Eclat: Automatic Generation and Classification of Test Inputs

Carlos Pacheco Michael D. Ernst
Technical Report MIT-LCS-TR-968
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 helps a test engineer select, from a large set of randomly generated 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 normal executions — say, from an existing test suite, or from observations of the software running properly. The technique works by extracting an operational model of the software's operation, and comparing each input's operational pattern of execution against the model. Test inputs whose operational pattern is suggestive of a fault are further reduced by selecting only one input per such pattern. The result is a small portion of the original inputs, deemed most likely to reveal faults. Thus, our technique can also be seen as an error-detection technique.

We have implemented these ideas in the Eclat tool, designed for unit testing of Java classes. Eclat generates a large number of inputs and uses our technique to select only a few of them as fault-revealing. The inputs that it selects are an order of magnitude more likely to reveal faults than non-selected inputs.

1 Introduction

Exhaustive software testing is infeasible. Therefore, much of the skill in testing a software artifact lies in carefully constructing a small set of test cases that can reveal 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 inputs for a program [10, 18, 16, 23, 7, 3, 19, 8, 12], but constructing an oracle for each input still remains a largely manual task (unless a complete 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 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 a technique that helps the tester by selecting from a large number of program inputs a few inputs. The resulting inputs are likely to reveal faulty behavior, so writing full-fledged test cases is worth the effort. Our goal is not to take the tester out of the loop—a human must inspect the inputs and make test cases out of them. Our goal instead is to focus the tester's effort on inputs most likely to reveal faults. Thus, our technique can also be viewed as an error-detection technique.

The 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 set of program executions, 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 heuristically 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.

We have implemented these ideas in the Eclat tool, which generates unit tests for Java classes. Eclat uses a simple generation strategy to create new inputs for a class, and it selects a small number of inputs likely to reveal faults. Our experiments demonstrate that our technique is effective. An input selected as potentially fault-revealing is 10 times more likely to reveal a fault than a non-selected input.

The rest of the paper is structured as follows. Section 2 introduces our technique with an example. Section 3 describes the test selection technique in detail. Section 4 describes Eclat, a tool for unit testing of Java classes that em-

```java
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;
  }
}
```

Figure 1. Class `BoundedStack` (abbreviated). Methods `pop()` and `equals()` contain errors.

Eclat Report

Input 1

```java
BoundedStack var8 = new BoundedStack();
var8.isMember(2);
```

The last method invocation violated these properties:

On entry: k in elems[]
On entry: elems[numElems..] sorted by >
⋆ ***On exit: orig(k) in elems[]***
On exit: elems[numElems..] sorted by >

During execution of the last method invocation, at least one high-confidence postcondition was violated (denoted by a star next to it). Since no high-confidence preconditions were violated, this suggests a fault.

Input 2

```java
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

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

The last method invocation violated these properties:

On entry: numElems one of 1, 2
On exit: numElems one of 0, 1
On exit: elems[numElems] == elems[orig(numElems)-1]
⋆ ***On exit: numElems >= 0***
On exit: elems[numElems] == orig(elems[numElems-1])

During execution of the last method invocation, at least one high-confidence postcondition was violated (denoted by a star next to it). Since no high-confidence preconditions were violated, this suggests a fault.

Input 4

