RanGen and SymGen [3, 12, 21]. RanGen creates random sequences of method calls with random values for method parameters. SymGen [17] executes method sequences with symbolic parameters, builds constraints on these parameters, and solves these constraints to produce actual tests with concrete values for method parameters.

Techniques for automated test classification include those based on uncaught exceptions (UCExp) [6, 16, 26, 32] and operational models (OpModel) [18, 33]. UCExp classifies a test as potentially faulty if it throws an uncaught exception. OpModel is a technique for automatically creating an approximate model (or oracle) that is inferred from the tests that programmers manually generate; many other techniques that automate test generation neglect the manually generated tests. OpModel first infers an operational model [14] that includes properties such as class 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 combinations 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 [16, 26, 32].

This paper makes the following contributions.

New Testing Approach: We propose a novel combination, model-based symbolic testing. The approach uses symbolic execution to automate both generation and classification of object-oriented unit tests. As in OpModel [14, 18, 33], 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 parameters and to classify the sequences that violate the model for some constraints on the method parameters. It finally solves the constraints on these symbolic parameters and outputs a small number of (concrete) test inputs that are likely to reveal faults. In principle, our new technique may allow operational models to more effectively guide test generation for violating the models than the previous techniques [18, 33].

Implementation: We have implemented our approach in a tool called Symclat. Symclat provides automatic symbolic execution for Java, whereas several previous studies [16,26,32,33] 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. 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 each support test classification based on OpModel and UCExp. Our study investigates the following two questions: (i) Is RanGen or SymGen more effective in revealing faults? and (ii) Is UCExp or OpModel more effective in revealing faults? Answering these questions not only provides insights for tool developers to improve the existing tools but also guidelines for tool users to understand how to use existing tools. Our empirical study uses 61 subjects originally taken from a variety of sources and used in a previous study on Eclat [18].

Suggested Improvements: Based on the previous findings, we give several suggestions for improving the existing techniques and tools for test generation and classification and for using such tools. Since the techniques are complementary in detecting faults, we propose that all techniques be used on the same code under test.

2. Framework

We describe different techniques for test generation and classification in terms of a common framework and present two tools, Eclat and Symclat, that implement instantiations of the framework. The input to the framework is a set of classes and a model of the correct behavior of the classes consisting of method pre- and post-conditions and object invariants. The framework uses the model to determine if a test input is normal, illegal, or fault-revealing. The framework has three components: generation, classification, and reduction:

1. **Test generation.** A test is a sequence of method calls (and arguments to the calls) that exercise the code under test. This component generates tests by exploring the state space in a random or systematic fashion.
2. **Test classification.** This component executes each test (concretely or symbolically), and based on the result of its execution, classifies the test as *normal* (the execution satisfies all pre- and post-conditions), *illegal* (the execution violates some pre-condition) or *fault-revealing* (the execution satisfies all pre-conditions but violates some post-condition).
3. **Test reduction.** This component selects a subset of the inputs labeled *fault-revealing* to provide a better guidance to the user. The component attempts to select only one of possibly many fault-revealing test inputs that uncover the same fault.

The framework outputs a (reduced) set of test inputs that are potentially fault-revealing for the code under test.

Figure 1 characterizes the four pairs of random/symbolic generation and model-based/exception-based classification that we evaluate in this paper. For the two OpModel techniques, a set of existing test cases is provided as input in order to infer an operational model of correct behavior of the input classes. The UCExp techniques use a trivial model and do not take advantage of the existing test cases.

Our study compares the four pairs with and without reduction. We have implemented in Symclat the same reduction used in Eclat [18]. It considers two test inputs to be equivalent if they lead to the same violation pattern, i.e., they violate the same set of model properties (if any) or throw the same exception from the same program point (if any). Reduction outputs one representative from each set of equivalent test inputs.

Three of the four pairs (model-based random testing, exception-based random testing, and exception-based symbolic testing) have been previously studied in isolation [6, 18, 31–33]. One contribution of this paper is a proposal for model-based symbolic testing, which has not been previously studied to the best of our knowledge. Our comparison uses two tools, Eclat and Symclat. Eclat [18] implements model-based random testing and exception-based random testing. Symclat is a novel symbolic execution engine that we engineered for this study; it implements model-based symbolic testing and exception-based symbolic testing.

