Finding latent code errors via machine learning over program executions

Yuriy Brun
Laboratory for Molecular Science
University of Southern California
Los Angeles, CA 90089 USA
brun@alum.mit.edu

Michael D. Ernst
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
mernst@csail.mit.edu

Abstract

This paper proposes a technique for identifying program properties that indicate errors. The technique generates machine learning models of program properties known to result from errors, and applies these models to program properties of user-written code to classify and rank properties that may lead the user to errors. Given a set of properties produced by the program analysis, the technique selects a subset of properties that are most likely to reveal an error.

An implementation, the Fault Invariant Classifier, demonstrates the efficacy of the technique. The implementation uses dynamic invariant detection to generate program properties. It uses support vector machine and decision tree learning tools to classify those properties. In our experimental evaluation, the technique increases the relevance (the concentration of fault-revealing properties) by a factor of 50 on average for the C programs, and 4.8 for the Java programs. Preliminary experience suggests that most of the fault-revealing properties do lead a programmer to an error.

1 Introduction

Programmers typically use test suites to detect faults in program executions, and thereby to discover errors in program source code. Once a program passes all the tests in its test suite, testing no longer leads programmers to errors. However, the program is still likely to contain latent errors, and it may be difficult or expensive to generate new test cases that reveal additional faults. Even if new tests can be generated, it may be expensive to compute and verify an oracle that represents the desired behavior of the program.

The technique presented in this paper can lead programmers to latent code errors. The technique does not require a test suite for the target program that separates succeeding from failing runs, so it is particularly applicable to programs whose executions are expensive to verify. The expense may result from difficulty in generating tests, from difficulty in verifying intermediate results, or from difficulty in verifying visible behavior (as is often the case for interactive or graphical user interface programs).

The new technique takes as input a set of program properties for a given program, and outputs a subset of those properties that are more likely than average to indicate errors in the program. The program properties may be generated by an arbitrary program analysis; the experiments reported in this paper use a dynamic analysis, but the technique is equally applicable to static analyses.

Figure 1 gives a simple example to motivate the technique. It shows some erroneous code, the properties that a dynamic analysis would produce for that code, the order in which the technique would rank those properties, and which of the properties truly are fault-revealing.

The intuition underlying the error finding technique is that many errors fall into a few categories, that similar errors share similar characteristics, and that those characteristics can be generalized and identified. For example, three common error categories are off-by-one errors (incorrect use of the first or last element of a data structure), use of uninitialized or partially initialized values, and exposure of representation details to a client.

The technique consists of two steps: training and classification. In the training step, the technique uses machine learning to train a model on properties of erroneous programs and fixed versions of them; it creates a machine learning model of fault-revealing properties, which are true of incorrect code but not true of correct code. (The experiments evaluate two different machine learning algorithms: support vector machines and decision trees.) In the classification step, the user supplies the precomputed model with properties of his or her code, and the model selects those properties that are most likely to be fault-revealing. A programmer searching for latent errors or trying to increase confidence in a program can focus on those properties.

This technique may be most useful when important errors in a program are hard to find. It is applicable even when a developer is already aware of (low-priority) errors. For example, the machine learning models could be

```
// Return a sorted copy of the argument.
double[] bubble_sort(double[] in) {
  double[] out = array_copy(in);
  for (int x = out.length - 1; x >= 1; x--)
    // lower bound should be 0, not 1
    for (int y = 1; y < x; y++)
      if (out[y] > out[y+1])
        swap(out[y], out[y+1]);
  return out;
}
```

Ranked properties	Fault-revealing?
out[1] $\leq$ in[1]	Yes
$\forall$ i: in[i] $\leq$ 100	No
in[0] = out[0]	Yes
size(out) = size(in)	No
in $\subseteq$ out	No
out $\subseteq$ in	No
in $\neq$ null	No
out $\neq$ null	No

Figure 1. A simple example illustrating our technique. The technique ranks code properties and outputs the highest-ranked ones for user examination. The goal is to identify fault-revealing properties, which are true of erroneous code but not true of correct code. Figure 5 shows how the ranking might be determined.

trained on properties of past projects' most critical errors, such as those that required updates in the field, those that caused problems that were escalated to upper management, or those that were hardest to discover or reproduce (such as concurrency-related problems). Use of such models may reveal similar problems in the program being analyzed.

We have implemented the technique in a fully automated tool called the Fault Invariant Classifier. It automatically determines the properties via dynamic analysis, so the user only needs to provide a program and some inputs. The experiments demonstrate that the implementation is able to recognize fault-revealing properties of code — that is, properties of the erroneous code that are not true of a fixed version. For C programs, the output of the machine learning technique's implementation has average relevance 50 times that of the complete set of properties; for Java programs, the improvement is a factor of 4.8. (The relevance, or precision, of a set of properties is the fraction of those properties with a given desirable quality.) Without use of the tool, the programmer would have to examine program properties at random or based on intuition.

The experiments indicate that a machine learner can identify fault-revealing program properties, which result from erroneous code. To determine whether these properties lead users to the errors, we conducted a study in which a programmer evaluated 410 of the reported properties and judged that 65% as many of them would have led him to the error as the relevance measurements indicated. If our experiments are characteristic of other situations, then programmers only need to examine (on average) 3 of the reported properties to locate an error.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 presents the Fault Invariant Classifier technique. Section 4 describes the components that comprise our prototype implementation. Section 5 describes the experimental evaluation, and Section 6 presents and analyzes the experimental results. Finally, Section 7 lays out future work, and Section 8 concludes.