```java
BoundedStack var8 = new BoundedStack();
var8.push(3);
int var7 = var8.numElems();
var8.push(var7);
```

The last method invocation violated these properties:

On entry: k != numElems
On exit: max <= elems[numElems-1]
On exit: elems[0..numElems-1] elements one of 2, 3
On exit: elems[] elements one of 0, 2, 3
On exit: numElems <= elems[numElems-1]
⋆ ***On exit: size(elems[])-1 != elems[max-1]***

During execution of the last method invocation, at least one high-confidence postcondition was violated (denoted by a star next to it). Since no high-confidence preconditions were violated, this suggests a fault.

Figure 2. Eclat's output for `BoundedStack`. Inputs 2 and 3 expose errors in the code under test. Inputs 1 and 4 are false reports: they merely indicate deficiencies in the original test suite.

bodies this paper's ideas. Section 5 details the experimental evaluation of the technique, and Section 6 discusses its applicability. Section 7 discusses related work, and Section 8 concludes.

2 Example: BoundedStack

We illustrate the technique with a bounded stack implementation used previously in the literature [22, 27, 8] (Figure 1). The implementation 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 (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 of the class. In this example, we provide Eclat with the stack implementation and the 8-test passing test suite.

Eclat's output (Figure 2) is a set of 4 new inputs, each accompanied by an explanation of why the input suggests a fault, and a set of violated properties. Each violated property was true during execution of the original test suite, but was violated by the new input.

Input 1 violates four properties during the call of method `isMember`: two on entry and two on exit. As Eclat's textual explanation points out, only one of the four violations leads the tool to classify this input as likely to expose a fault: when `isMember(k)` is called, `k` fails to be an element of the array `elems[]`. This input reveals no fault; Eclat has made a mistake. The input, however, does point out a rather glaring omission in the test suite, which should be augmented to include a call to `isMember` with an element not already in `elems[]`.

Input 2 violates no properties, but throws an exception from within the `equals` method. Eclat considers the exception severe and suggestive of a fault. Inspection of the `equals` method (Figure 1) reveals that the method incorrectly handles a `null` argument. This fault went undetected in all previous analyses of the class [22, 27, 8].

Input 3 violates five properties, but only one leads to its classification as a fault. The variable `numElems` becomes negative after a call of `pop` on an empty stack. Eclat has revealed another true error; the `pop` method (Figure 1) always decrements the top-of-stack pointer, even on an empty stack. This is a particularly subtle error, because it silently corrupts the stack's state, and a fault only arises on a subsequent access to the stack.

Input 4 is much like Input 1. It reflects a peculiarity of the test suite, not an error in the program.

Eclat selects 4 inputs that quickly lead a user to discover two errors. Behind the curtains, Eclat generates and analyzes 475 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 despite violating some properties, none is considered severe. Finally, some inputs are discarded because they behave too similarly to already-chosen inputs: 3 of the 475 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 The Technique

We describe the classification technique in the context of unit testing in an object-oriented programming language. An input is a method call, possibly preceded by other statements that set up state for the call. For example, Input 4 in Figure 2 tests the method `push` via the method call `var8.push(var7)`, and the first three statements are setup. The technique can also be applied to non-object-oriented programs and to components larger than methods and constructors.

3.1 Selecting inputs likely to reveal faults

