Eclat: Automatic Generation and Classification of Test Inputs

Carlos Pacheco and Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab,
The Stata Center, 32 Vassar Street,
Cambridge, MA 02139 USA
{cpacheco, mernst}@csail.mit.edu

Abstract. This paper describes a technique that selects, from a large set of test inputs, a small subset likely to reveal faults in the software under test. The technique takes a program or software component, plus a set of correct executions—say, from observations of the software running properly, or from an existing test suite that a user wishes to enhance. The technique first infers an operational model of the software's operation. Then, inputs whose operational pattern of execution differs from the model in specific ways are suggestive of faults. These inputs are further reduced by selecting only one input per operational pattern. The result is a small portion of the original inputs, deemed by the technique as most likely to reveal faults. Thus, the technique can also be seen as an error-detection technique.

The paper describes two additional techniques that complement test input selection. One is a technique for automatically producing an oracle (a set of assertions) for a test input from the operational model, thus transforming the test input into a test case. The other is a classification-guided test input generation technique that also makes use of operational models and patterns. When generating inputs, it filters out code sequences that are unlikely to contribute to legal inputs, improving the efficiency of its search for fault-revealing inputs.

We have implemented these techniques in the Eclat tool, which generates unit tests for Java classes. Eclat's input is a set of classes to test and an example program execution—say, a passing test suite. Eclat's output is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails. In our experiments, Eclat successfully generated inputs that exposed fault-revealing behavior; we have used Eclat to reveal real errors in programs. The inputs it selects as fault-revealing are an order of magnitude as likely to reveal a fault as all generated inputs.

1 Introduction

Much of the skill in testing a software artifact lies in carefully constructing a small set of test cases that reveals as many errors as possible. A test case has two components: an *input* to the program or module, and an *oracle*, a procedure that determines whether the program behaves as expected on the input. Many techniques can automatically generate candidate inputs for a program [10, 18, 17, 23, 8, 4, 19, 9, 12], but constructing an oracle for each input remains a largely manual task (unless a formal specification of the software exists, which is rare). Thus, a test engineer wishing to use automated input generation techniques is often faced with the task of inspecting each resulting candidate input, determining whether it is a useful addition to the test suite, and writing an oracle for the input or somehow verifying that the output is correct. Doing so for even a few dozen inputs—much less the thousands of inputs automated techniques can generate—can be very costly in manual effort.

This paper presents three techniques that help the tester with the difficult task of creating new test cases. The first technique is an input selection technique: it selects, from a large set of test inputs, a small subset likely to reveal faults in the software under test—inputs for which writing full-fledged test cases is worth the effort. The goal of the technique is to focus the tester's effort on inputs most likely to reveal faults. Thus, the technique can also be viewed as an error-detection technique, and we have used it to find real errors in practice.

The input selection technique works by comparing the program's behavior on a given input against an operational model of correct operation. The model is derived from an example program execution, which can be an initial test suite or a set of program runs. If the program violates the model when run on the input, the technique classifies the input as (1) likely to constitute an illegal input that the program is not required to handle, (2) likely to produce normal operation of the program (despite violating the model), or (3) likely to reveal a fault. A second component of the technique (called the reducer) discards redundant inputs—inputs that lead to similar program behavior.

The other two techniques complement the input selection technique, by converting its output (test inputs) into a test suite (consisting of full-fledged test cases), and by providing a source of candidate test inputs for it to operate on.

Converting a test input into a test case requires the addition of an oracle, which determines whether the test succeeds or fails. We use an oracle that checks the properties in the operational model. Since the model was derived from correct executions, those properties are suggestive of correct behavior. By construction, the selected inputs will fail on these oracles. Together, the input selection and oracle generation techniques produce a set of failing test cases. This is a great starting point for the tester, whose job is to inspect each input, determine if its execution is in fact faulty, and determine if the oracle captures the proper behavior of the input. The tester can accept, reject, or modify each test input and test oracle.

The third technique is a generation-guided test input generation technique that makes use of operational-model-based classification to construct legal inputs. The input selection technique requires a set of candidate inputs; this technique provides it, while avoiding the generation of many illegal inputs.

We have implemented these techniques in the Eclat tool, which generates unit tests for Java classes. Eclat's input is a set of classes to test and an example program execution (say, a passing test suite). Eclat's output is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails. Our experiments show that Eclat reveals real errors in programs, and the inputs it selects are an order of magnitude as likely to reveal a fault as all generated inputs. Eclat is publicly available at `http://pag.csail.mit.edu/eclat/`.

The rest of the paper is structured as follows. Section 2 introduces the techniques with an example use of Eclat, a tool that implements them. Section 3 describes the techniques in detail. Section 4 describes the Eclat tool. Section 5 details our experimental evaluation of the technique. Section 6 discusses related and future work, and Section 7 concludes.

2 Example: BoundedStack

We illustrate the test generation and selection technique by describing the operation of the Eclat tool, when applied to a bounded stack implementation used previously in the literature [22, 30, 9]. The bounded stack implementation (Figure 1) and testing code were written in Java by two students, an "author" and a "tester." The tester wrote a set of axioms on which the author based the implementation. The tester also wrote two small test suites by hand (one containing 8 tests, the other 12) using different methodologies [22]. The smaller test suite reveals no errors, and the larger suite reveals one error (the method `pop` incorrectly handles popping an empty stack).

Eclat takes two inputs: the class under test, and a set of correct uses, in the form of an executable program that exercises the class. In this example, the set of correct uses is the 8-test passing test suite.

```
public class BoundedStack {
  private int[] elems;
  private int numElems;
  private int max;

  public BoundedStack() { ... }
  public int getNumberOfElements() { ... }
  public int[] getArray() { ... }
  public int maxSize() { ... }
  public boolean isFull() { ... }
  public boolean isEmpty() { ... }
  public boolean isMember(int k) { ... }
  public void push(int k) { ... }
  public int top() { ... }

  public void pop() {
    numElems --;
  }

  public boolean equals(BoundedStack s) {
    if (s.maxSize() != max)
      return false;
    if (s.getNumberOfElements() != numElems)
      return false;
    int[] sElems = s.getArray();
    for (int j=0; j<numElems; j++)  {
      if (elems[j] != sElems[j])
        return false;
    }
    return true;
  }
}
```

Fig. 1. Class `BoundedStack` [22] (abbreviated). Methods `pop` and `equals` contain errors

Eclat Report

Input 1

```
BoundedStack var8 = new BoundedStack();
var8.push(2);
int var9 = var8.getNumberOfElements();
var8.push(var9);
```

The last method invocation violated this property:

On exit: $size(\texttt{var8.elems}[]) - 1 \neq \texttt{var8.elems}[\texttt{var8.max} - 1]$

During execution of the last method invocation, a postcondition was violated. Since no preconditions were violated, this suggests a fault.