2 Related Work

Our research aims to indicate to the user specific program properties that are likely to result from code errors. Several other researchers have taken a similar tack to locating errors.

Xie and Engler [26] demonstrate that program errors are correlated with redundancy in source code: files containing idempotent operations, redundant assignments, dead code, or redundant conditionals are more likely to contain an error. That research is complementary to ours in four respects. First, they use a statically computed metric, whereas we use a dynamic analysis. Second, they increase relevance for C programs by 45%–100%, whereas our technique increases relevance by an average of 4860% (a factor of 49.6). Third, their experimental analysis is at the level of an entire source file. By contrast, our technique operates on individual program properties. Rather than demonstrating that a file is more likely to contain an error at some unspecified location, our experiments measure whether the specific run-time properties identified by our technique (each of which involves two or three variables at a single program point) are more likely to arise as the result of an error. Fourth, they did not consider the use of their reports by humans, whereas we performed a preliminary investigation of this issue.

Like our research, Dickinson et al. [4] use machine learning over program executions, with the assumption that it is cheap to execute a program but expensive to verify the correctness of each execution. Their goal is to indicate which runs are most likely to be faulty. They use clustering to partition test cases, similar to what is done for partition testing, but without a guarantee of internal homogeneity. Executions are clustered based on "function call profile", or the number of times each procedure is invoked. Verifying the correctness of one randomly-chosen execution per cluster outperforms random sampling; if the execution is erroneous, then it is advantageous to test other executions in the same cluster. Their experimental evaluation measures the number of faulty executions detected rather than number of underlying faults detected. Our research identifies suspicious properties rather than suspicious executions, but

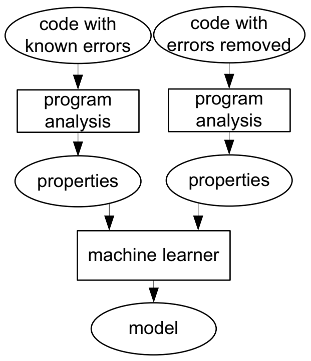


Figure 2. Creating a program property model. Rectangles represent tools, and ovals represent tool inputs and outputs. The model is used as an input in Figure 4. The program analysis is described in Section 4.1, and the machine learner is described in Section 4.3.

relies on a similar assumption regarding machine learning being able to make clusters that are dominantly faulty or dominantly correct.

Subsequent work by Podgurski et al. [15] uses clustering over function call profiles to determine which failure reports are likely to be manifestations of the same underlying error. A training step determines which features are of interest by evaluating which features enable a model to distinguish failures from non-failures, but the technique itself does not consider non-erroneous runs. In their experiments, for most clusters, the cluster contains failures resulting mostly from a single error. By contrast, we aim to identify errors.

Hangal and Lam [10] use dynamic invariant detection to find program errors. They detect a set of likely invariants over part of a test suite, then look for violations of those properties over the remainder of the test suite. Violations often indicated erroneous behavior. Our research differs in that it uses a richer set of properties; Hangal and Lam's set was very small in order to permit a simple yet fast implementation. Additionally, our technique can find latent errors that are present in most or all executions, rather than applying only to anomalies.

Groce and Visser [9] use dynamic invariant detection to determine the essence of counterexamples: given a set of counterexamples, they report the properties that are true over all of them. (The same approach could be applied to the successful runs.) These properties abstract away from the specific details of individual counterexamples or successes, freeing users from those tasks. Our research also generalizes over successes and failures, but applies the resulting models to future runs.

3 Technique

This section describes the technique of the Fault Invariant Classifier. The technique consists of two steps: training and classification. Training is a preprocessing step (Section 3.1) that extracts properties of programs containing known errors, converts these into a form amenable to machine learning, and applies machine learning to form a model of fault-revealing properties. During classification (Section 3.2), the tool applies the model to properties of new code and selects the fault-revealing properties, then the programmer uses those properties to locate latent errors in the new code.

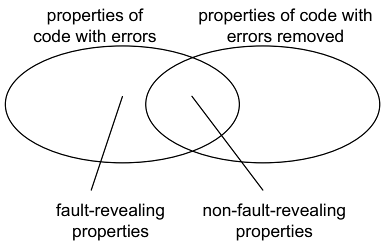

properties of code with errors
properties of code with errors removed
fault-revealing properties
non-fault-revealing properties

Figure 3. Fault-revealing program properties are those that appear in code with errors, but not in code without errors. Properties that appear only in non-faulty code are ignored by the machine learning step.

3.1 Creating Models

Figure 2 shows how to produce a model of error-correlated properties. This preprocessing step is run once, offline. The model is automatically created from a set of programs with known errors and corrected versions of those programs. First, program analysis generates properties of programs with errors and of programs with those errors removed. Second, a machine learning algorithm produces a model from these properties. Figure 4 shows how the technique uses the model to classify properties.

The training step requires pairs $\langle P, P' \rangle$ where P is a program containing at least one error and P' is a version of P with at least one error removed. The program P' need not be error-free, and the ones used in our experimental evaluation did contain additional errors [11]. The unknown errors do not hinder the technique; however, the model only captures the errors that are removed between the versions.

Before being inputted to the machine learning algorithm, each property is converted to a characteristic vector, as described in Section 4.2. Additionally, properties that are present only in faulty programs are labeled as fault-revealing, properties that appear in both faulty and non-faulty code are labeled as non-fault-revealing, and properties that appear only in non-faulty code are not used during training (Figure 3).