This section describes the technique for selecting inputs likely to reveal faults. The technique requires three things: (1) the program under test, (2) a set of normal executions of the program (for instance, a small initial test suite for the program), and (3) a source of candidate inputs (the candidates may constitute illegal inputs, or cause the program to behave normally, or reveal a fault). The goal is to select a small subset of the candidates likely to reveal faulty program behavior. The Eclat implementation only requires the user to provide (1) and (2), because it automatically generates the candidates (Section 3.2).

The technique has three main steps:

1. Observe the program's behavior on the provided correct executions, and create an *operational model* of correct behavior (Section 3.1.1).
2. Classify each candidate as (1) *illegal*, (2) *normal operation*, or (3) *fault-revealing*. Do this by running the program on each candidate and comparing the program's behavior against the operational model (Section 3.1.2). Discard candidates classified as *illegal* or *normal operation*.
3. Partition the *fault-revealing* candidates based on their *violation pattern*: the set of violated properties, if any. Report one candidate from each partition.

3.1.1 Modeling previous program behavior

The technique uses an operational model of the program to classify inputs. An operational model consists of a set of properties that hold at the boundary of the program's components (e.g., on public method entry and exit). The technique imposes no limitation on the program behavior captured by a property, but it requires that every property is evaluable in the context of a new program execution.

The technique also requires that every property has an associated number indicating the likelihood that the property is universally true of all program executions. We call this the *confidence measure* of the property. Properties with higher confidence measure are more likely to be universally true. The way that confidence measures are calculated is left up to the model extraction procedure.

Figure 3 shows a simple operational model for `BoundedStack`. In this model, properties are mathematical statements about the state of the stack at various program points, and have a confidence value of *high* or *low*.

3.1.2 Classifying an input

The classifier takes four things: a candidate input, the program under test, an operational model, and a confidence

Invariants properties (hold on entry and exit to all methods)	
(conf=high)	$\texttt{max} = \texttt{elems.length}$
(conf=high)	$\texttt{elems} \neq \texttt{null}$
(conf=high)	$\texttt{max} = 2$
(conf=high)	$\texttt{numElems} \geq 0$
Properties that hold on entry to `pop`	
(conf=low)	$\texttt{elems}[] \in \{[3,0],[3,2]\}$
Properties that hold on exit from `pop`	
(conf=low)	$\forall i : \texttt{numElems} \leq i < \texttt{elems.length} : \texttt{elems}[i] = 0$
Properties that hold on entry to `isMember`	
(conf=low)	$\texttt{k} \in \texttt{elems}[]$
Properties that hold on exit from `isMember`	
(conf=low)	$\texttt{elems}[] = orig(\texttt{elems}[])$
(conf=low)	$orig(\texttt{k}) \in \texttt{elems}[]$

Figure 3. A simple operational model for `BoundedStack`. Each property has an associated confidence value: high or low.

Severe entry violations?	Severe exit violations?	Classification
yes	yes	***illegal***
—	no	***normal operation***
no	yes	***fault-revealing***

Figure 4. Decision table for classifying a violation pattern.

threshold. The classifier runs the program on the candidate input and collects the (possibly empty) set of violated model properties. Violations of properties with confidence measure above the threshold are considered *severe*, and violations of properties with confidence below or at threshold are considered *mild*.

Figure 4 shows how to classify the input based on the set of violated properties:

1. *Illegal*. One or more severe entry violations occur, and one or more severe exit violations occur as well.
2. *Normal operation*. No severe exit violations occur.
3. *Fault-revealing*. No severe entry violations occur, but one or more severe exit violations occur.

3.1.3 Reducing inputs based on violation patterns

A violation pattern not only determines the classification of an input. It also induces 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. The technique presents only one input from each partition.

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