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 (1) a set of classes to test and (2) an existing test suite (known to be correct). Eclat's output is a set of tests that are likely to reveal faults not exposed by the existing test suite.

Eclat uses Daikon [9] to dynamically infer an operational model consisting of a set of likely program invariants. The model is obtained by observing the execution trace of the existing test suite. Eclat instruments the classes with the invariants that it gets from Daikon so that it can detect invariant violations at runtime.

Eclat generates random test inputs, executes each of them, and detects and runtime violations of the model. If the execution leads to no violations of invariants, the input is classified as normal. If the execution leads to a pre-condition violation, the input is classified as illegal and discarded. Finally, if the execution leads to no pre-condition violations and some post-condition violations (or the execution throws an uncaught exception), the test input is classified as fault-revealing. All fault-revealing inputs are given to the reducer, which selects a subset to report to the user.

2.2. Symclat: Symbolic Exploration

This section describes 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, and outputs test cases that are likely to reveal faults. Like Eclat, Symclat uses Daikon to derive an operational model based on the existing test suite. Unlike Eclat, Symclat explores the model symbolically.

Symclat builds on our previous work on symbolic exploration of Java programs [32]. The key conceptual extension that Symclat provides is symbolic execution of model invariants. Symclat also provides a significant advance in terms of implementation: Symclat has a fully automatic symbolic execution, whereas several previous projects [16, 26, 32, 33] required manual instrumentation. Without an automatic symbolic engine, we could not easily conduct a large study of SymGen. 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 [32]. 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. In general, execution of a single path can be implemented by modifying the virtual machine or instrumenting the program. The exploration of different paths can be implemented in a in a stateful manner as done for example in JPF or in a stateless manner with code re-execution as done say in Verisoft [11]. Since JPF already implements a Java virtual machine (in Java), we chose to modify the virtual machine for execution of one path and to perform stateful exploration of paths. JPF works by interpreting bytecode instructions. We have modified the interpretation of all bytecode instructions in JPF such that they operate on the symbolic state. We use the support that JPF provides for exploring Java bytecodes, storing and searching states, backtracking, and so on. Our version allows symbolic expressions to be used as values and thus to be stored in heap or local variables, to be passed as arguments to methods, return values, etc. Combining JPF's features for exploration together with our symbolic expressions allows exhaustive exploration of method sequences.

State representation, infeasibility, subsumption. A symbolic state consists of a symbolic rooted heap and a path condition [32]. Without loss of generality, Symclat assumes that there is one object under test for which method sequences are generated. The symbolic heap is rooted in a reference to such an object and may contain symbolic variables and expression. A path condition is composed of a set of constraints accumulated up to a program point and denotes the decisions (on symbolic variables) that have been 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 by implementing state subsumption introduced in our previous work [32].

Exploration of method sequences. Symclat uses drivers to explore exhaustively sequences of declared methods up to a given bound for the sequence length. The driver specifies the methods that should be explored and their arguments. In its basic form, Symclat allows only primitive arguments. (See below how Symclat uses wrappers to handle non-primitive arguments.) These arguments are symbolic variables from which symbolic execution will build symbolic expressions and path conditions. Symclat uses a depth-first traversal to explore all paths within one method

pair	tool/input	generation	classification
RanGen+ OpModel	**Implementation:** Eclat. **Input to tool:** classes under test and an existing test suite. **Model:** operational model derived from test suite.	**(RanGen)** Generates test inputs by constructing method sequences and input data in a random fashion.	**(OpModel)** A test whose operational pattern of execution differs from the model is labeled illegal or fault-revealing. A test whose operational pattern does not differ from the model is labeled normal.
RanGen+ UCExp	**Implementation:** Eclat, modified to work without an operational model. **Input to tool:** classes under test. **Model:** exception-based model (uncaught exceptions are fault-revealing).	Same as (RanGen) above.	**(UCExp)** A test whose execution throws an uncaught exception is labeled fault-revealing.
SymGen+ OpModel	**Implementation:** Symclat. **Input to tool:** classes under test and an existing test suite. **Model:** operational model derived from test suite.	**(SymGen)** Generates symbolic test inputs by exploring exhaustively method sequences providing symbolic variables as parameters to methods. Symbolic tests are translated to concrete ones by solving constraints accumulated during the execution of the symbolic test.	Same as (OpModel) above, but the operational model is interpreted symbolically.
SymGen+ UCExp	**Implementation:** Symclat. **Input to tool:** classes under test. **Model:** exception-based model (uncaught exceptions are fault-revealing).	Same as (SymGen) above.	Same as (UCExp) above.