In order to avoid biasing the machine learning algorithms, the Fault Invariant Classifier normalizes the training set to contain equal numbers of fault-revealing and non-fault-revealing properties. This normalization is necessary because some machine learners interpret non-equal class

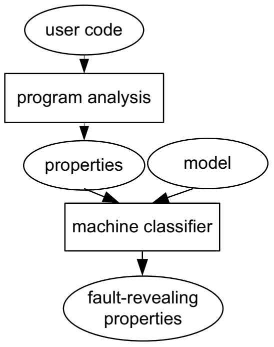


Figure 4. Finding likely fault-revealing program properties using a model. Rectangles represent tools, and ovals represent tool inputs and outputs. The model is produced by the technique of Figure 2.

sizes as indicating that some misclassifications are more undesirable than others.

3.2 Detecting Fault-Revealing Properties

Figure 4 shows how the Fault Invariant Classifier identifies fault-revealing properties of code. First, a program analysis tool produces properties of the target program. Second, a classifier ranks each property by its likelihood of being fault-revealing. A user who is interested in finding latent errors can start by examining the properties classified as most likely to be fault-revealing. Since machine learners are not guaranteed to produce perfect models, this ranking is not guaranteed to be perfect, but examining the properties labeled as fault-revealing is more likely to lead the user to an error than examining randomly selected properties.

The user only needs one fault-revealing property to detect an error, so the user should examine the properties according to their rank, until an error is discovered, and rerun the tool after fixing the program code.

4 Tools

This section describes the tools used in our Fault Invariant Classifier implementation. The three main tasks are to extract properties from programs (Section 4.1), convert program properties into a form acceptable to machine learners (Section 4.2), and create and apply machine learning models (Section 4.3).

4.1 Program Property Detector: Daikon

Our technique uses a program analysis in both the training and the classification steps (Figures 2 and 4). Both steps should use the same program analysis (or at least ones whose outputs are compatible), but the technique may use any program analysis, static or dynamic.

The prototype implementation, the Fault Invariant Classifier, uses a dynamic (runtime) analysis to extract semantic properties of the program's computation. (This choice is arbitrary; alternatives that do not require use of a test suite or execution of the program include using a static analysis (such as abstract interpretation [3]) to obtain semantic properties, or searching for syntactic properties such as duplicated code [26].) The dynamic approach is attractive because semantic properties reflect program behavior rather than details of its syntax, and because runtime properties can differentiate between correct and incorrect behavior of a single program.

We use the Daikon dynamic invariant detector to generate runtime properties [5]. Its outputs are likely program properties, each a mathematical description of observed relationships among values that the program computes. Together, these properties form an *operational abstraction* that, like a formal specification, contains preconditions, postconditions, and object invariants.

Daikon detects properties at specific program points such as procedure entries and exits; each program point is treated independently. The invariant detector is provided with a trace that contains, for each execution of a program point, the values of all variables in scope at that point.

For scalar variables x, y, and z, and computed constants a, b, and c, some examples of checked properties are: equality with a constant ($\mathsf{x} = \mathsf{a}$) or a small set of constants ($\mathsf{x} \in \{\mathsf{a},\mathsf{b},\mathsf{c}\}$), lying in a range ($\mathsf{a} \le \mathsf{x} \le \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), linear relationships ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), ordering ($\mathsf{x} \le \mathsf{y}$), and functions ($\mathsf{y} = \mathsf{fn(x)}$). Properties involving a sequence variable (such as an array or linked list) include minimum and maximum sequence values, lexicographical ordering, element ordering, properties holding for all elements in the sequence, and membership ($\mathsf{x} \in \mathsf{y}$). Given two sequences, some example checked properties are elementwise linear relationship, lexicographic comparison, and subsequence relationship. Finally, Daikon can detect implications such as "if $\mathsf{p} \neq \mathsf{null}$ then $\mathsf{p.value} > \mathsf{x}$" and disjunctions such as "$\mathsf{p.value} > \mathsf{limit}$ or $\mathsf{p.left} \in \mathsf{mytree}$".

The properties are sound over the observed executions but are not guaranteed to be true in general. In particular, different properties are true over faulty and non-faulty runs. The Daikon invariant detector uses a generate-and-check algorithm to postulate properties over program variables and other quantities, to check these properties against runtime values, and then to report those that are never falsified. Daikon uses additional static and dynamic analysis to further improve the output [6].

4.2 Property to Characteristic Vector Converter

Machine learning algorithms take characteristic vectors as input, so the Fault Invariant Classifier converts the properties reported by the Daikon invariant detector into this form. (This step is not shown in Figures 2 and 4.)

A characteristic vector is a sequence of boolean, integral,

Property	Equation				Variable type			#	Score
	$\leq$	$=$	$\neq$	$\subseteq$	int	double	array	vars	
`out[1]` $\leq$ `in[1]`	1	0	0	0	0	1	0	2	19
$\forall$ `i: in[i]` $\leq$ 100	1	0	0	0	0	1	0	1	16
`in[0]` = `out[0]`	0	1	0	0	0	1	0	2	15
size(`out`) = size(`in`)	0	1	0	0	1	0	0	2	13
`in` $\subseteq$ `out`	0	0	0	1	0	0	1	2	12
`out` $\subseteq$ `in`	0	0	0	1	0	0	1	2	12
`in` $\neq$ null	0	0	1	0	0	0	1	1	10
`out` $\neq$ null	0	0	1	0	0	0	1	1	10
Model weights	7	3	2	1	4	6	5	3	

Figure 5. The properties of Figure 1, their characteristic vectors (each characteristic vector consists of 8 numbers that appear in a row), and the scores assigned by a hypothetical weighting model.

and floating point values. It can be thought of as a point in multidimensional space. Each dimension (each value in the sequence) is called a slot.