```
BoundedStack var1 = new BoundedStack();
var1.push(1);
var1.pop();
var1.pop();
```

```
BoundedStack var2 = new BoundedStack();
var2.push(0);
int var3 = var2.numberOfElements();
var2.pop();
var2.isMember(var3);
var2.pop();
```

Figure 5. Three Eclat-generated inputs that reveal the same fault in the `pop` method.

In practice, we have observed that the reduction technique works best if only high-confidence properties are considered when building partitions. Consider the example in Figure 5. All inputs pop an empty stack, which results in an erroneous stack state. The inputs violate different sets of properties, but they all violate exactly the same high-confidence property: $\texttt{numElems} \geq 0$ on exit from `pop()`.

3.2 Efficient input space exploration

We have presented a technique that selects from a set of candidate inputs a subset likely to reveal faults. In this section, we use the technique to avoid generating illegal inputs in a standard bottom-up input generation strategy. Generation starts with a small set of initial values (for example, in Java, a few primitive values and `null`). New inputs are created by calling methods and constructors with these values as arguments. The resulting objects and primitives are added to the value pool, and the process is repeated any number of times.

We now present two enhancements to bottom-up input generation. Eclat implements these enhancements in its input generation phase.

1. **Avoiding illegal inputs.** As before, the algorithm proceeds in rounds. For each round:
   (a) Construct a new set of inputs from the existing pool of values.
   (b) Classify the new inputs using the technique from Section 3.1.2.
   (c) Discard inputs labeled *illegal*, add inputs labeled *normal operation* to the pool, and save inputs labeled *fault-revealing* (but do not add them to the pool).

   After the last round, reduce the saved *fault-revealing* inputs and select one input from each partition.

2. **Choosing initial values from sources.** The initial pool of values is important: all inputs are derived from them. Input generation can be enhanced by adding to the initial pool all the primitive values found in the program source code (and the test suite's code, if available). This increases the chances that relevant constants (e.g. constants used in conditional expressions) are used to generate inputs.

4 Eclat

We have implemented our ideas in Eclat, a tool that suggests potentially fault-revealing inputs for Java classes.

Eclat takes as input a class C (more precisely, a source file `C.java`), and a program P that uses the class. Eclat performs the following steps.

Deriving an operational model. Eclat uses the Daikon dynamic invariant detector [11, 9] to derive a model of C's behavior on P; an example of Daikon's output appeared in Figure 3. Eclat assigns a confidence (low or high) to each property based on a set of heuristics. Here are the two most important ones:

- Properties such as object invariants, which hold at all program points, inspire higher confidence than method entry and exit properties, because the former are derived from more program samples.
- Properties relating the input and output of a method call inspire higher confidence than properties solely describing the range of inputs or outputs of the method, because input-output relationships are less sensitive to the particular set of executions from which the model was derived.

A valuable extension to this work would be the development of statistical methods to mechanically compute confidence measures and thresholds for particular properties.

Eclat treats exceptions as violations. Some kinds of unchecked exceptions are considered severe; all other are considered mild.

Compiling for runtime property checking. As part of this research, we have implemented the Jicama compiler. Jicama uses the Java Modeling Language (JML) toolset [17, 5] to convert a Daikon model into code that can be checked at runtime. Jicama's input is the source file of the tested class (say, `C.java`) and the operational model derived by Daikon. It produces a class file `C.class` instrumented to check model properties during execution. Jicama's instrumentation is transparent: a violation does not alter the behavior of the class. Violated properties are simply recorded in a log.

Generating new inputs. Finally, Eclat generates, classifies and reduces new inputs. It uses Java's reflection mechanism to execute them. Eclat's output is an XML file, viewable in a web browser (as shown in Figure 2) or manipulable programmatically. The file contains a list of inputs, along with the properties they violated and a brief explanation of the reason each input is considered fault-revealing.

Program	versions	formal spec?	classes	public methods	NCNB LOC
BoundedStack	1	yes	1	11	88
DSAA	1	no	8	64	640
JMLSamples	1	yes	28	174	1392
utilMDE	1	no	2	188	1832
RatPoly	97	yes	1	17	512
Directions	80	yes	6	42	342

Figure 6. Subject programs. For programs with multiple versions, numbers are average per version. NCNB means non-comment, non-blank lines of code.

5 Evaluation

This section quantifies our technique's effectiveness on a set of Java programs. Section 5.1 introduces the programs and experimental methodology. Section 5.2 evaluates the fault-revealing characteristics of Eclat's selected inputs. Finally, Section 5.4 evaluates the technique's three internal components: the input generator, the classifier, and the reducer.

5.1 Subject Programs