Figure 1. Instantiations of the general evaluation framework.

and a breadth-first traversal to explore method sequences. For more details, see our previous work on exploration [32] that Symclat builds on.

Model execution. Symclat uses Daikon [7] to generate models for the classes under test. Symclat instruments these classes to check model properties at method entry and exit points. (At entry, the check involves object invariants and method pre-conditions, and at exit, the check involves object invariants and method post-conditions.) The tool discards the properties that it cannot handle due to limitations of theorem provers (non-integer constraints, non-linear integer constraints etc.).

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. Once the variable get concrete values, Symclat outputs a test that consists of method sequences with concrete arguments.

Non-primitive arguments and wrappers. Symclat uses wrapper classes when non-primitive or non-integer arguments appear in the methods of the subjects under test. Suppose we want to test the method `equals(Object)` that the subject declares. The wrapper defines two fields of the subject type, one standing for the receiver and the other for the argument. It also declares operations to construct and mutate the receiver, and to create aliasing between the argument and the receiver. The wrapper looks like the following:

```
class SubjectWrapper {
  Subject receiver;
  Subject argument;
  ...
  public void equals() { receiver.equals(argument) ; }
  ...
  public void alias() { argument = receiver ; }
  public void cons() { receiver = new Subject() ; }
  public void mutate() { receiver...(); }
}
```

Drivers as well as the tests they generate operate on these wrapper classes instead of the actual subject classes. Sequences of method calls (*without* non-primitive arguments) on the wrapper can be translated into sequences of method calls (*with* non-primitive arguments) on the class 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. More precisely, they are unsupported due to the undecidability of the logic used to express constraints. When these operators appear in path conditions, Symclat backtracks the execution. When symbolic expressions appear in the conditional of a loop, Symclat "unfolds" the loop for only a limited number of times. This mechanism corresponds to setting a bound on the depth of the symbolic execution tree [17]. Thus, Symclat cannot catch stack overflow exceptions or detect infinite recursion. In addition, Symclat uses arbitrary precision integer arithmetic provided by the theorem prover and constraint-solver. As a result, Symclat cannot 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 to the branching tree making it possible to report infinite loops and stack overflows in the expense of decreasing precision. Integers could be encoded in the finite domain with bitvectors so to report on arithmetic overflows and be able to decide on expressions with % and /, and arrays could have symbolic representations in order to allow symbolic dereferences. But the underlying theoretical limits would prevent symbolic execution to completely handle all programs.

3. Experimental Study

Our study compares the techniques appearing in Figure 1 with and without reduction of test suites. We used Eclat and Symclat for implementing respectively RanGen and SymGen for test generation. Both Eclat and Symclat can use either UCExp or OpModel for classification. We first describe the experimental setup, then compare the techniques, and finally summarize the results of comparison.

3.1. Experimental Setup

We set a time bound of two minutes for running each test generation in Symclat. 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.

Figure 2 lists the subject programs that we 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. The subject are as follows:

- UBStack refers to the implementation of the unique bounded stack used in previous studies on testing [6, 18, 22, 33]. This code comes with two test suites, consisting of 8 and 12 test cases.
- ExpMDE is a supporting class from Daikon's codebase. The current Symclat implementation can explore only the two (overloaded) methods `create_combinations` as the others heavily use `String` objects, `float` numbers, or other constructs that the current Symclat implementation does not support.
- The subjects BinarySearchTree, StackAr, and StackLi are drawn from a textbook [29]. These classes are provided with a set of example uses.
- The next 9 subjects refer to a set of example classes provided in the JML distribution [15]. They are provided with formal specifications.
- RatPoly refers to 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.

Figure 2. Size of the subjects.

subject	NCNB LOC	#methods
UBStack (8)	88	11
UBStack (12)	88	11
ExpMDE	1832(37)	69(2)
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	582.51	17.20

RanGen can explore all methods from the subjects, while SymGen may not be able to execute all methods. For our subjects, RanGen and SymGen differ only in ExpMDE; we show the numbers of lines of code that SymGen explores in parentheses. For RatPoly, the numbers are averages over 46 different implementations. We selected our subjects from those used in a previous study on Eclat [18]. We selected all subjects that the current Symclat implementation can explore; the study on Eclat used 631 implementations, but approximately 90% of them contain programming constructs that the current Symclat implementation cannot explore (`float` numbers, `String` objects, symbolic indexing into arrays, etc.). Even for the selected subjects, the current Symclat implementation cannot explore all the methods.