Input 2

```
BoundedStack var8 = new BoundedStack();
var8.equals((BoundedStack)null);
```

The last method invocation signaled a `java.lang.NullPointerException`.

There were no violations, but a throwable was signaled. Since the throwable is considered severe, this suggests a fault.

Input 3

```
BoundedStack var8 = new BoundedStack();
var8.pop();
```

The last method invocation violated this property:

On exit: $\texttt{numElems} \geq 0$

During execution of the last method invocation, a postcondition was violated. Since no preconditions were violated, this suggests a fault.

Fig. 2. Eclat's XML output for `BoundedStack` (formatted for presentation). Inputs 2 and 3 expose errors in the code under test. Input 1 is a false report: it merely indicates a deficiency in the original test suite

Eclat's output is a set of 3 new inputs—uses of the stack—that are classified as fault-revealing by the tool because their behavior differs from the provided test suite. Eclat can produce output in text, XML, or a JUnit test suite. Figure 2 shows the output in XML form. Each input is accompanied by an explanation of why the input suggests a fault, including any violated properties. Each violated property was true during execution of the original test suite, but was violated by the new input.

Input 1 violates one property during the call of `var8.push(var9)`. The violated property says that the last element of array `elems` is never equal to its index. This input reveals no fault; Eclat has made a mistake. The input, however, does point out a stack state not covered by the original test suite, so it may be a good addition to the test suite.

Execution of Input 2 violates no properties, but the `equals` method throws an exception. Eclat classifies the input as fault-revealing. The `equals` method (Figure 1) incorrectly handles a `null` argument. This fault went undetected in all previous analyses of the class [22, 30, 9].

```
public void test_3_pop() throws Exception {

  ubs.BoundedStack var8 = new ubs.BoundedStack();

  // Check preconditions.
  checkPreconditions_pop(var8);
  checkObjectInvariants(var8);

  var8.pop();

  // Check postconditions.
  checkPostconditions_pop(var8);
  checkObjectInvariants(var8);

}

public static void checkPreconditions_pop(Object thiz) {

  // Check: elems[max-1] >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(this, "elems")[eclat.Helper.intField(this, "max")-1] >= 0);
}

public static void checkPostconditions_pop(Object thiz) {

  // Check: elems[max-1] >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(this, "elems")[eclat.Helper.intField(this, "max")-1] >= 0);
}

public static void checkObjectInvariants(Object thiz) {

  // Check: max == elems.length
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "max")
    == eclat.Helper.intArray(thiz, "elems").length);

  // Check: elems != null
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(thiz, "elems") != null);

  // Check: max == 2
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "max") == 2);

  // Check: numElems >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "numElems") >= 0);
}
```

Fig. 3. JUnit test created by Eclat corresponding to Input 3 of Figure 2. When this JUnit test is executed, the last assertion in `checkObjectInvariants` fails during the second call (at the end of `test_3_pop`). This test detects an error in `BoundedStack`'s handling of `pop` when applied to an empty stack. Fields like `this.elems` are accessed via reflection, through method calls like `eclat.Helper.intArray(this, "elems")`. This allows the JUnit test suite to access non-public members of the tested class

Input 3 is classified as fault-revealing because its execution violates the property `numElems` $\geq$ 0. The variable `numElems` becomes negative after a call of `pop` on an empty stack. Eclat has revealed another true error: the `pop` method always decrements the top-of-stack pointer, even on an empty stack. This is a subtle error, because it silently corrupts the stack's state, and a fault only arises on a subsequent access to the stack. In particular, Input 3 itself has no user-observable fault; Eclat detects the corrupted stack state before it leads to an observable fault. A more complicated input—for example, an input that attempts to push an element when the stack pointer is negative and leads to an out-of-bounds exception—would probably be harder to understand and less useful for debugging.

Figure 3 shows a portion of Eclat's JUnit output. The figure shows the JUnit test created for Input 3, and its associated helper methods. Each test in the JUnit test suite will fail upon execution, indicating the violated property.

In summary, Eclat creates 3 inputs that quickly lead a user to discover two errors, and provides a JUnit test suite that exhibits the faulty behavior. Behind the curtains, Eclat generates and analyzes 806 distinct inputs. Some are discarded because they violate no properties and throw no exceptions (and thus suggest no faults). Some are discarded because they violate properties but are determined to constitute illegal uses of the class instead of faults. Some are discarded because they violate properties but are considered a new but non-faulty use of the class. Finally, some inputs are discarded because they behave similarly to already-chosen inputs: 5 of the inputs expose the pop-on-empty-stack fault (for example, one input pushes two items and then pops three times) but only one is selected.

3 Selection and Generation via Classification

This section describes the technique for selecting test inputs likely to reveal faults (Sections 3.1–3.3), the use of an operational model to create test cases from test inputs (Section 3.4), and the technique for generating candidate inputs (Section 3.5). We describe the techniques in the context of unit testing in an object-oriented programming language. The techniques can also be applied to non-object-oriented programs and to components larger than methods and constructors (see Section 3.6).

Figure 4 shows the input selection technique. The technique requires three things: (1) the program under test, (2) a set of correct executions of the program (for instance, an existing passing test suite for the program that a user wishes to enhance), and (3) a source of candidate inputs (each candidate may be an illegal input, or cause the program to behave normally, or reveal a fault).

The selection technique has three steps.

– **Model generation.** Observe the program's behavior on the provided correct executions, and create an *operational model* of correct behavior (Section 3.1).
– **Classification.** Classify each candidate as (1) *illegal*, (2) *normal operation*, or (3) *fault-revealing*. Do this by executing each candidate and comparing the program's behavior against the operational model (Section 3.2).
– **Reduction.** Partition the *fault-revealing* candidates based on their *violation pattern*: the set of violated properties. Report one candidate from each partition (Section 3.3).