As an example, suppose there were four slots to indicate the equation, three slots to indicate the type of the variables involved, and one slot to indicate the number of variables. Then Figure 5 shows the characteristic vectors for the properties of Figure 1. For example, the characteristic vector of `out[1]` $\leq$ `in[1]` is $\langle 1, 0, 0, 0, 1, 0, 0, 2\rangle$ (where 0 and 1 stand for false and true).

Figure 5 also gives a sample model that applies weights to the vectors' slots to rank the properties. (The actual models produced by our implementation are more sophisticated.) This model identifies `out[1]` $\leq$ `in[1]` as most likely to be fault-revealing. A programmer who examined that property would notice the off-by-one error in Figure 1.

A characteristic vector is intended to capture as much of the information in the property as possible. The machine learning algorithms of Section 4.3 can ignore irrelevant slots. Thus, it is advantageous to include as many slots as possible. In our experience, the specific choice of slots does not appear to be critical, so long as there are enough of them. We did not expend any special effort in selecting the slots.

In our implementation, the characteristic vectors contain 388 slots. A complete listing of the 388 slots appears elsewhere [1], and the code that extracts them is distributed with the Daikon tool. Here we briefly describe how they are obtained and give some examples. The Daikon invariant detector represents properties as Java objects. The converter uses reflection to extract all possible boolean, integral, and floating point fields and zero-argument method results for each property. Each such field and method fills exactly one slot. For instance, some slots of the characteristic vector indicate the number of variables in the property; whether a property involves static variables (as opposed to instance variables or method parameters); and the (floating-point) result of the null hypothesis test of the property's statistical validity [6]. 111 of the slots represent the equation of a property (e.g., equality such as $x = y$, or containment such as $x \in val\_array$). Each property's characteristic vector had 46 of its slots filled on average, and every slot was filled in at least one vector.

During the training step only, each characteristic vector is labeled as fault-revealing, labeled as non-fault-revealing, or ignored, as indicated in Figure 3. The machine learner builds models that refer to slots, not directly to properties. This is a crucial choice that lets the models generalize to other properties and programs than were used in the training step.

4.3 Machine Learning Algorithms

Machine learners generate models during the training step, and provide a mechanism for applying those models to new points in the classification step. Machine learners treat each characteristic vector as a point in multidimensional space. The learning algorithm accepts labeled points (in these experiments there are exactly two labels: fault-revealing and non-fault-revealing). The goal of a machine learner is to generate a function (known as a model) that best maps the input set of points to those points' labels.

Our experiments use two different machine learning algorithms: support vector machines and decision trees.

4.3.1 Support Vector Machine Algorithm

A support vector machine (SVM) [2] tries to separate the labeled points via mathematical functions called kernel functions. The support vector machine transforms the point space by using kernel functions and then chooses the hyperplane that best separates the labeled points; for example, in the case of a linear kernel function, the SVM selects a hyperplane in the canonical point space. Once a model is trained, new points can be classified according to the side of the model function on which they reside.

Support vector machines are attractive in theory because they can deal with data of very high dimensionality and they are able to ignore irrelevant dimensions while including many weakly relevant dimensions. In practice, support vector machines were good at ranking the properties by their likelihood of being fault-revealing, so examining the top few properties often produced at least one fault-revealing property. The two implementations that we tried, SVMlight [13] and SVMfu [19], dealt poorly with modeling multiple separate clusters of fault-revealing properties in multi-dimensional space. That is, if the fault-revealing properties appeared in many clusters, these support vector machines were not able to capture all the clusters in a single model. The models did, however, represent some clusters, so the top ranking properties were often fault-revealing.

The results reported in this paper use the SVMfu implementation [19].

4.3.2 Decision Tree Algorithm

A decision tree [17] (or identification tree [25]) machine learner separates the labeled points of the training data using hyperplanes that are perpendicular to one axis and parallel to all the other axes. The decision tree machine learner follows a greedy algorithm that iteratively selects a partition whose entropy (randomness) is greater than a given threshold, then splits the partition to minimize entropy by adding a hyperplane through it. (By contrast, SVMs choose one separating function, but it need not be parallel to all the axes or even be a plane.) Some implementations of decision trees, called oblique, apply hyperplanes that are not parallel to the axis, but the implementation used in this paper does not use oblique decision trees.

A decision tree is equivalent to a set of if-then rules (see Section 6.3 for an example). Decision trees usually are not used to rank points, but only to classify them. (Decision trees can rank using boosting [7] or class probabilities at the leaves [16], but the decision tree implementation used in this paper does not rank points.) The decision tree technique is more likely to isolate clusters of like properties than SVMs, because each cluster can be separated by its own set of hyperplanes.

The experiments use the C5.0 decision tree implementation [18].

5 Experiments

This section describes our experimental evaluation of the Fault Invariant Classifier.

5.1 Subject Programs

Our experimental evaluation of the Fault Invariant Classifier uses twelve subject programs. Eight of these are written in C, and four are written in Java. There are 373 faulty versions of the twelve programs. Figure 6 gives statistics about the programs.