We next describe how we label the tests generated by different techniques. The goal is to determine which tests reveal *actual* faults. We use formal JML specifications for each subject to detect tests that violate the specifications. We consider especially arithmetic (integer) overflows, as several tests for RatPoly implementations produce them. This potential type of fault is inherent in all RatPoly subjects as the problem set asked the students to use a staff-provided class for rational numbers based on fixed precision numbers (Java 32-bit `int` numbers). We instrumented this class to detect overflows during test execution. We keep overflows in separate and do not count them as actual faults.

We label each test as follows. A test that produces a JML violation *before* an overflow is labeled as a specification violation. Such tests reveal a mismatch between the code and the specification, and in our study all such tests but one were caused by actual faults in either the code or the specification.

We also determine for each test suite the number of distinct specification violations. The rationale is that a test suite with 10 test cases that are all faulty but all due to the same fault may not be as valuable as a test suite with 10 test cases, only two of which are faulty but due to two distinct faults. Each test that violates a JML specification either throws an uncaught exception at some program point (violating the implicit specification that the code should not throw uncaught exceptions) or violates some explicit part of the specification (class invariant, internal method pre-condition, or method post-condition) at some program point. We consider two tests to have 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 reduction as pointed out in Section 2 but using JML specifications.

3.2. Results with Reduction

Figure 4 shows the results of running the subjects in all four techniques with reduction of generated test suites. The columns list as TG the difference between the number of tests generated and the number of tests that result in arithmetic overflows, as JV the sum of tests that either violate JML specifications or raise uncaught exceptions, and as Pr. the precision given by the ratio JV/TG. The number of distinct violations appears in parentheses in the JV column.

For UBStack (8), each generation-classification pair finds the same two faults. The two pairs that use UCExp classification generate no false positives, whereas the two pairs that use OpModel classification generate one false positive. UBStack (12) differs from UBStack (8) in the tests from which the operational model is inferred. Thus, the results for these two subjects differ only for the two pairs that use OpModel classification. Each of those two pairs generates one test that detects an actual fault. These pairs miss the other fault because the operational model for UBStack (12) includes a pre-condition that the argument of `equals` should not be `null`; thus, the test that sets the parameter to `null` is classified as illegal, although it is legal and fault-revealing.

For ExpMDE, the techniques based on SymGen generate only false positives: all tests call `create_combinations` with a parameter set to `null` that directly leads to a `null` dereference exception. However, these tests are actually illegal as they violate the actual method pre-condition, but OpModel does not infer this pre-condition. RanGen detects a real fault in `create_combinations` as it ends up in an infinite recursion when two of the parameters are 0. RanGen detects this scenario because the test execution results in a stack overflow. If the method had an infinite loop, RanGen would not detect it but instead it would report that the test times out. SymGen did not detect the infinite recursion as it requires a very long execution sequence. RanGen also detects an index out of bounds exception when given a negative value to one of the arguments. SymGen misses this fault because its exploration time expires before reaching the fault, i.e., the symbolic search gave priority to branches that did not reach the fault.

For 10 out of 12 subjects from the textbook [29] and JML samples [15] none of the techniques detect any actual fault. To the best of our knowledge, these subjects do not have any fault that any of the four combinations could detect.

For `DLList` and `SLList`, the techniques based on SymGen generates only false positives due to weak preconditions used in the exploration. The implementation of `toString` in these data structures do not admit lists with self references. Such inputs lead to infinite recursion causing stack overflow error. The formal specification did not expose this constraint as a pre-condition of `toString`. `DLList` is a subclass of `SLList` and overrides the method `toString` but the fault was revealed in both (recursive) implementations and with tests that create self references.

For RatPoly, there are a total of 46 subjects. For 37 subjects, none of the four pairs with reduction detects any fault. For the remaining 9 subjects, these pairs find a total of 9 distinct actual faults.