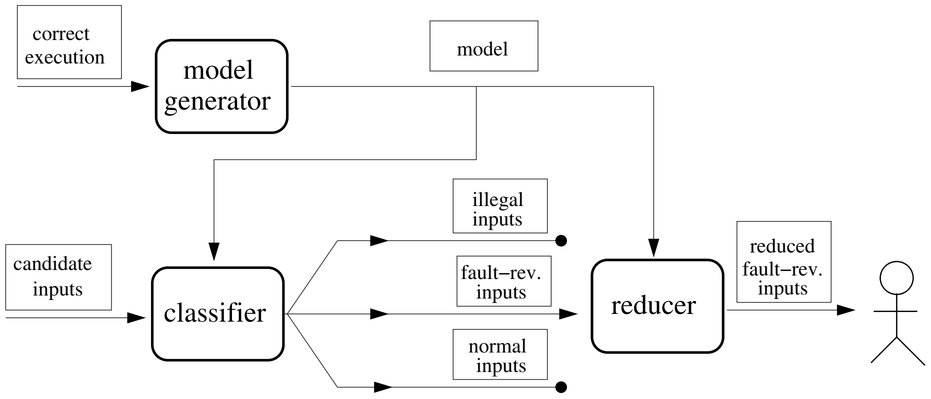

Fig. 4. The input selection technique. Implicit in the diagram is the program under test. Rectangles with rounded corners represent steps in the technique, and rectangles with square corners represent artifacts

Object invariants (hold on entry and exit of all public methods)

$\texttt{max} = \texttt{elems.length}$
$\texttt{elems} \neq \texttt{null}$
$\texttt{max} = 2$
$\texttt{numElems} \geq 0$

Properties that hold on entry to `pop`

$\texttt{elems}[\texttt{max} - 1] \geq 0$

Properties that hold on exit from `pop`

$\texttt{elems}[\texttt{max} - 1] \geq 0$

Properties that hold on entry to `push`

$\texttt{numElems} \in \{0, 1\}$

Properties that hold on exit from `push`

$\texttt{numElems} \in \{1, 2\}$
$size(\texttt{elems}[]) - 1 \neq \texttt{elems}[\texttt{max} - 1]$

Fig. 5. Part of an operational model for `BoundedStack` with respect to an 8-element test suite, generated by the Daikon [11] tool. An operational model reflects particulars of the test suite used to derive it; for example, the last property states that the last element in array `elems` is never equal to its index

3.1 Model Generation

The first step is to generate an operational model of the program. An operational model consists of properties that hold at the boundary of the program's components (e.g., on a public method's entry and exit). Our techniques impose no constraints on the program behavior captured by a model, but they require that every property can be evaluated at runtime.

The Eclat implementation uses operational abstractions generated by the Daikon invariant detector [11]. There are other techniques for generating models of program behavior based on an example use of the program [14, 26, 1, 16]. The models that these techniques generate vary in the kinds of properties they express, from legal sequences of method calls [26] to algebraic specifications of method behavior [16].

Figure 5 shows a simple operational model for `BoundedStack`. In this model, properties are observations about the state of the stack at various program points.

3.2 The Classifier

The classifier takes a candidate input and labels it *illegal*, *normal operation*, or *fault-revealing*. The classifier takes three arguments: a candidate input, the program under test, and an operational model. The classifier runs the program on the candidate input and records which model properties are violated during execution.

A violation means that the candidate input's behavior deviated from previous behavior of the program. Since the previously-seen behavior may be incomplete, such a violation does not necessarily imply faulty behavior. Depending on its violation pattern (the set of violated properties), the classifier labels a candidate input as *illegal*, *normal operation*, or *fault-revealing*. Figure 6 shows the decision table.

Executing an input can result in two kinds of violations: entry or exit violations. Entry violations suggest illegal program inputs, and exit violations suggest improper program behavior. The four possible categories of entry/exit violations are:

– **No entry or exit violations.** This category means that according to the operational model, the program received legal inputs and behaved properly. The technique labels the input *normal operation*.
– **No entry violations, some exit violations.** According to the model, a legal program input led to improper program behavior. The technique labels the input *fault-revealing*.
– **Some entry violations, no exit violations.** The program behaved properly on an illegal input. Since the program behaved properly, the technique labels the input *normal operation*. The program's satisfaction of the exit properties means that it is normal behavior; violation of the entry properties man that it is new behavior not seen in the example correct execution from which the model was generated.
– **Some entry and some exit violations.** The program behaved improperly on an illegal input. The technique labels the input *illegal*.

3.3 The Reducer

Section 3.2 described how an input's violation pattern leads to its classification. Violation patterns also induce a partition on all inputs, with two inputs belonging to the same partition if they violate the same properties. Inputs exhibiting the same pattern of violations are likely to be manifestations of the same faulty program behavior. Consider

Entry violations?	Exit violations?	Classification
no	no	***normal operation***
no	yes	***fault-revealing***
yes	no	**(new)** ***normal operation***
yes	yes	***illegal***

Fig. 6. Decision table for classifying a candidate input, based on the model violations that result from its execution

```
BoundedStack var0 = new BoundedStack();
var0.pop();
```

```
BoundedStack var0 = new BoundedStack();
var0.push(3);
var0.pop();
int var1 = var0.top();
var0.pop();
```

Fig. 7. Two Eclat-generated inputs that reveal the same error in the `pop` method. Both inputs violate the single property `numElems` ≥ 0 on exit from the last `pop`

Figure 7, which contains two fault-revealing inputs. Both inputs violate the same set of properties—namely, the single property `numElems` $\geq$ 0—and they uncover the same error in method `pop`. Presenting only one input will save the user the time to inspect a redundant input.

3.4 Oracle Generation: From Test Input to Test Case

A test engineer's goal is to find errors and to write tests that may find errors in the future. A test consists of an input and an oracle, so providing test inputs, even ones that are likely to be fault-revealing, leaves the test engineer responsible for determining both how the program ought to behave on the input, and how to verify that behavior. This section describes a technique that automatically converts a test input into a test case by proposing an oracle. The human remains the final arbiter of the test suite and should check and/or modify each test case, but the effort can be greatly eased by providing complete test cases rather than partial ones.

The oracle generation technique uses the model described in Section 3.1. Since the properties can be evaluated at run time, they can be converted into assertions and used as test oracles. These oracles check for deviation from previously-observed behavior. In addition to checking behavior, the properties serve as a human-readable explanation of what is being checked, which is important in a test case. Figure 3 shows an example of a test case output by our implementation.