Seven of the C programs were created by Siemens Research [12], and subsequently modified by Rothermel and Harrold [20]. Each program comes with a single non-erroneous version and several erroneous versions that each have one error that causes a slight variation in behavior. The Siemens researchers created faulty versions by introducing errors they considered realistic. The 132 faulty versions were generated by 10 people, mostly without knowledge of each others' work. Their goal was to introduce realistic errors that reflected their experience with real programs. The researchers then discarded faulty versions that failed fewer than 3 or more than 350 of their automatically generated white-box tests. Each faulty version differs from the canonical version by one to five lines of code. Though some of these programs have similar names (print_tokens and print_tokens2, and schedule and schedule2), they are independent implementations of distinct specifications.

	Average			Faulty
Program	Funs	NCNB	LOC	versions
print_tokens	18	452	539	7
print_tokens2	19	379	489	10
replace	21	456	507	32
schedule	18	276	397	9
schedule2	16	280	299	10
space	137	9568	9826	34
tcas	9	136	174	41
tot_info	7	334	398	23
C Total	245	11881	12629	166
Geo	49	825	1923	95
Pathfinder	18	430	910	41
Streets	19	1720	4459	60
FDAnalysis	277	5770	8864	11
Java Total	363	7145	16156	207

Figure 6. Programs used in the experimental evaluation. "NCNB" is the number of non-comment non-blank lines of code; "LOC" is the total number of lines with comments and blanks. The print_tokens and print_tokens2 programs are unrelated, as are the schedule and schedule2 programs.

The eighth C program, space, is an industrial program that interprets Array Definition Language inputs. It contains versions with errors made as part of the development process. The test suite for this program was generated by Vokolos and Frankl [24] and Graves et al. [8].

The FDAnalysis Java program calculates the times at which regression errors were generated and fixed [21]. The FDAnalysis program was written by a single graduate student at MIT, who made and discovered eleven regression errors of his own over the course of 9 weeks of work. He took snapshots of the program at small time intervals throughout his coding process, and thus has available the versions of programs immediately after (unintentionally) inserting each regression error, and immediately after removing it.

The other three Java programs were written by students in MIT's *6.170 Laboratory in Software Engineering* class. Each student submitted a solution, and then, after receiving feedback, got a chance to correct errors and resubmit. We used all solutions such that the initial submission contained errors and the final submission did not. The three programs do not share any common code. Whereas the versions of other programs are related (for instance, by having the same corrected version, in the case of the Siemens programs, or by being snapshots in a single development effort, in the case of the space and FDAnalysis programs), the versions of the 6.170 programs are independent code (albeit written to the same specification), written by a total of 120 authors.

For each program, we used the test suite that was provided with it. Our dynamic analysis (Daikon) tends to be little affected by changes in test suite [11]. Had we used a static rather than a dynamic program analysis, then no execution of the programs would have been required during any part of our experiments.

5.2 Methodology

Our evaluation of the Fault Invariant Classifier implementation uses two experiments regarding automatic recognition of fault-revealing properties, plus a third experiment regarding whether fault-revealing properties help users find errors.

The first experiment uses support vector machines as the machine learner, and the second experiment uses decision trees. Each experiment is performed twice: once on the eight C programs, and once on the four Java programs. The goal of these experiments is to determine whether a model of fault-revealing properties of some programs can correctly identify the fault-revealing properties of another program. Two machine learning techniques — support vector machines and decision trees — train models on the fault-revealing and non-fault-revealing properties of all but one of the programs. The classifiers use each of these models to classify the properties of each faulty version of the last program. We measure the accuracy of the classification against the known correct labeling, determined by comparing the properties of the erroneous version and the version with the error removed (Figure 3).

The third experiment uses human examination of the properties reported by the Fault Invariant Classifier. A fault-revealing property is a side effect of erroneous code, but a user presented with such a property may not be able to use it to locate the error. To assess this issue, we examined a subset of the reported properties; for each one, we used our best judgment to determine whether the property would have led us to the error.

5.3 Measurements

Our experiments measure two quantities: relevance and brevity. These quantities are measured over the entire set of properties, over the set of properties classified as fault-revealing by the technique, and over a fixed-size set.

Relevance [22, 23] is a measure of usefulness of the output. It is defined as the ratio of the number of correctly identified fault-revealing properties to the total number of properties identified as fault-revealing:

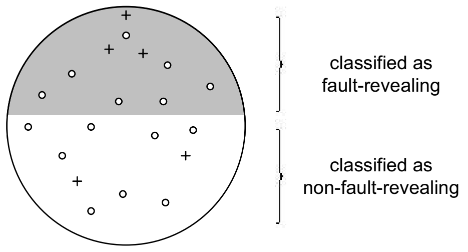
$$\text{relevance} = \frac{\text{correctly identified fault-revealing properties}}{\text{all properties identified as fault-revealing}}.$$

The relevance of a set of properties represents the likelihood of a property in that set being fault-revealing. *Overall relevance* is the relevance of the entire set of all program properties for a given program. *Classification relevance* is the relevance of the set of properties reported as fault-revealing. *Fixed-size relevance* is the relevance of a set of preselected size; we selected 80 because it is the size that maximized average relevance for the C programs.

The brevity of a set of properties is the inverse of the relevance, or the average number of properties a user must examine to find a fault-revealing one. The best achievable brevity is 1, which happens when all properties are fault-revealing. Like relevance, brevity is measured over the overall set of properties, classification set, and fixed-size set.