In s5, s13, and s24 SymGen+UCExp detects a fault in the method `div` of the rational polynomial class. This fault results in an array index out of bounds exception. Model-based symbolic testing detects the fault only in s24 because in s5 and s13 one of the methods in the test that could detect a fault has a too strong pre-condition inferred by OpModel and thus appears to be illegal. RanGen did not detect this fault as it does not select the appropriate values for the method parameters in the tests.

All techniques reported a fault in s7 and s33 when calling the operation `antiDifferentiate` passing the argument 0. These implementations add a new rational term using the argument as the coefficient and the constant 0 as the exponent, and the specification includes an invariant stating that no polynomial should include zero-coeffcient terms. In s21, the techniques based on UCExp reported the same

Figure 3. Comparison of generation-classification pairs with reduction of test suites.

subject	RanGen+OpModel			RanGen+UCExp			SymGen+OpModel			SymGen+UCExp		
	#TG	#JV	Pr.	#TG	#JV	Pr.	#TG	#JV	Pr.	#TG	#JV	Pr.
UBStack (8)	3	2(2)	0.67	2	2(2)	1.00	3	2(2)	0.67	2	2(2)	1.00
UBStack (12)	1	1(1)	1.00	2	2(2)	1.00	1	1(1)	1.00	2	2(2)	1.00
ExpMDE	2	1(1)	0.50	3	2(2)	0.67	1	0(0)	0.00	2	0(0)	0.00
DLList	0	0(0)	NaN	5	1(1)	0.20	4	0(0)	0.00	4	0(0)	0.00
SLList	3	1(1)	0.33	7	1(1)	0.14	6	0(0)	0.00	6	0(0)	0.00
s5	0	0(0)	NaN	1	0(0)	0.00	1	0(0)	0.00	7	1(1)	0.14
s7	20	0(0)	0.00	17	2(1)	0.12	4	1(1)	0.25	10	1(1)	0.10
s13	1	0(0)	0.00	1	0(0)	0.00	2	0(0)	0.00	7	1(1)	0.14
s14	2	0(0)	0.00	3	2(1)	0.67	3	0(0)	0.00	7	0(0)	0.00
s21	0	0(0)	NaN	14	4(1)	0.29	0	0(0)	NaN	12	1(1)	0.08
s24	1	1(1)	1.00	1	0(0)	0.00	2	1(1)	0.50	7	1(1)	0.14
s33	2	2(1)	1.00	3	2(1)	0.67	3	1(1)	0.33	8	1(1)	0.12
s39	1	1(1)	1.00	2	1(1)	0.50	1	0(0)	0.00	6	0(0)	0.00
s46	0	0(0)	NaN	1	0(0)	0.00	2	1(1)	0.50	7	1(1)	0.14
total	93	9(8)	0.24	151	19(13)	0.10	98	7(7)	0.06	335	11(11)	0.05
for 4 subjects	0	0(0)	NaN	0	0(0)	NaN	0	0(0)	NaN	0	0(0)	NaN
for 43 subjects	57	0(0)	0	89	0(0)	0	65	0(0)	0	248	0(0)	0

Figure 4. Comparison of generation-classification pairs without reduction of test suites.

subject	RanGen+OpModel			RanGen+UCExp			SymGen+OpModel			SymGen+UCExp		
	#TG	#JV	Pr.	#TG	#JV	Pr.	#TG	#JV	Pr.	#TG	#JV	Pr.
UBStack (8)	29	21(2)	0.72	15	15(2)	1.00	100	63(2)	0.63	91	90(2)	0.99
UBStack (12)	12	12(1)	1.00	15	15(2)	1.00	26	25(1)	0.96	91	90(2)	0.99
ExpMDE	11	10(1)	0.91	27	26(2)	0.96	2	0(0)	0.00	17	0(0)	0.00
DLList	318	21(2)	0.07	320	21(2)	0.07	75	0(0)	0.00	75	0(0)	0.00
SLList	256	21(1)	0.08	684	27(1)	0.04	1535	0(0)	0.00	1535	0(0)	0.00
s5	0	0(0)	NaN	24	0(0)	0.00	33	0(0)	0.00	112	1(1)	0.01
s7	320	19(1)	0.06	301	33(1)	0.11	152	49(1)	0.32	505	162(1)	0.32
s13	1	0(0)	0.00	24	0(0)	0.00	16	0(0)	0.00	83	1(1)	0.01
s14	11	2(1)	0.18	31	3(1)	0.10	34	8(2)	0.24	136	33(1)	0.24
s21	0	0(0)	NaN	65	23(1)	0.35	0	0(0)	NaN	309	67(1)	0.22
s24	0	0(0)	NaN	24	0(0)	0.00	13	1(1)	0.08	52	1(1)	0.02
s33	29	28(1)	0.97	52	28(1)	0.54	29	7(1)	0.24	144	24(1)	0.17
s37	0	0(0)	NaN	24	0(0)	0.00	24	0(0)	0.00	149	1(1)	0.01
s39	10	10(1)	1.00	37	13(1)	0.35	19	0(0)	0.00	51	0(0)	0.00
s46	0	0(0)	NaN	24	0(0)	0.00	38	5(1)	0.13	129	2(1)	0.02
total	1925	144(11)	0.20	3649	204(15)	0.08	5204	158(9)	0.05	10136	472(13)	0.05
for 4 subjects	0	0(0)	NaN	0	0(0)	NaN	0	0(0)	NaN	0	0(0)	NaN
for 42 subjects	928	0(0)	0	1982	0(0)	0	3108	0(0)	0	6657	0(0)	0