Figure 6 lists our subject programs. The programs encompass 46 distinct interfaces, and a total of 567 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; we did not use synthetic fault-injection techniques, which can have very different characteristics.

Four of the six subject programs have formal specifications. Of course, in the presence of a formal specification our technique is not necessary: the specification can determine if an input is illegal, normal, or fault-revealing. We use the specifications to evaluate our technique, with the specification representing an ideal classifier.

- BoundedStack is the stack implementation discussed in Section 2. We wrote the formal specification. We report separately the results of running Eclat with the 8-test suite, and with the 12-test suite.
- 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 28 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 class that augments the `java.util` package. We report two results: one running Eclat with the test suite written by the authors of the class, and the other with a sample run of an unrelated program 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 programs implement the core of a graphing calculator for polynomials over rational numbers. We wrote the formal specification. The course staff provided a test suite to the students as part of the assignment. Successful completion of the assignment meant passing all tests in the suite (which most, but not all, students accomplished).
- Directions is a different set of student solutions in MIT class 6.170, written by the same students that wrote the RatPoly solutions. The Directions library is used by a MapQuest-like program that outputs directions for going from one location to another along Boston-area streets. This time, 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. We wrote the formal specifications.

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. (Eclat re-discovered the failure, and also revealed a previously unknown fault.)

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 6 count only versions we kept.

Measurements. We tested BoundedStack by running Eclat on its single class. For DSAA, JMLSamples, and utilMDE, which contain unrelated sets of classes, we ran Eclat separately on each class. (For example, DSAA contains unrelated implementations of a binary tree, a disjoint set, a treap, a stack, a queue, a red-black tree, and a linked list.) For these programs, the figures we report are averages per run. For RatPoly and Directions, we ran Eclat once for each program version, and also report averages per run. In computing global results (results that span several subject programs), we give the same weight to each subject program, regardless of the number of versions or runs of Eclat that the program represents. We do this to avoid over-representing programs with multiple versions or runs.

Comparison with other tools. JCrasher [8], Jtest [19], and Jov [27] have the same goals as Eclat: to generate random inputs and select potentially fault-revealing ones. We report results from running JCrasher. Jov and Jtest were unusable in many instances (e.g., Jov sometimes exited abnormally, and Jtest sometimes failed to terminate).

true label	Generated inputs		Selected inputs	
	total	fraction	total	fraction
normal	1606	0.90	6.27	0.56
illegal	213	0.08	0.91	0.22
fault	38	0.02	1.23	0.22
total	1856		8.40	

Figure 8. Average generated and selected inputs for BoundedStack, JMLSamples, RatPoly and Directions (the programs with formal specifications). These results represent a total of 440,000 inputs.

5.2 Evaluating Eclat's output

Figure 7 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 end-to-end results on the subject programs. For programs with formal specifications, we counted an input as fault-revealing (mechanically, using `jmlc` [5]) if it satisfied all preconditions of the tested method, and the method invocation caused a postcondition violation. For programs without formal specifications (DSAA and utilMDE), we manually inspected each selected input, erring on the side of declaring an input non-faulty unless the input was clearly legal and caused incorrect behavior. On average, Eclat selected 6.87 inputs per run, and approximately 1.03 of those revealed a fault. By comparison, JCrasher selected on average 11.79 inputs per run, and approximately 0.02 revealed a fault.

For the four programs with formal specifications, we also determined the *true label* of every generated input, i.e. the label assigned by the formal specification. Figure 8 summarizes the results. The inputs that the technique selects are 11 times more likely to reveal faults than non-selected inputs: 22% of selected inputs reveal faults, faults, and 2% of non-selected inputs reveal them.

Program	Generated inputs: inputs generated	Generated inputs: reveal faults	Selected inputs: inputs selected	Selected inputs: reveal faults	JCrasher inputs: inputs selected	JCrasher inputs: reveal faults
BoundedStack (8-test suite)	475	12	4.00	2.00		
BoundedStack (12-test suite)	854	13	2.00	1.00	0.00	0.00
DSAA	400	n/a	0.38	0.00	0.00	0.00
JMLSamples	264	2	0.50	0.00	2.43	0.00
utilMDE (test suite)	423	n/a	6.00	1.00		
utilMDE (sample run)	3510	n/a	5.00	1.00	62.00	0.00
RatPoly	2862	17	2.48	0.33	4.42	0.11
Directions (student suite)	3197	93	15.58	1.21		
Directions (staff suite)	3484	86	25.86	2.77	1.91	0.00
average	1719	37	6.87	1.03	11.79	0.02

Figure 7. Summary of Eclat's results. The first two numeric columns represent inputs internally generated by Eclat. The "n/a" entries denote programs without oracles, for which do not know how many of all internally-generated inputs reveal faults. The next two columns represent inputs selected and reported to the user. The last two columns represent inputs selected as fault-revealing by JCrasher.