A related measure is *recall* [22, 23]: the percentage of all fault-revealing properties that are reported to the user. This measure is not relevant in our domain, because it is only necessary to find a single property that indicates an error, not all properties that indicate an error. After fixing the error, a user can rerun the Fault Invariant Classifier to find additional errors.

Figure 7 gives an example of the relevance and brevity measures. In the example, the original relevance is 0.25 and the fixed-size relevance, for a fixed size of 2, is 0.5. The improvement in relevance is a factor of 2.



classified as fault-revealing
classified as non-fault-revealing

	Overall	Classification	Fixed-size (2)
Relevance	$\frac{5}{20} = 0.25$	$\frac{3}{10} = 0.3$	$\frac{1}{2} = 0.5$
Brevity	$\frac{20}{5} = 4$	$\frac{10}{3} = 3.3$	$\frac{2}{1} = 2$

Figure 7. Example of the relevance and brevity measures. Fault-revealing properties are labeled with crosses, non-fault-revealing properties are labeled with circles. The large circle is a 2-dimensional projection of a multidimensional space. The properties in the shaded region are the ones classified by the machine learner as fault-revealing, and the ranking of the properties is proportional to their height (i.e., the property at the top of the shaded region is the highest ranked property).

6 Results and Discussion

Our experimental evaluation shows that the Fault Invariant Classifier effectively classifies properties as fault-revealing. Ranking and selecting the top properties is more advantageous than selecting all properties considered fault-revealing by the machine learner. For C programs, on

Program	Relevance			
		SVMfu		C5.0
	Overall	Class-ification	Fixed-size	Class-ification
print_tokens	0.013	0.177	0.267	0.015
print_tokens2	0.012	0.222	0.050	0.012
replace	0.011	0.038	0.140	0.149
schedule	0.003	0.002	0.193	0.003
schedule2	0.011	0.095	0.327	0.520
space	0.008	0.006	0.891	0.043
tcas	0.021	0.074	0.233	0.769
tot_info	0.027	0.013	0.339	0.190
C Average	0.009	0.010	0.446	0.047
C Brevity	111	100	2.2	21.3
C Improvement	—	1.1	49.6	5.2
Geo	0.120	0.194	0.548	0.333
Pathfinder	0.223	0.648	0.557	0.307
Streets	0.094	0.322	0.690	0.258
FDAnalysis	0.131	0.227	0.300	0.422
Java Average	0.122	0.332	0.586	0.336
Java Brevity	8.2	3.0	1.7	3.0
Java Improvement	—	2.7	4.8	2.7

Figure 8. Relevance results for the Fault Invariant Classifier. The data from each program corresponds to the classifier's output using a model built on the other programs written in the same language. The fixed size is 80 properties. Brevity of a set is the size of an average subset with at least one fault-revealing property, or the inverse of relevance.

average 45% of the top 80 properties are fault-revealing. For Java programs, 59% of the top 80 properties are fault-revealing. Not all fault-revealing properties necessarily directly lead a programmer to the error, but in a preliminary study, most did. Therefore, on average the user only has to examine 3 of the properties (for programs in either language) to be led to an error.

This section is organized as follows. Section 6.1 presents the results of the experimental evaluation. Section 6.2 presents data on user experience with the tool. Section 6.3 compares support vector machine and decision tree machine learning algorithms. Section 6.4 explores what makes some properties fault-revealing.

6.1 Results

Figure 8 shows the data for the recognition experiments.

For C programs, the SVMfu classification relevance (0.010) differed little from overall relevance (0.009). However, the SVM was very effective at ranking: the fixed-size relevance is $\frac{0.446}{0.009} = 49.6$ times as great as the overall relevance for C programs and $\frac{0.586}{0.122} = 4.8$ times as great for the Java programs. The C5.0 classification relevance was $\frac{0.047}{0.009} = 5.2$ times as great as the relevance of all the program properties. For Java programs the improvement was $\frac{0.336}{0.122} = 2.7$ times. Since decision trees can classify but not rank results, fixed-size relevance is not meaningful for decision trees.

While some of the C programs are small, the technique performs best on the largest program, space, suggesting that the technique may scale to large programs. The Java program improvements are smaller than for C programs because there is less room for improvement for the Java programs. The C programs averaged 0.009 relevance before application of the technique, while Java programs averaged 0.122 relevance. Because the student Java programs often contain multiple errors, a larger fraction of their properties are fault-revealing than for the other programs.

Figure 8 reports results for each program, given a model that was trained on the other programs written in the same language. We repeated the experiment using all 11 other programs to train each model. The results were marginally worse than those of Figure 8, suggesting that training on similar programs results in a slightly better model.

The SVMfu machine learner classifies properties in clusters. That is, when ordered by rank, properties are likely to appear in small groups of several similar fault-revealing or non-fault-revealing properties in a row, as opposed to a random distribution of fault-revealing and non-fault-revealing properties. We believe that these clusters form because some fact about the code is captured by more than one property, and if one such property exposes a fault, then all those properties are fault-revealing. The clusters suggest that it may be possible to filter properties by selecting those that lie in clusters.