fault. The techniques based on OpModel could not detect this fault for this student due to strong pre-conditions inferred to one internal operation to `antiDifferentiate`.

In s14, RanGen+UCExp reported a fault when trying to evaluate a polynomial that represents NaN (Not a Number).

In s39, RanGen detects a fault due to parsing and unparsing of strings that represent rational polynomials. Symclat drivers did not explore these methods as they involve string objects.

In s46, SymGen could find an index out of bounds exception raised from the `div` operation when trying to access the 0-index term in the remainder polynomial which can be 0. RanGen misses this fault as it does not pick the appropriate input value to exercise the method.

3.3. Results without Reduction

Figure 4 shows the results without reduction of generated test suites obtained by running the tools that implement the techniques in Figure 1. Without reduction, the generated test suites can detect more faults, but often result in lower precision.

In s37, SymGen+UCExp detects a fault produced when the `NaN` polynomial is passed to `add`. RanGen misses this fault as it does not systematically produce `NaN` polynomials and provides them as inputs to methods.

In s14, model-based symbolic testing without reduction detects a fault that it does not detect with reduction. Without reduction, model-based symbolic testing detects this fault in an indirect way. The operational model of a method infers a post-condition that is stronger than the real post-condition. As a result, OpModel labels as faulty some actually normal tests and some actually fault-revealing tests then violate the inferred post-condition. The reducer chooses as a representative test for these violations a test that is actually a normal test (since the reducer does not know the actual specification) and thus misses the fault.

3.4. Summary

Our study evaluates how effective is RanGen as opposed to SymGen in finding faults, and how effective is OpModel as opposed to UCExp in finding faults. Based on the experimental results, we give the following answers.

RanGen and SymGen are complementary techniques for test generation. When SymGen can explore some code (i.e., the code has no programming constructs that SymGen cannot explore), it can generate tests that reveal faults, and it can reveal all the faults than RanGen can. However, SymGen often cannot explore code; we were able to run our Symclat tool 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 some more subjects, but in general, SymGen cannot explore arbitrary code because of 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, RanGen itself can miss some faults in code that it explores. For instance, it can miss a fault if it does not generate a fault-revealing method sequence or it does generate a potentially fault-revealing sequence but does not generate appropriate values. We have found that case for our experiments: RanGen missed faults because it did not generate the appropriate values.

Furthermore, UCExp and OpModel are complementary techniques for test classification. In our experiments, UCExp revealed more faults, but each technique revealed some faults that the other missed. The use of non-trivial models in OpModel sometimes makes it better and sometimes worse than UCExp, e.g., OpModel can label as illegal a test that violates an incorrectly inferred pre-condition although the test is legal and throws an exception, but OpModel can also label as potentially fault-revealing a test that violates a post-condition although the test does not thrown an exception.