3.5 Classifier-Guided Input Generation

We have presented a technique that selects from a set of candidate inputs a subset likely to reveal faults, and a technique that converts an input into a test case. This section describes a similar methodology to avoid generating illegal inputs in a bottom-up input generation strategy. First we present an unguided strategy for generating inputs, and then we present an enhancement to the strategy that makes use of the classifier from Section 3.2.

We describe input generation in the context of inputs like those in Figure 7, where an input is a sequence of method calls. The last method call is the tested call, with all previous method calls setting up state for the tested call. For example, the second input in Figure 7 has five method calls; the first four are setup, and the fourth one tests the method `pop` via the method call `var0.pop()`.

Unguided Bottom-up Generation. The unguided bottom-up generation strategy maintains a growing pool of values used to construct new inputs. Every value in the pool is accompanied by a code snippet (usually a sequence of method calls) that can be run to construct the value. Each code snippet can be viewed as a test input.

New values are created by combining existing values through method calls. For example, given stack value s and integer value i, the method call s`.isMember(`i`)` creates a new boolean value. Methods that return `void` are treated as producing a new value for the receiver. For example, method call s`.push(`i`)` creates a new stack value.

Bottom-up input generation proceeds in rounds. The pool is initialized with a set of initial values (for example, in Java, a few primitive values and `null`). In each round, new values are created by calling methods and constructors with values from the pool. Each new value is added to the pool and its code is emitted as a test input. The process is repeated any number of times.

Combining Generation and Classification. The unguided generation strategy is likely to produce both interesting inputs and a large number of illegal inputs, since there are no constraints on the arguments passed to method calls. The guided generation technique takes advantage of the classifier to guide the generation process.

As before, input generation proceeds in rounds. For each round:

1. Construct a new set of candidate values (and corresponding inputs) from the existing pool.
2. Classify the new candidate inputs with the classifier.
3. Discard inputs labeled *illegal*, add the values represented by the candidates labeled *normal operation* to the pool, and emit inputs labeled *fault-revealing* (but do not add them to the pool).

Figure 8 illustrates the process (it also adds the oracle generation technique discussed in Section 3.4, to give a complete view of the multiple techniques in a single framework). In the classifier-guided technique, a set of candidate inputs is no longer a required input—it has been replaced by an input generator that uses the classifier to avoid creating illegal inputs.

This enhancement removes *illegal* and *fault-revealing* inputs from the pool as soon as they are discovered, preventing these inputs from being used as building blocks to new method calls (any input that makes such a call would also be classified *illegal*, and is therefore useless to construct).

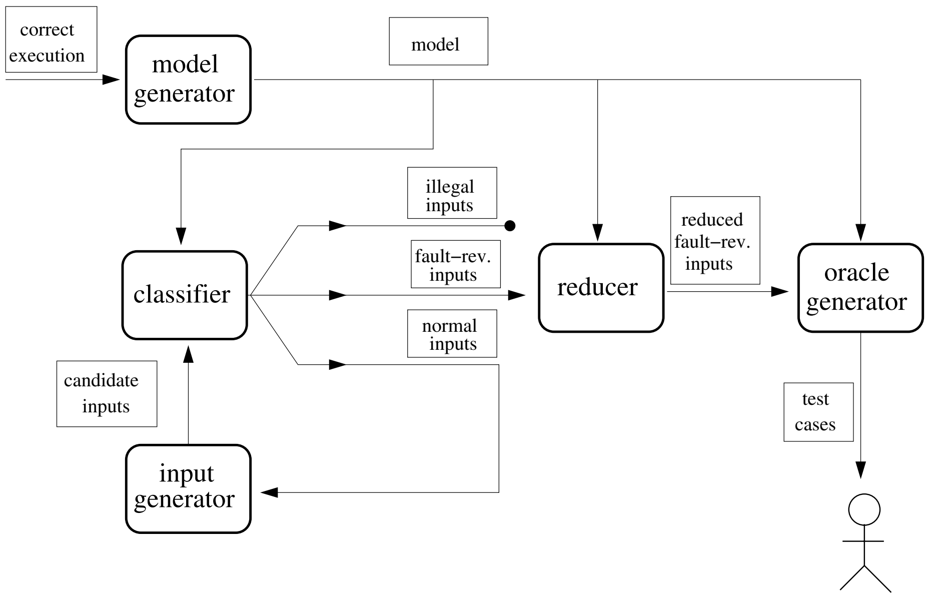

Fig. 8. The input selection technique of Figure 4, augmented with an input generator that uses the classifier to avoid creating illegal inputs, and with an oracle generator that produces test cases from test inputs. This diagram shows all the paper's techniques in a single integrated framework

3.6 Discussion

Applicability. We have presented our test selection technique in the context of an object-oriented programming language. The technique is also applicable in other programming contexts, as long as an operational model can be obtained, the model can be evaluated in the context of new program executions, and the model can be partitioned into entry and exit properties (preconditions and postconditions).

The technique reveals faults that are violations of the model properties. Eclat uses the Daikon invariant detector to infer a model. Daikon infers many kinds of properties about data structures, including heap-based ones, but does not infer, for instance, temporal properties of a program. Thus, one would not expect Eclat to be particularly good at finding faults that have to do with temporal properties.

Integration with manually-written specifications. Our research addresses a testing situation in which the tester has no access to a formal specification, but has a set of correct program executions from which an operational model can be derived. Increasingly, programmers write partial specifications to capture important properties of their software; safety-critical systems, for instance, sometimes contain at least a partial specification of the critical parts of the system. These specifications can be used to generate and classify test inputs. Partial specifications can erroneously classify inputs; for example, an illegal input may be labeled legal because the partially-specified precondition is not strong enough. Our classification technique permits use of manually-written or

Number of rounds	4
Goal number of new invocations per method per round	100
Failed tries after which generation attempts stop for a given method	100
Time limit (generation stops after limit is exceeded)	no limit

Fig. 9. Eclat's default parameters for generating test inputs

mechanically-derived properties, or both. The operational model can be complemented with manually-written specifications that capture important properties not mechanically derived. Conversely, partial specifications can be complemented with inferred properties to improve the input generation and classification process.

4 Implementation: Eclat

We have implemented our input generation, input selection, and oracle generation techniques in Eclat, a tool that automatically creates unit tests for Java classes. Eclat can produce output in text, XML, or a JUnit test suite. Eclat can be used through a command-line interface or as an Eclipse plugin. Eclat is publicly available at `http://pag.csail.mit.edu/eclat/`.

Eclat takes as input a set of classes to test and a program or test suite P that uses the classes. Eclat performs the following steps.