The clusters indicate that in the absence of such filtering, it is not advantageous for a user to investigate the properties in order of their rank: for each program version, the first few program properties are either all fault-revealing, or all non-fault-revealing. Instead, a user should select properties to examine at random from among a set of the highest-ranked properties. (This is why we report fixed-size relevance rather than the rank of the first fault-revealing property.) The average fixed-size relevance, over all programs, is maximal for a set of size 80 properties. We computed this number by measuring the relevance of each C program version and computing the average for each fixed-size set. Leaving out one program (that under test) did not significantly affect the results, and the Java data is similar.

6.2 User Experience

The recognition experiments indicate that machine learning can identify properties that are fault-revealing. Intuition, and the related work in Section 2, indicate that fault-revealing properties should help users find errors in programs. We have performed a preliminary study to investigate this claim.

We considered all 32 versions of the replace program,

Procedure	Program	Description of error	Fault-revealing property	Non-fault-revealing property
addstr	replace	*maxString* is initialized to 100 but *maxPattern* is 50	*maxPattern* ≥ 50	$lin \neq null$
upgrade_process_prio	schedule	*prio* is incorrectly set to 2 instead of 1	$(prio \geq 2) \Rightarrow return \leq 0$	*prio_queue* contains no duplicates

Figure 9. Sample fault-revealing and non-fault-revealing properties. The fault-revealing properties provide information such as the methods and variables that are related to the error. All four properties were classified correctly by the Fault Invariant Classifier.

and all 9 versions of the schedule program. For each of these independent erroneous versions, we chose at random 10 of the properties reported in the fixed-size set. For each of the 410 properties, we examined it and the erroneous version and judged whether the property would lead a programmer to the error. Such a process is inherently subjective; we tried to be scrupulously fair, but it is possible that we under- or over-estimated a typical programmer's debugging ability, or that the properties would have other benefits in addition to indicating errors.

For the replace program, the Fault Invariant Classifier has a fixed-size relevance of .14, but only .094 of the reported properties would lead a programmer to the relevant error (in our judgment). For the schedule program, the corresponding numbers are respectively .19 and .11. In other words, over all 41 versions, the reported properties are only 65% as effective at leading programmers to errors as is indicated by measuring fault-revealing properties. This suggests that, for our subject programs, a user must examine on average 3, not 2, of the reported properties before locating the error — still a very positive result.

To give the reader an intuition of how fault-revealing properties can lead users to errors, Figure 9 provides examples, from our experiments, of fault-revealing and non-fault-revealing properties for two faulty versions.

One version of the regular expression search-and-replace program replace limited the maximum input string to length 100 but the maximum allowed pattern to only 50. For patterns longer than 50 characters, this version never reported a match. The single difference in the properties of this version and a version with a correctly initialized pattern is that procedure addstr in the faulty version was always called when *maxPattern* was greater than or equal to 50.

As another example, one version of the job scheduling program schedule incorrectly processes a command to increase the priority of a job to 1: instead, the faulty version sets the priority to 2. The fault-revealing property for this program version is that a function returned a non-positive number every time the priority was 2 or greater.

In these examples, the fault-revealing properties refer to the variables that are involved in the error and even the specific constants involved, while the non-fault-revealing properties do not. Thus if a programmer were to examine the fault-revealing properties, that programmer would likely be led to the errors in the code.

6.3 Machine Learning Algorithm Comparison

While decision trees and support vector machines try to solve the same problem, their approaches are quite different. This section compares the advantages of each and discusses boosting.

Support vector machines are capable of ranking properties. Ranking proved more useful than classification. Whereas the classification relevance for SVMfu was only marginally better than the overall relevance, SVMfu was effective at ranking. In particular, for the C programs, selecting the top-ranked properties was 45 times as effective as classification alone; for the Java programs, the improvement was a factor of 1.8.

Decision tree models were able to improve the classification relevance slightly more than the support vector machine models, but because decision trees do not support ranking, it was not possible to optimize the set size using decision trees. However, unlike SVM models, the rule sets produced by decision trees are human-readable if-then rules that may provide insights into the reasons why some properties are classified as fault-revealing (see Section 6.4). For example, one rule produced by a decision tree read "If a property has 3 or more variables, and at least one of the variables is a boolean, and the property does not contain a sequence variable (such as an array), then classify the property as non-fault-revealing."

We attempted to improve the performance of decision trees via boosting [7]. Boosting trains an initial model, and then trains more models on the same training data, where subsequent models emphasize the points that are incorrectly classified by the previous models. During the classification stage, the models vote on each point, and the points' classifications are determined by the majority of the models. In our experiments, boosting had no significant effect on relevance: the resulting models classified more fault-revealing properties correctly, but also misclassified more outliers. We suspect that a nontrivial subset of the training properties misclassified by the original model were outliers, and the overall models were neither hurt nor improved by training additional models while paying special attention to those outliers.

6.4 Important Property Slots

Our technique does not learn specific properties, such as "$x = y$". Rather, it learns about attributes (slots, described in Section 4.2) of those properties, such as being an equality property or involving formal parameters. This permits models generated from one set of programs to be applied to completely different programs.

Some machine learning algorithms, such as neural networks and support vector machines, have predictive but not explicative power. The models perform well but do not yield insight into the (possibly quite complicated) aspects of the problem domain that they successfully capture. This makes it hard to explain exactly why our technique works so well in our experiments. An additional complication is that the models may be capturing a manifestation of some other, harder-to-quantify, quality of the code, such as programmer confusion or inexperience.

As a first step toward understanding the models that our technique produces, we examined decision trees produced by comparing all faulty and non-faulty versions of each single program. While decision trees substantially underperformed support vector machines (for SVMs using ranking and a fixed-size output set), they are human-readable and permit a preliminary examination of what slots appear to be most important.