	Generated inputs: inputs generated	Generated inputs: reveal faults	Selected inputs: inputs selected	Selected inputs: reveal faults
original trace	1719	37	6.87	1.03
10% of trace	1722	34	8.17	0.79

Figure 9. Running Eclat on a reduced execution trace. The first line repeats the "average" line of Figure 7, and the second line reports the results for the reduced trace.

5.3 Sensitivity to the initial set of program runs

As Figure 7 shows, when we ran Eclat on the same program but on a different set of program runs, the number of inputs that Eclat generated and selected changed. But one thing remained close to constant: the fraction of selected inputs that were fault-revealing. In other words, the chances of a selected input being fault-revealing—the main measure of the tool's effectiveness—is not too sensitive to the set of correct executions given. In a second experiment, we artificially reduced the set of correct executions used by Eclat to construct an operational model. We re-ran the experiments of Figure 7, using only the first 10% of the execution trace that had been used to generate the operational model. Figure 9 shows the results.

When given a smaller trace, Eclat selected more inputs, and these inputs revealed fewer faults. The difference is not large, considering that we reduced the trace by an order of magnitude: for the original trace, 1 in 7 inputs are faults revealing; for the 90%-reduced trace, about 1 in 10. Eclat is effective even with impoverished traces, because it assigns high confidence to properties like object invariants, which are established even for small traces. (e.g., BoundedStack's `numElems > 0`). Properties that are more dependent on a particular set of executions (e.g., BoundedStack's `k ∈ elems[]` on entry to `isMember(k)`) are assigned lower confidence, and play no role in the generation and selection process.

5.4 Evaluating Eclat's components

Eclat consists of three main components: an input generator, a classifier, and a reducer. In the following sections, we evaluate each component in turn, to better understand how and why the technique works.

5.4.1 The input generator

Eclat generated on average 1719 inputs per run. Of these, 37 were fault-revealing. Random generation produced fault-revealing inputs even for programs that passed passed their original test suite. For example, it revealed an error in the staff solution for RatPoly, and in a method of the utilMDE library that went undetected in a test suite with hundreds of unit tests.

Faults in specifications. While testing formal specifications was not part of our goals, running all the inputs that Eclat generated for JMLSamples against their formal specifications revealed a number of errors in the specifications (the inputs revealed no errors in the implementation). This is encouraging because the JMLSamples specifications were written as part of the JML formal methods project [17], and the fact that our test generation strategy revealed errors that the program's test suite missed illustrates the effectiveness of the generation strategy.

true label	Eclat label normal	illegal	fault	recall
normal	0.731	0.023	0.109	0.84
illegal	0.039	0.043	0.038	0.36
fault	0.007	0.002	0.007	0.36
precision	0.93	0.43	0.09	

Figure 10. Each entry shows the average proportion of generated inputs with Eclat label and true label, for BoundedStack, JMLSamples, RatPoly and Directions. The nine middle entries sum to 1.

5.4.2 The Classifier

For the four formally-specified programs, every input has two labels, one assigned by Eclat and the other one assigned by the formal specification (i.e., the true label). Figure 10 shows the proportion of inputs falling into each ⟨Eclat label, true label⟩ category.

The last row in the figure shows the *precision* [21, 24] of Eclat's classifier. Precision is defined as the ratio of correct labellings to the total number of labellings:

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

The last column in the figure shows the *recall* of the classifier. Recall is defined as the ratio of correct labellings 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 (0.93 precision, 0.84 recall for normal inputs),
- recognizes more than a third of all fault-revealing inputs (0.36 recall for fault-revealing inputs), but
- labels fault-revealing many inputs that are not (0.09 precision for fault-revealing inputs)

Another way of characterizing the classifier is as a *pessimistic classifier*, in the sense that it classifies many normal inputs as illegal, not the other way around. A pessimistic classifier is preferable, because it sheds light on the weakness of the existing test suite. 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. In cases like BoundedStack's, inspecting spurious illegal inputs can help find weaknesses in a test suite.

Identifying new behavior. We experimented with a four-label set that split *normal operation* into *old* and *new* behavior. Inputs that violated no properties were labeled *old behavior*—these are inputs that did not diverge at all from the test suite. Inputs that violated only low-confidence properties were labeled *new behavior*. We found that new behaviors were no more effective than old behaviors in revealing faults. 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.

5.4.3 The Reducer