Deriving an operational model. Eclat uses the Daikon dynamic invariant detector [11] to derive a model of the classes' behavior on P; an example of Daikon's output appeared in Figure 5.

Compiling for runtime property checking. We have implemented a run-time-check instrumenter (distributed as part of Daikon at `http://pag.csail.mit.edu/daikon/`). The instrumenter takes the source files of the tested classes and the operational model derived by Daikon. It transforms the sources to check model properties during execution. Instrumentation is transparent: a violation does not alter the behavior of the class. Violated properties are recorded in a log.

Generating candidate inputs. Eclat generates candidate inputs using the classifier-guided, bottom-up generation strategy outlined in Section 3.5. Each round, new inputs are created by calling methods of the tested classes, selecting parameters at random from the pool. For each round, Eclat attempts to create a fixed number of new inputs for a given method using existing values from the pool. After a fixed number of failed attempts, it moves on to the next method. Figure 9 gives Eclat's default parameters. Section 5.6 evaluates Eclat's behavior when varying these parameters.

5 Evaluation

We have run a series of experiments to quantify the effectiveness of our test input generation and selection techniques. Section 5.1 introduces the programs and experimental methodology. Section 5.2 evaluates how well Eclat's selected inputs reveal faults.

Program	versions	suites per version	independent components	classes per component	public methods	NCNB LOC
BoundedStack	1	2	1	1	11	88
DSAA	1	1	9	1.5	110	640
JMLSamples	1	1	25	1.9	221	1392
utilMDE	1	2	1	1	69	1832
RatPoly	97	1	1	4	17	512
Directions	80	2	1	6	42	342

Fig. 10. Subject programs. For programs with multiple versions, numbers are average per version. NCNB LOC means non-comment, non-blank lines of code. These numbers do not include testing code

Section 5.3 measures Eclat's effectiveness when supplied small initial test suites. Sections 5.4–5.6 evaluate the classifier, the reducer, and the classifier-guided input generator individually.

5.1 Subject Programs and Methodology

Figure 10 lists our subject programs. The programs encompass 64 distinct interfaces, and a total of 631 implementations of those interfaces in 75,000 non-comment non-blank lines of code. All subject programs implement modestly-sized libraries designed to support larger programs; thus, unit testing is appropriate for them. All errors are real errors inadvertently introduced by the author(s) of the program.

- BoundedStack is the stack implementation discussed in Section 2. We report separately the results of running Eclat with the 8-test suite, and with the 12-test suite (with the one fault-revealing test removed).
- DSAA is a collection of data structures from an introductory textbook [25]. The author of the classes wrote a small set of example uses of the class: they are not exhaustive tests.
- JMLSamples is a collection of 25 classes that illustrate the use of the JML specification language. It is part of the JML distribution (`www.jmlspecs.org`). The test suites and specifications were written by the authors of the classes.
- utilMDE is a utility package that augments the `java.util` package. We report two results: one running Eclat with the test suite written by the authors of utilMDE, and the other via the unit tests of an unrelated program (Daikon [11]) that uses part of the utilMDE package.
- RatPoly is a set of student solutions to an assignment in MIT class 6.170, Laboratory in Software Engineering. The RatPoly library implements the core of a graphing calculator for polynomials over rational numbers. The course staff provided a test suite to the students as part of the assignment.
- Directions is a different set of student solutions in MIT class 6.170, written by the same students who wrote the RatPoly solutions. The Directions library is used by a MapQuest-like program that outputs directions for traveling from one location to another along Boston-area streets. For this assignment, students wrote their own

test suites. We report separately the results of running Eclat with the student-written suite, and with the suite used by the staff to grade the assignment, which was not provided to the students.

Eclat assumes a correct set of executions. Before running Eclat on BoundedStack and its 12-test suite, which contains one failing test, we removed the failing test.

For RatPoly, we discarded submissions that did not pass the staff test suite, which was provided as part of the assignment. For both RatPoly and Directions, we also discarded submissions for which Eclat generated more than 10 times the average number of fault-revealing inputs. These were solutions so faulty that finding fault-revealing inputs was not challenging, making input selection techniques unnecessary. The numbers in Figure 10 count only versions we kept.

Measurements. We organized our subject programs into nine experiments, each corresponding to using Eclat with a particular subject program and test suite. For a given experiment, we ran Eclat separately on each independent component (for example, we ran Eclat separately on DSAA's nine components: a binary tree, a disjoint set, a treap, an array-backed stack, a list-backed stack, a queue, a red-black tree, a linked list, and a binary heap). Thus, each experiment consisted of potentially many runs of Eclat: one per $\langle$ component, version $\rangle$ pair. For each experiment, we report results that are the average over all runs.

When computing average results for all experiments, we give the same weight to each experiment, regardless of the number of versions or runs of Eclat that the program represents. We do this to avoid over-representing experiments with multiple versions or components.

We wrote formal specifications for all the subject programs (except for JMLSamples, which already had formal specifications written by its authors). We use the specifications to evaluate the classification technique, with the specification representing an ideal classifier. Of course, in the presence of a formal specification our classification technique is not necessary: the specification indicates whether an input is illegal, normal, or fault-revealing. Our techniques are intended for use when formal specifications are not available, as was the case for most of the programs.

Comparison with other tools. JCrasher [9], Jtest [19], and Jov [30] have the same goals as Eclat: to generate random candidate inputs and select potentially fault-revealing ones. We report results from running JCrasher. We tried the other tools, but Jov and Jtest were unusable in many instances (Jov sometimes exited abnormally, and Jtest sometimes failed to terminate).