The following are some if-then rules that appeared most often in the decision trees. (Each decision tree referred to multiple slots; no one slot was sufficient on its own, nor did any of the properties map to specific errors such as use of an uninitialized variable.)

If a property was based on a large number of samples during test suite execution and these properties did not state equality between two integers or try to relate three variables by fitting them to a plane, then that property was considered fault-revealing. In other words, equality and plane-fit properties tended not to be fault-revealing in frequently-executed code.

If a property states that a sequence does not contain any duplicates, or that a sequence always contains an element, then it is likely fault-revealing. If the property does not involve field accesses, then the property is even more likely to be fault-revealing.

If a property is over variables deep in the object structure (e.g., obj.left.down.x), then the property is most likely non-fault-revealing. Also, if a property is over a sequence that contained fewer than two elements, then that property is non-fault-revealing.

7 Future Work

In the experiments, the Fault Invariant Classifier technique accurately classified and ranked properties as fault-revealing and non-fault-revealing. Our preliminary study indicates that fault-revealing properties can help a programmer to locate errors. Validation of both points from larger experiments is necessary. (It is encouraging that we did not need to tune or specially select the programs, properties, or features in order to obtain our results.) It would also be interesting to compare our technique to other techniques such as anomaly detection; we do not know in which circumstances each of the techniques is preferable to the other.

This paper has demonstrated the application of the property selection and ranking technique to aid error location by humans. It may be possible to apply the technique to select properties that are helpful in other tasks. As one example (still related to bug fixing), our technique ignores properties that occur only in the fixed version of a program, but associating those with the fault-revealing properties would provide a before-and-after picture of a fix. When the fix pre-properties apply, the system could suggest the appropriate post-property [14].

Further examination of the models produced, as begun in Section 6.4, may provide additional insight into the reasons properties reveal faults, explain why the Fault Invariant Classifier technique works, and indicate how to improve the grammar of the properties and slots.

The machine learning aspect of this work could be augmented by first detecting clusters of fault-revealing properties in the training data, and then training separate models, one on each cluster. A property would be considered fault-revealing if any of the models classified it as such. This is similar to boosting, which was not effective, but it differs in eliminating outliers from each model rather than adjusting their weight. Another approach is to boost using decision stubs, rather than decision trees.

8 Contributions

This paper presents the design, implementation, and evaluation of a novel program analysis technique that uses machine learning to select program properties. The technique could be summarized as "learning from fixes": the machine learner is trained on pairs of programs where one has an error and the other is a fixed version that corrects the error. (The machine learner operates not directly on the programs, but on an encoding of run-time properties of the programs.) The goal of the technique is to assist users in locating errors in code by automatically presenting the users with properties of code that are likely to be fault-revealing (true of erroneous code but not of correct code). It is a promising result that a machine learner trained on faults in some programs can successfully identify different faults in unrelated programs.

The experimental evaluation of the technique uses a fully automated implementation called the Fault Invariant Classifier. The experimental subjects are twelve programs with 373 errors — 132 seeded and 241 real — introduced by at least 132 different programmers. In the experiments, the 80 top-ranked properties for each program were on average 45% fault-revealing for C programs and 59% for Java programs. This represents a 50-fold and 4.8-fold improvement, respectively, over the fraction of fault-revealing properties in the input set of properties. (The Java improvement is less because the baseline results were better, due to the Java programs containing more errors on average.)

We began an analysis of machine learning models that reflect the important aspects of fault-revealing properties, that may help programmers better understand errors. We also provide some preliminary evidence that links fault-revealing properties to errors in code. A programmer judged which of the reported properties would lead him to an error. The bottom line is that, on average, examining just 3 properties for the subject programs is expected to lead a programmer to an error.

Acknowledgments

We are grateful to Pedro Domingos, Stephen McCamant, and the anonymous referees for their helpful comments.

References

[1] Y. Brun. Software fault identification via dynamic analysis and machine learning. Master's thesis, MIT Dept. of EECS, Aug. 16, 2003.

[2] N. Christianini and J. Shawe-Taylor. *An Introduction To Support Vector Machines (and other kernel-based learning methods)*. Cambridge University Press, 2000.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[4] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, May 2001.

[5] 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.

[6] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[7] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *ICML*, pages 148–156, July 1996.

[8] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM TOSEM*, 10(2):184–208, Apr. 2001.

[9] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN 2003*, pages 121–135, May 2003.

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

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

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, May 1994.

[13] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. J. C. Burges, and A. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, chapter 11. MIT Press, Cambridge, MA, 1998.

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, Nov. 2001.

[15] 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.

[16] F. Provost and P. Domingos. Tree induction for probability-based ranking. *Machine Learning*, 52(3):199–216, Sept. 2003.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[18] J. R. Quinlan. Information on See5 and C5.0. `http://www.rulequest.com/see5-info.html`, Aug. 2003.

[19] R. M. Rifkin. *Everything Old Is New Again: A Fresh Look at Historical Approaches in Machine Learning*. PhD thesis, MIT Sloan School of Management, Sept. 2002.

[20] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, 24(6):401–419, June 1998.

[21] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

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

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

[24] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *ICSM*, pages 44–53, Nov. 1998.

[25] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition, 1992.

[26] Y. Xie and D. Engler. Using redundancies to find errors. In *FSE*, pages 51–60, Nov. 2002.