The reducer takes the inputs labeled *fault-revealing*, and selects a subset. The table below summarizes its behavior for the four programs with formal specifications. The first numeric column shows the average distribution of all inputs that the classifier labeled *fault-revealing* (the input to the reducer). The last column shows the distribution of inputs selected (the output of the reducer). Each column sums to 1.

true label	labeled as fault by classifier	reduced (selected)
normal	0.58	0.56
illegal	0.32	0.22
fault	0.10	0.22

The reduction step increases the proportion of fault-revealing inputs by a factor of 2. 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.

6 Discussion

Applicability. We have presented our test selection technique in the context of an object-oriented programming language. The technique is 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).

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. 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 doesn't care whether properties are obtained manually or mechanically, and its use of confidence measures makes it amenable to mixing derived properties with partial specifications: the latter simply translate to high-confidence properties. The operational model can benefit from manually-written specifications that capture important properties not mechanically derived. On the other hand, partial specifications can benefit from derived properties that may be crucial for the input generation and classification process to be effective.

7 Related Work

The most closely related work to ours is the Jov [27] and JCrasher [8] tools, which share the goal of selecting, from a randomly-selected set of test 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 [27]. 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 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 enhances the previous, automated work on test selection by placing it in a loop with human interaction and iterating as many times as desired:

1. Create an operational model (invariants) for 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 [27] are illegal inputs and precondition violations, not true errors [26].

Our work extends that of Xie and Notkin in several ways. Our technique explicitly addresses the imperfect nature of a derived operational model, for instance by using confidence measures for model properties. 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 much faster; it 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 (567 classes rather than 12). Even though we count only actual errors, not illegal inputs, our approach outperforms the previous one.

JCrasher [8], 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.

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 [7, 3], in order to filter the tests before being presented to a user. Our technique is attractive because it does not require a formal specification; when one is present, much more powerful testing methodologies are possible [1, 6].

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, 4, 2]. It would be interesting to apply such techniques in order to further improve Eclat.

8 Conclusion

We have presented an input selection technique that combines a classifier and a reducer, both of which make use of a model of correct program operation. The inputs that the technique selects are an order of magnitude more likely to reveal faults than non-selected inputs. When the technique fails, the user is not heavily inconvenienced, because only a few inputs are selected.

Testing by comparing a program's execution against a model of correct program behavior is not a new idea. Models are typically written by hand, which is often too costly to do in practice. The software engineering community has developed automatic—if imprecise—techniques that infer program models. By allowing for imprecision in the model, our technique provides many of the advantages of testing against a model, without requiring the test engineer to write the model by hand.

References

[1] M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. In R. A. Kemmerer, editor, *TAV3*, pages 210–218, Dec. 1989.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, pages 195–205, July 2004.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, July 2002.

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, May 2004.

[5] 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 *FMICS*, Trondheim, Norway, June 2003.

[6] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *ESEC/FSE*, pages 285–302, Sept. 6–10, 1999.

[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, Sept. 2000.

[8] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*. To appear.

[9] *The Daikon Invariant Detector User Manual*, Dec. 7, 2001. Version 2.3.2. `http://pag.csail.mit.edu/daikon/`.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE TSE*, 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 TSE*, 27(2):1–25, Feb. 2001.

[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 TSE*, 16(12):1402–1411, Dec. 1990.

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

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

[16] 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.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[18] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *CACM*, 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*, pages 465–475, May 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 *XP/Agile Universe*, pages 131–143, Aug. 2002.

[23] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *ASE '98*, pages 285–288, Oct. 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] T. Xie. Personal communication, Aug. 2003.

[27] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003*, pages 40–48, Oct. 2003.