5.2 Evaluating Eclat's Output

Figure 11 shows how many inputs per run Eclat generated, how many it selected, and how many of those revealed faults. The figure also shows JCrasher's results on the subject programs. The results for JCrasher are the same for experiments that use the same programs with different test suites because JCrasher does not make use of the test suite. We also executed all the inputs against the formal specifications (using `jmlc` [6]). We

	Generated inputs			Selected inputs			JCrasher inputs		
Program	inputs generated	reveal faults	precision	inputs selected	reveal faults	precision	inputs selected	reveal faults	precision
BoundedStack (8-test suite)	806	13	1.6%	3	2	67%	0	0	—
BoundedStack (12-test suite)	1411	22	1.6%	1	1	100%	0	0	—
DSAA	806	0	0%	1.3	0	0%	0.89	0	0%
JMLSamples	396	0.50	0.13%	0.72	0.061	8.4%	0.12	0	0%
utilMDE (test suite)	1787	92	5.1%	18	4	22%	1	0	0%
utilMDE (sample usage)	1774	63	3.6%	18	2	11%	1	0	0%
RatPoly	2862	29	1.0%	1.5	0.65	42%	4	0.13	3.3%
Directions (student suite)	1099	40	3.6%	1.3	0.081	6.4%	1.6	0.025	1.6%
Directions (staff suite)	1099	41	3.8%	0.45	0.079	18%	1.6	0.025	1.6%
average	1338	33	2.3%	5.0	1.1	30%	1.13	0.02	0.92%

Fig. 11. Summary of Eclat's results. The first three numeric columns represent inputs internally generated by Eclat. The next three columns represent inputs reported to the user (after selection and reduction). The last three columns represent inputs selected as fault-revealing by JCrasher. Precision is the percentage of inputs that are fault-revealing. We calculated the average precision by taking the average of the individual experiments; this gives each experiment equal weight, but is slightly different from dividing the average number of fault-revealing inputs by the average number of selected inputs

true label	inputs generated	inputs selected
normal	74%	31%
illegal	24%	38%
fault	2.3%	30%

Fig. 12. True labels of generated and selected inputs. The entries in each column sum to 100% (modulo rounding imprecision). These results represent a total of 440,000 inputs

considered an input fault-revealing if it satisfied all preconditions of the tested method, and the method invocation caused a postcondition violation.

On average, Eclat selected 5.0 inputs per run, and 30% of those revealed a fault. By comparison, JCrasher selected 1.13 inputs per run, and 0.92% of those revealed a fault.

The inputs that Eclat selects are an order of magnitude as likely to reveal faults as the original candidate inputs (30% vs. 2.3%). Figure 12 shows another view of the results: it gives the true label of the generated and selected inputs, i.e., the label assigned by the formal specification. Selection is effective at improving a set of inputs by increasing the ratio of fault-revealing to non-fault-revealing ones.

5.3 Effectiveness on Small Initial Test Suites

Classification depends on a set of correct program executions to derive an approximate model of correct program behavior. This section measures the effect of the initial test suite on Eclat's fault-finding effectiveness. To evaluate the technique's performance on smaller suites, we artificially reduced the set of correct executions used by Eclat to construct an operational model. We compared our previous results with running Eclat using only the first 10% of the original execution trace (which was itself sometimes quite small). The table below shows the results.

	inputs generated	reveal faults	inputs selected	reveal faults
original trace	1338	33	5.0	1.1
10% of trace	1219	29	5.6	1.2

When given a smaller trace, Eclat selected more inputs (5.6 for the small trace, 5.0 for the original trace). Of those, almost the same percentage were fault-revealing.

Generating inputs based on the full-sized trace yields only slightly better results—fewer inputs to inspect, and almost the same number of fault-revealing ones among them. The technique is still effective with an impoverished trace, which makes it useful in the presence of a small test suite that does not cover all aspects of the program's behavior.

The table below shows the percentage of methods covered per test suite, and average number of calls made to each covered method. The number of calls per method covered does not give the whole story, since the distribution is highly non-uniform: in each case (even when test suites exist), a few methods are called many times and most methods are called very few times.

Program	methods covered	calls per method covered
BoundedStack (8-test suite)	82%	8
BoundedStack (12-test suite)	100%	18
DSAA	90%	679
JMLSamples	84%	102
utilMDE (test suite)	46%	13747
utilMDE (sample usage)	1.5%	4
RatPoly	83%	501
Directions (student suite)	85%	330
Directions (staff suite)	85%	3015

For the programs with multiple test suites (BoundedStack, DSAA, and utilMDE), the difference in coverage and number of calls per method is large, but the difference in Eclat's results is smaller.

5.4 Evaluating the Classifier

Every input has two labels, one assigned by Eclat and the true label assigned by the formal specification. Figure 13 shows the proportion of inputs falling into each ⟨Eclat label, true label⟩ category

The last row in Figure 13 shows the *precision* [21, 24] of Eclat's classifier. Precision is the ratio of correct labelings to the total number of labelings:

$$\text{precision} = \frac{\text{inputs correctly labeled as } L}{\text{inputs labeled as } L}$$

true	Eclat label			
label	normal	illegal	fault	**recall**
normal	0.67	0.045	0.030	90%
illegal	0.057	0.17	0.012	24%
fault	0.013	0.0035	0.0058	59%
precision	90%	78%	12%	

Fig. 13. Each entry shows the average proportion of generated inputs with the given Eclat label and true label. The sum of the nine middle entries is 1. The sum of each row in the nine middle entries yields the percentages in the middle column of Figure 12

The last column in Figure 13 shows the *recall* [21, 24] of the classifier. Recall is the ratio of correct labelings to the total number of inputs that belong to the label:

$$\text{recall} = \frac{\text{inputs correctly labeled as } L}{\text{inputs that are actually } L}$$

In summary, the classifier:

- correctly labels the vast majority of inputs as non-fault-revealing (90% precision, 90% recall for normal inputs),
- recognizes most fault-revealing inputs (59% precision for fault-revealing inputs), but
- labels fault-revealing many inputs that are not (12% precision for fault-revealing inputs).

The degree to which the technique overclassifies normal inputs as illegal depends on the accuracy with which the operational model captures the legality of the program's inputs. An operational model that is out of sync with the true input space of the program can indicate a poor test suite. A good example of this is BoundedStack. This interface permits arbitrary sequences of method calls with arbitrary parameters, so it is impossible to produce an illegal input, but the technique classifies many inputs as such, due to the test suite's poor coverage. When a test engineer inspects an input that is incorrectly classified as fault-revealing, the engineer is likely to find weaknesses in the test suite, permitting the engineer to improve it.

Identifying new behavior. Our technique classifies inputs into one of three labels: *illegal*, *normal operation* and *fault-revealing*. As shown in Figure 6, there are two kinds of normal inputs: those that violate no model properties, and those that violate some preconditions but no postconditions. The latter, called *new* inputs, are inputs that diverge from the original test suite, but the properties they violate are not considered indicative of faults; instead they are considered indicative of an overconstrained model. We experimented with outputting the *new* inputs for user inspection along with the fault-revealing ones, but we found that new behaviors were no more effective in revealing faults than normal behaviors that violate no properties. However, distinguishing new behaviors from old ones might help the programmer improve a test suite's coverage by suggesting normal program operation not already covered by the suite.