In summary, we give several suggestions for improving the existing techniques and tools for test generation and classification and for using such tools:

- 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, this means that 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 also 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 for that sequence. This leads to selecting values that are equal in a sequence. The tools should allow the users to provide heuristics for selection [5].
- In addition to biasing selection of values that are inputs to method calls, RanGen may benefit from biasing the selection of method calls. For example, an error that RanGen was unable to find requires using two equivalent polynomials as parameters to a method call. Equivalent polynomials can be obtained by repeating a sequence of method calls that creates one polynomial—such a sequence is unlikely to be produced with random generation. Repeated patterns of method calls can detect faults that would be highly unlikely to be found via random generation; for example, data structure implementations such as array-based lists and hash tables require several insertions before code is reached that resizes the containers. Random generation is unlikely to produce repeated additions necessary to reach this code.
- 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, 12, 21]. But there can be 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.
- Users should use both UCExp and OpModel classifications with the tools. These two classifications complement each other, and it is easy to use them both in any tool that provides OpModel. (Recall that UCExp is just

a special case with all models being trivial.) Additionally, it is helpful when the users can provide as complete specifications as possible, because they enable more precise classification.

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 ExpMDE are relatively small. These threats could be reduced by more experiments on wider types of subjects and tools in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in the tools that implemented four techniques might cause such effects. To reduce these threats, we manually inspected the results of a dozen of subject programs. One threat to construct validity is that our experiments use the violations of manually written JML specifications as symptoms of fault exposure. These manually written JML specifications may not be strong enough to catch some faults that are in fact exposed by tests exported by a certain technique.

4. Related Work

Duran and Ntafos [8] and Hamlet and Taylor [13] empirically compared random testing and partition testing such as path testing. They observed that random testing could be a potentially cost-effective alternative technique 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 [30] analytically showed that partition testing is more effective than random testing in fault detection when at least one subdomain of the partition has a high concentration of fault-inducing inputs.

To the best of our knowledge, there was no empirical comparison of test generation and classification tools in previous research. But there existed comparative studies on static bug-finding tools. For example, Rutar et al. [19] compared five Java static bug-finding tools against five open source projects. Their experimental results show that none of these five tools strictly subsumes another in terms of bug-finding capability and these tools often find non-overlapping bugs. They then proposed a meta-tool for combining and correlating these five tools' outputs.

Symclat developed in this research extends the research on using generated operational models to guide test-input generation and classification. Previous techniques implemented in Jov [33] and Eclat [18] exploit the guidance of generated operational models in test-input generation and classification. A commercial tool called Agitar Agitator [1] also infers operational models from test executions but it suggests these models to developers so that the developers can selectively promote them to assertions rather than aggressively classifying test inputs against the inferred models. In contrast to these previous tools, Symclat uses symbolic execution to systematically explore paths in both the code for checking operational models and the code under test.

5. Conclusions

This paper presents an empirical study comparing different techniques for test generation and classification. Specifically, we considered random and symbolic exploration as means to generate tests, and operational models and unchecked exceptions as means to classify tests. We developed a tool, Symclat, that implements SymGen+OpModel . The results show the following findings. (i) RanGen and SymGen detect different faults. When SymGen can explore some code under test, it can detect faults that RanGen misses. However, SymGen cannot explore certain programming constructs (partly because Symclat does not support them but more importantly because of the underlying theoretical limits of tools used in symbolic execution, such as theorem provers and constraint solvers), and RanGen can generate tests that find faults for code with such constructs. (ii) UCExp and OpModel detect different faults. OpModel's models find some faults that do not throw an exception but do violate a post-condition. However, OpModel misses some faults when a too strong inferred pre-condition filters out as illegal a test that is actually fault revealing.

References

[1] Agitar Agitatior 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 International Conference on Computer Aided Verifi cation*, 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 European Conference Object-Oriented Programming*, 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 International Conference on Functional Programming*, 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] Publications using the Daikon invariant detector tool, 2004. `http://www.pag.csail.mit.edu/daikon/pubs-using/`.

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

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

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

[11] P. Godefroid. Model checking for programming languages using Verisoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[12] 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.

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

[14] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. 25th International Conference on Software Engineering*, pages 60–71, 2003.

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

[16] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.

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

[18] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 504–527, Glasgow, Scotland, July 2005.

[19] N. Rutar, C. B. Almazan, and J. S. Foster. A Comparison of Bug Finding Tools for Java. In *Proc. 15th IEEE International Symposium on Software Reliability Engineering*, 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, Washington, October 2005.

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

[26] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 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 International Conference on Automated Software Engineering*, pages 414–417, November 2005.

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

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

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

[31] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[32] 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, April 2005.

[33] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.