5.5 Evaluating the Reducer

The reducer takes the inputs labeled *fault-revealing*, and retains a representative subset. The table below summarizes its behavior. The first numeric column shows the average distribution of all inputs that the classifier labeled *fault-revealing* (the input to the reducer). The next column shows the distribution of inputs selected (the output of the reducer). Each column sums to 100%, modulo rounding imprecision.

true label	inputs labeled as fault by classifier	inputs selected (reduced)
normal	63%	31%
illegal	25%	38%
fault	12%	30%

The reduction step increases the percentage of fault-revealing inputs from 12% to 30%. For these programs (and, we suspect, for programs in general), fault-revealing program behavior is more difficult to produce than illegal or normal behavior, and thus more difficult to produce repeatedly by different inputs. This makes fault-revealing inputs less reducible than other inputs, because there are fewer inputs per partition, resulting in an increased proportion of selected fault-revealing inputs.

5.6 Evaluating the Input Generator

Classifier-guided Input Generation. Section 3.5 describes the use of the classifier in a bottom-up input generation strategy in which only inputs classified as *normal operation* are added to the growing pool of inputs. The first line in Figure 14 shows the results of this strategy (Eclat's default) for the formally-specified programs (this line repeats the averages from Figure 11). The second line shows the result of running Eclat using unguided generation: all inputs from previous rounds are added to the pool regardless of their classification.

Unguided generation leads to a larger number of inputs generated. The reason is that the pool has a larger number of building blocks to create new inputs from. Despite the larger number of inputs generated, fewer of those inputs are fault revealing. This is reflected in the results: with the unguided generation strategy, Eclat reports a larger number of inputs and yet fewer inputs are fault-revealing.

We can gain insight into this difference by looking back at Figure 12, which shows that the input selection technique selects not only more *fault-revealing* inputs, but also more *illegal* inputs. Eclat is most effective at correctly classifying normal inputs, but less so for illegal ones. When we remove the classifier from the generation process, the number of illegal inputs among candidate inputs increases, and Eclat selects more of them as fault-revealing, which decreases the tool's precision. Constraining the building blocks used by the generator to inputs classified as *normal operation* reduces these false positives.

Generation Parameters. This section evaluates Eclat's output under varying parameters. We varied two parameters:

	inputs generated	reveal faults	inputs selected	reveal faults
classifier-guided generation	1338	33	5.0	1.1
unguided bottom-up generation	3217	17	5.3	0.80

Fig. 14. Comparison of unguided and enhanced bottom-up generation. The first line summarizes the results for classifier-guided generation (averages reproduced from Figure 11). The second line uses unguided input generation

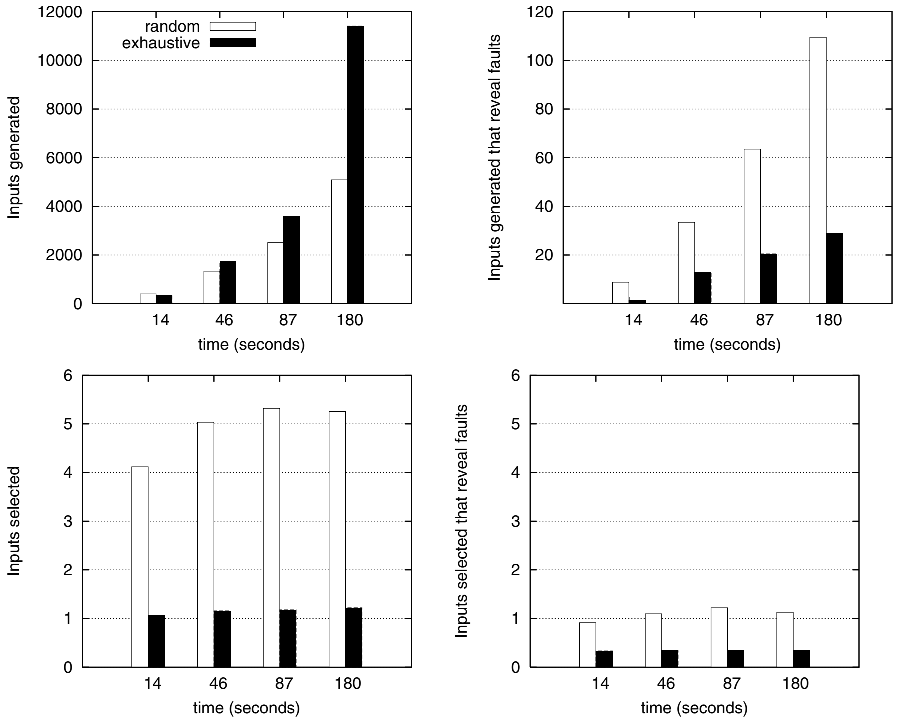

Fig. 15. Number of inputs generated and selected by Eclat, when varying the number of rounds and the generation strategy. The white bars are the results of running Eclat using random generation. The four data points are for the end-to-end time Eclat takes doing 2, 4, 6, and 8 rounds of random generation. The black bars are the results of running Eclat using exhaustive generation. The times shown are averages over all experiments

– The number of rounds of bottom-up generation. Eclat's default is 4 rounds; we also ran the experiments using 2, 6, and 8 rounds of generation.
– The number of new inputs generated per round. Eclat's default is to randomly generate 100 new inputs per method per round. To compare this approach against a more systematic approach, we added exhaustive generation to Eclat: for each round, it exhaustively generates all new inputs that are possible to generate given the current pool of values. To compare this approach against random generation, we mea-

sured how random and exhaustive generation performed given the same amount of time. We measured the time that Eclat spent generating, classifying and reducing inputs using random generation for a given number of rounds, and we ran Eclat again, using exhaustive generation and setting a time limit equal to the time spent by random generation.

Figure 15 shows the results for the eight possible combinations of parameter variations described above. Given the same amount of time, random generation generates fewer candidate inputs (upper-left plot). At every attempt to generate a new input for a method, Eclat's random generation algorithm randomly chooses a set of parameters, and then checks to see if the input has already been generated. This adds two costs to random generation: the cost of comparing a newly-generated random input for membership in the set of existing inputs, and the wasted cost of generating an input that is already in the pool. Exhaustive generation, on the other hand, never re-generates an already-existing input.

Despite creating fewer candidate inputs, random generation produces better-quality candidates—candidates that are fault-revealing (upper-right plot). Exhaustive generation creates many inputs that exercise the class in ways that are indistinguishable for the purpose of fault detection. Random generation produces a more diverse collection of inputs and more fault-revealing inputs than exhaustive generation (bottom plots). In future work, we plan to investigate exhaustive generation combined with techniques for avoiding generation of duplicate inputs [28, 29].

6 Related Work

The most closely related work to ours is the Jov [30] and JCrasher [9] tools, which share the goal of selecting, from a randomly-generated set of candidate inputs, a set most likely to be useful. This reduces the number of test inputs a human must examine.

Our research was inspired by Jov [30]. Jov builds on earlier work [15] that identified a test as a potentially valuable addition to a test suite if the test violates an operational abstraction built from the suite: the test represents some combination of values that differs from all tests currently in the suite. (The DIDUCE tool [14] takes a similar approach, though with the goal of identifying bugs at run time rather than improving test suites: a property that has held for part of a run, but is later violated, is suggestive of an error.) The Jov tool uses the operational abstraction not just to select tests, but also to guide test generation, by iterated use of the Jtest tool [19]. Jov also differs from the previous, automated work on test selection [15] by placing it in a loop with human interaction and iterating as many times as desired:

1. Create an operational model (invariants) from a test suite.
2. Generate test inputs that violate the invariants.
3. A human selects some of the generated tests and adds them to the test suite.

Often, overconstrained preconditions rendered Jtest incapable of producing any outputs, so Xie and Notkin report on the effectiveness of Jov after eliminating all preconditions from the operational model generated in step 1. Essentially, this permitted Jtest to generate any input that violates the postconditions (including many illegal ones), not just inputs similar to the ones in the original test suite. However, the user gets no help in recognizing such illegal inputs. In fact, the majority of errors that Jov finds [30] are illegal inputs and precondition violations, not true errors [27].

Our work extends that of Xie and Notkin in several ways. Our technique explicitly addresses the imperfect nature of a derived operational model. Our technique explicitly distinguishes between illegal and fault-revealing inputs. Our technique is more automated: it requires only one round of examination by a human, rather than multiple rounds. Our technique uses operational abstractions in a different way to direct test input generation. Our implementation is more robust and faster; Eclat takes less than two minutes for a class that took Jov over 10 minutes to process, primarily because the Jtest tool is so slow. We have performed a more extensive experimental evaluation (631 classes rather than 12). Even though we count only actual errors, not illegal inputs, our approach outperforms the previous one.

JCrasher [9], like Eclat, generates a large number of random inputs and selects a small number of potentially fault-revealing ones. An input is considered potentially fault-revealing if it throws an undeclared runtime exception. Inputs are grouped (reduced) based on the contents of the call-stack when the exception is thrown. JCrasher and Eclat have similar underlying generation techniques but different models of correct program behavior, which leads to different classification and reduction techniques. JCrasher's model takes into account only exceptional behavior, and Eclat augments the model with operational behavior, which accounts for its greater effectiveness in uncovering faults.

6.1 Future Work

Future work on this research centers around two themes.

- **Input generation.** While it may not help in establishing the reliability of a program, random testing seems to be remarkably effective in exposing errors and may be as effective as more formally founded techniques [10, 13]. However, it is primarily useful when all inputs are legal, or when a specification of valid inputs is available. Therefore, techniques that make it more effective are valuable contributions. Our technique could be combined with any technique for generating tests [8, 4], in order to filter the tests before being presented to a user. Our technique is attractive because it does not require a human-written formal specification; when one is present, much more powerful testing methodologies are possible [2, 7].
- **Input classification.** Eclat's reduction step clusters test inputs in order to reduce their number, and JCrasher has a similar step. Several researchers have used machine learning to classify program executions as either correct or faulty [20, 5, 3]. It would be interesting to apply such techniques in order to further improve Eclat.

7 Conclusion

We have presented an input selection technique that incorporates a classifier and a reducer, both of which make use of a model of correct program operation. We have combined our input selection technique with two other techniques. One technique uses the classifier to guide input generation towards legal inputs, which improves the efficiency of the input search space by pruning illegal sequences of methods calls as early as they are encountered. The other additional technique uses the operational model to produce oracles for the selected test inputs, which converts the test inputs into full-fledged test cases. Together, these techniques result in an effective test generation and selection methodology.

We have implemented the methodology in Eclat, a tool for Java unit testing, and demonstrated its effectiveness in producing fault-revealing test inputs. The input generation technique creates legal, fault-revealing candidate inputs for the methods in our subject programs, and the input selection technique selects inputs that are an order of magnitude as likely to reveal faults as the candidate inputs. The methodology reveals real, previously unknown errors in the subject programs. When the test inputs fail to reveal faults, the user is not heavily inconvenienced, because only a few inputs are selected.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, Jan. 16–18, 2002.

[2] M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 210–218, Dec. 1989.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 195–205, Boston, MA, USA, July 12–14, 2004.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 123–133, Rome, Italy, July 22–24, 2002.

[5] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.

[6] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, Trondheim, Norway, June 5–7, 2003.

[7] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 285–302, Toulouse, France, Sept. 6–9, 1999.

[8] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00, Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, Montreal, Canada, Sept. 18–20, 2000.

[9] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1117, Sept. 2004.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[12] Foundations of Software Engineering group, Microsoft Research. *Documentation for AsmL 2*, 2003. `http://research.microsoft.com/fse/asml`.

[13] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, Dec. 1990.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 2002.

[15] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[16] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP 2003 — Object-Oriented Programming, 17th European Conference*, pages 431–456, Darmstadt, Germany, July 23–25, 2003.

[17] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215. ACM Press, 1996.

[18] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[19] Parasoft Corporation. *Jtest version 4.5*. `http://www.parasoft.com/`.

[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, Portland, Oregon, May 6–8, 2003.

[21] G. Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[22] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proceedings of 2nd XP Universe and 1st Agile Universe Conference (XP/Agile Universe)*, pages 131–143, Chicago, IL, USA, Aug. 4–7, 2002.

[23] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE'98)*, pages 285–288, Honolulu, Hawaii, Oct. 14–16, 1998.

[24] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

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

[26] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 218–228, Rome, Italy, July 22–24, 2002.

[27] T. Xie. Personal communication, Aug. 2003.

[28] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE 2004: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 196–205, Linz, Australia, Nov. 9–11, 2004.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, Apr. 4–8, 2005.

[30] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 40–48, Montreal, Canada, Oct. 8–10, 2003.